Codd's self-replicating computer

Tim J. Hutton
http://www.sq3.org.uk
tim.hutton@gmail.com

October 29, 2009

Abstract

Edgar Codd's 1968 design for a self-replicating cellular automata machine has never been implemented. Partly this is due to its enormous size but we have also identified four problems with the original specification that would prevent it from working. These problems potentially cast doubt on Codd's central assertion, that the 8-state space he presents supports the existence of machines that can act as universal constructors and computers. However all these problems were found to be correctable and we present a complete and functioning implementation after making minor changes to the design and transition table. The body of the final machine occupies an area that is 22,254 cells wide and 55,601 cells high, comprising over 45 million non-zero cells in its unsheathed form. The data tape is 208 million cells long, and self-replication is estimated to take at least 1.7×10^{18} timesteps.

Keywords: cellular automata, self-replication, universal constructor, universal computer, evolvability

1 Introduction

John von Neumann pioneered the use of cellular automata (CA) in the 1940's [36]. His design for a two-dimensional self-replicating machine uses 29 states and what has since become known as the von Neumann neighbourhood, with no rotation or reflection symmetry.

Von Neumann was interested in the evolutionary growth of complexity [18]; in how it is possible for machines to make copies of themselves and to make machines more complex than themselves, as happens in nature. His solution was to show that it is possible to make a 'universal constructor': a machine capable of constructing any machine from a set M, where M includes the machine itself and also machines of greater complexity (however measured) [17, 16].

The first implementation of von Neumann's solution was published in 1995 [19, 21]. While the 29-state machine presented was a universal constructor, it was not capable of repeated self-replication. The authors also made a slight modification of von Neumann's CA to allow signals to cross easily, and did publish a self-replicator in this 32-state space.[1] More recently an implementation of a universal constructor capable of self-replication that uses the original 29-state CA has been published [3, 2].

In the 1960's, Edgar F. Codd (who would later invent the relational database) showed that von Neumann's work could be reproduced in a cellular automaton with 8 states and rotation symmetry [5]. He gave a detailed specification of his design but it was far too large to consider implementing at the time, and only very recently has it become possible to check experimentally whether his design would have functioned correctly.

Codd was motivated by the desire to find a simpler CA space that supported universal construction and computation. Later researchers have continued this quest - in 1971 Roger Banks found a 4-state CA with both rotation and reflection symmetry that fulfilled these requirements [1]. Following Smith's demonstration that computation universality implies construction universality [29, 30], attention switched to the requirements for computation universality in simple one-dimensional CA, culminating in the recent proof that even the one-dimensional two-state CA known as Rule 110 is sufficient [6, 38].

In terms of a practical demonstration of self-replication, it was Chris Langton who provided the first example in 1984 [14], creating the field of Artificial Life in the process. His loops are closely based on Codd's CA, retaining the sheathing and signal propagation, and with many of the same states used for the same purposes. His loops are not capable of universal construction or computation but replicate as a special case, carrying their instructions along with them.

Many later researchers have extended Langton's loops, as part of the ongoing effort to recreate the evolutionary growth of complexity that von Neumann was interested in. Several attempts have been made to augment the loops with construction or computation capacity [33, 20], or to make them more evolvable by ensuring mutated versions have a chance of functioning, with different behaviour [27, 26]. In each of these cases the aim is to add back some of the higher functionality that was present in Codd's CA but lost when Langton made his simplification.

The practicality of exploring complex CA was recently given an enormous boost when Golly's implementation [35] of Gosper's hashlife algorithm [10, 24] was extended to support multi-state CA. The hashlife algorithm uses a multi-level quad-tree representation of cells, and remembers the results of previous iterations by storing sub-patterns in a hash table. On CA that contain a great deal of regularity, such as von Neumann's and Codd's, this results in an enormous speed-up over traditional techniques, and running speeds of many millions of timesteps per second are easily achieveable.

Thus for the first time it has become possible to implement Codd's design in full, and to test its correctness. As will be seen, certain aspects of the design appear to be incorrect, and while none of these issues turned out to be show-stoppers, they could have been. In the next section we present some details of our implementation and discuss the problems encountered. Section 3 shows the completed design while section 4 discusses the implications.

[1] It had been claimed (with some force!) [2] that the 32-state design presented in [21] is not capable of self-replication but this is incorrect - a demonstration can be found in Golly [35].

2 Implementation

Before discussing our implementation in detail, we must briefly review the cellular automata that Codd created, as we will refer to many of these concepts later.

Codd's CA has 8 states and uses the von Neumann neighbourhood with rotational symmetry. As in von Neumann's CA, signals travel along wires made of edge-neighbouring cells, permitting right-angled turns and right-angled junctions. Unlike von Neumann's CA, wires in Codd's CA are sheathed and can carry signals in either direction, which is achieved by providing each signal with a trailing zero. Fig 1 shows the evolution of a simple pattern.

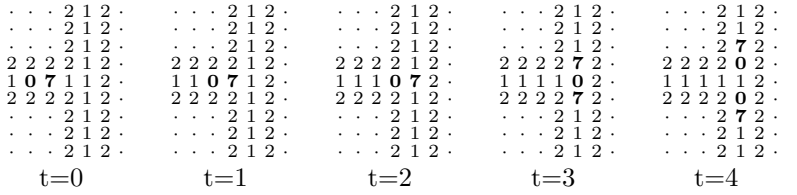


Figure 1: Signals travel along wires made from cells in state 1 sheathed by cells in state 2. Here a 7 signal travels to the right, its direction determined by the trailing cell in state 0. (The background cells are also of state 0 but are shown as dots for clarity.) By time t=4 the signal has been duplicated by the T-junction.

Signals can be blocked by the use of gates, which are formed by signals sent into the side of wires by others. Gates on the right of a travelling signal delete it, while those on the left allow it to pass. Gates can be switched on and off by successive 7 signals.

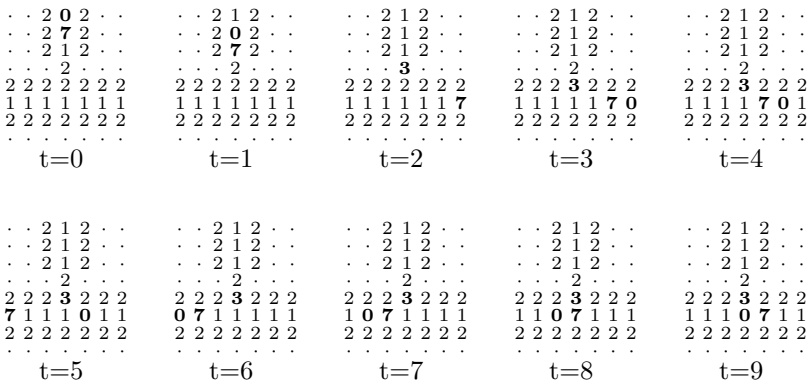


Figure 2: A gate on the top side of a horizontal wire is turned on by the arrival of a 7 signal: a cell in the horizontal wire's sheath adopts state 3. A signal from the right is blocked, while one from the left is allowed to pass.

A simple component that can be made from two gates is the one-way unit, shown in Fig. 3. If a signal arrives from the right, it is first duplicated to turn-on a gate, preventing the signal (which has to travel the longer path on the bottom) from passing through. Signals from the left are not blocked, since the gate lies in the left side of the sheath relative to the direction of travel. A second gate is used to block the path to the first (and itself), to prevent the gates from being turned off by later signals.

In addition to the 7 signal there are three more types: 4, 5 and 6. These signals travel along wires in the same way but play a different role when they reach the end of a wire. For example the signal-train 4-4-5-6 codes for 'extend left' while 7-6 codes for 'extend'. Gates, such as those shown in Figs. 2 and 3, use the end of a wire and so the signal type is important - only a 7 signal can turn-on a gate. Components such as the one-way unit must therefore receive 7 as their first signal type else they

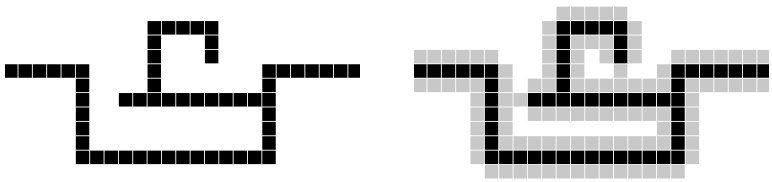


Figure 3: A one-way unit, allowing signals to travel from left-to-right but not from right-to-left. On the left the unit is shown in its unsheathed form, while the version on the right is sheathed. One gate is turned on to block the signals, another is used to prevent later signals from turning the gates off. Black cells have state 1, while grey cells have state 2 and white cells are the background, state 0.

will function incorrectly. After being initialized, the unit works on signals of any type. Since gates are the only way of altering the flow of signals, this constraint has an impact on many components, as we will see in section 2.3.

Additionally the end of a wire can write a cell in state 1 (7-6-4-5-7-6), erase one if it is present (6-7-4-5-6-6) and retract along its own length. By steering a construction path around an initially empty area, any pattern of cells in state 0 and 1 can be written.

2.1 Sheathing

For self-replication, the written pattern of 1's and 0's must first be sheathed, in order for the wires to carry signals correctly. This is achieved using the 6 signal, which traverses the structure sheathing as it goes.

One benefit of the sheathing is shown in Fig. 4. In the unsheathed case (right), when an 'extend left' instruction is received, for example, there is no easy way (with the von Neumann neighbourhood) for cell 'p' to know that it is at the end of the wire and on the left of the end and that it should change state. In the sheathed case (left), however, cell 'p' knows from its 4 neighbours that it lies either on the left of a corner or left of an end, and so from the interaction of a signal with the end of a wire the various actions carried by each signal type can be performed. Alternatives to sheathing for this situation include using a larger neighbourhood, using more states, or weakening the strong rotational symmetry of the transitions. Codd did consider these possibilities but chose sheathing instead.

Figure 4: One reason for the sheathing is that it disambiguates certain situations. At the ends of wires, for example, it helps to distinguish a 'shoulder' cell 'p' from the cell just beyond the end of the wire, or one lying anywhere alongside a wire.

Sheathing is sometimes claimed to have been used by Codd for signal propagation along wires, but Codd himself never mentions this and in fact as Wireworld [9] shows sheathing is not necessary for directed signal propagation, even in a CA space with strong rotational symmetry. Codd does mention that the sheathing assists in providing insulation for the signals during the process of adding to a partially-constructed structure, in addition to the directional benefits mentioned above.

When later researchers managed to dispense with the sheathing [4, 22], it was partly retained in the form of a temporary 'growth cap' that appears at the end of the construction path for the same

reason as seen in Fig. 4 - to disambiguate the shoulder cells. If the weak form of rotational symmetry is used, there is no need for the growth cap [23].

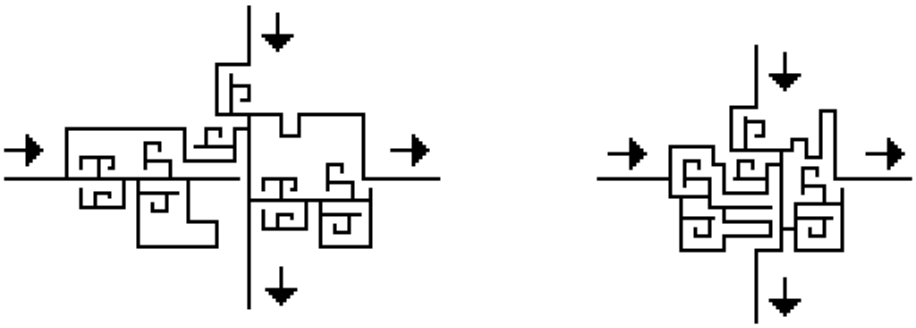
Sheathing in Codd's CA is performed by the traversal of a pattern of 0/1 cells by a type 6 signal. We revisit this issue in section 2.6, where Fig. 11 shows some configurations that are in the middle of being sheathed.

2.2 Crossovers

The design of Codd's machine requires that it be possible for two wires to cross over each other. See Fig. 6 for an example of where crossovers are necessary.

Codd provides two designs for crossover units - one that crosses two unidirectional paths and one that crosses a bidirectional path with a unidirectional path. An implementation of the first type is shown in Fig. 5(a) and occupies an area of 88×54 cells.[2] An implementation of the second type of crossover is shown in Fig. 7(b) (inside the area delimited by labels 'g', 'd', 'e' and 'n') where it is used in the decoder unit.

Both of these designs function in a similar fashion: one-way units are used to prevent signals from exiting along input lines, and ingenious 'turn-off loops' are used to temporarily block one of the remaining output lines. Incoming signals can then safely be allowed to fan-out onto all the 3 other paths and will only exit on the path opposite.



(a) The original crossover unit for two unidirectional paths.

(b) A more compact crossover, for type 7 signals only.

Figure 5: Two implementations of the crossover unit for unidirectional paths.

These crossovers require that the first signal in every input direction is of type 7, as there are gates that are only reached by signals from these inputs. This constraint creates the first of our implementation problems and will be discussed in section 2.3. A further constraint is that incoming signals cannot arrive with too small a time delay between them as they would interfere.

In section 2.5 we will see that a great number of unidirectional crossover units are required within the 'control section' of Codd's machine. It is therefore worth briefly discussing how this crossover can be reduced in size.

One optimization that is possible is to remove the 'type 7' transformers, since we know that only signals of type 7 pass through the control section. Through compaction of the remaining components we obtain a smaller device, at 51×43 cells, shown in Fig. 5(b). It is expected that further small improvements could be made but the unit remains a considerable size.

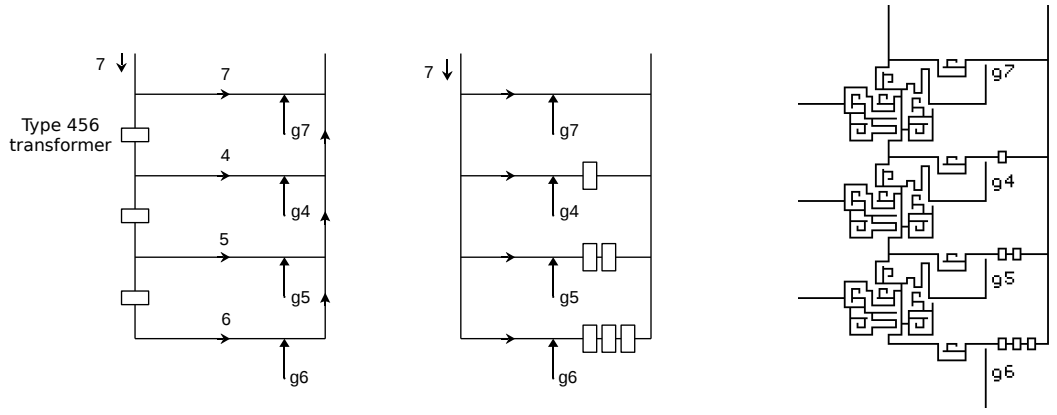
[2]The cell configuration for this crossover was found, credited to Eric S. Raymond, in XLife, an open source software package. This package was very useful in developing our implementation, supplying patterns for the unidirectional crossover as well as many of the other components.

2.3 Problem 1: Initialising crossovers in the coder

Consider Fig. 6.2 (part 3) in [5], reproduced here as Fig. 6(a). The purpose of this unit is to convert a 7 signal arriving on the left-hand wire into one of type 4, 5, 6 or 7, depending on which of gates g_4–g_7 is open. The 'type 456 transformer' is simply a loop of wire as shown, which causes the collision between two signals to produce a new type in the sequence $7 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 6$ (see section 4.12 in [5]).

For signals to reach gate g_4 from the outside requires a crossover on one of the wires depicted in this figure. However the only wires that are available to cross are ones that carry signals 4, 5 and 6, and the crossover unit must be initialized by a 7 signal in both directions or it will fail. Therefore this component cannot function correctly as presented. The same problem affects gate g_5 but not gate g_7, the path to which can safely cross the line above the gate.

Note that for other reasons Codd provides that the first signal generated will be a 7 which could[3] travel downwards on the vertical line on the right of Fig. 6(a), before being blocked. However this doesn't help with the crossover initialization problem: even if the line to g_4 crosses the wire to its right the bidirectional crossover required will only be initialized correctly from two of its three input directions - once from the signal to g_4 and once from the downwards-travelling 7 signal generated by the first call to the coder. The upwards-directed input will still break when it receives the first signal, of type 5 or 6.



(a) The coder as originally presented. Unfortunately gate g_4 is unreachable with the crossover units available.

(b) The replacement coder.

(c) The actual cells of the replacement coder shown with the crossovers in place.

Figure 6: The problem with the coder section and one possible solution.

Several solutions are possible. Perhaps the simplest is shown in Fig. 6(b). Instead of chaining type 456 transformers as in Codd's original design, we have each path code its signal from scratch. This allows crossovers to occur safely on lines that only ever carry signals of type 7. Fig. 6(c) shows the cell configuration for the coder in our implementation. Note that exiting signals will be duplicated at each T-junction and will travel back along the coding channels but are safely blocked by one-way units.

A similar crossover problem initially appears to affect gate M, in Fig. 6.2 (part 4) on page 91 in [5]. The most obvious path to it crosses the (bidirectional) program path but there is no guarantee that the first return signal from the program tape will be of type 7. However here we have a simple solution which is to route the path to M over the marker tape branch first, which Codd ensures will return a 7 signal as the first echo. This requires two bidirectional crossovers instead of one but avoids the initialization problem. An alternative solution would have been to swap the positions of the program and marker tapes but this is less desirable because it would prevent the possibility of delayed starting

[3]With other diagrams Codd states that the (non-gate) arrows indicate where one-way units should be placed but doesn't seem to apply this rule consistently - they appear on lines where one-way units are not required.

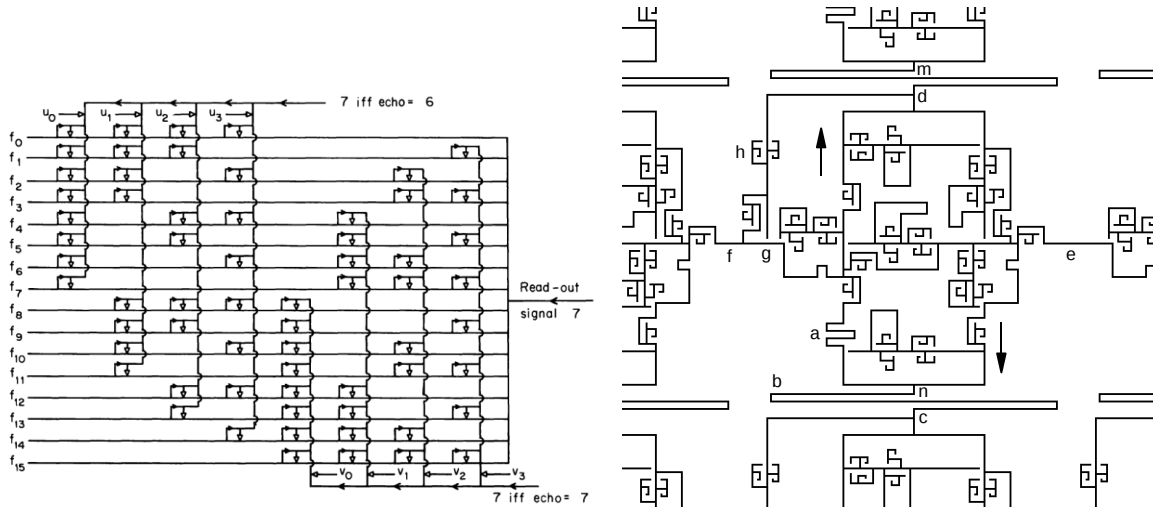
by having some other machine write a 1 to the start of the program tape, as mentioned in section 6.10 in [5].

2.4 Problem 2: Reset signals in the decoder

The second problem is encountered in the decoder, shown in Fig. 5.14 in [5] and reproduced as Fig. 7(a) here. Eight input lines at the top and bottom specify the 0/1 state of 4 bits, opening gates in the 16 read-out lines as appropriate. When a read-out signal arrives from the right, only the read-out line corresponding to the 4-bit integer emits a signal. The role of the decoder is to take the next 4 bits directly from the program tape - the outgoing signal causes that program instruction to be executed.

The read-out signal also resets each gate as it passes, preparing the decoder for the next set of 4 bits. Since only some horizontal lines carried a signal as far as a given column, the vertical lines carry the reset signals both up and down, resetting any gates as needed. Columns that were reached by more than one read-out line may therefore generate multiple reset signals. Codd never made this point explicitly but presumably the intention was that these reset signals would annihilate each other by colliding head-on. In his diagram this looks plausible, since the crossovers are shown with the simple 'jump over' symbol.

In implementation, however, the crossover unit (from Fig 5.11 in [5]) between a bidirectional path and a unidirectional path splits the two directions into separate paths in order to cross each one in turn (see the arrows in Fig. 7(b)). This makes a problem for the decoder since the reset signals cannot be relied upon to collide cleanly - they will often either break the crossover unit, causing spurious signals to be generated, or will pass through each other and reopen gates that had already be reset.



(a) Codd's decoder design, with the crossovers shown with simple 'jump over' symbols. (Adapted from Fig. 5.14 in [5].)

(b) One gate of the decoder in our implementation. One of 16 read-out lines travels from 'e' to 'f' and is gated at 'g' by an input signal arriving on the vertical line.

Figure 7: The decoder unit. At the top and bottom are the 4 input lines that specify whether each of 4 bits is zero or one. These inputs cause the 16 read-out lines to be gated in such a way that only the one corresponding to the integer stored will emit a signal.

If a different design of crossover were available this issue might be avoided but this seems unlikely given the limited set of engineering possibilities for gating. Thus we must find a way to ensure that the collisions between reset signals occur at points known to be safe. To achieve this means that we must control the timings of the signals that arrive at each column: we use delays in the lines and make each crossover identical to ensure predictability.

Fig. 7(b) shows some of these delays. The one marked 'a' ensures that a signal travelling from 'd' to 'n' will take exactly as long as one takes from 'n' to 'd'. The delay marked 'b' ensures that 'cn' is equal to 'dn'. Additional delays on the horizontal lines ensure that all the read-out signals reach each column at the same time.

These arrangements mean that reset signals generated on neighbouring lines will reach 'c' and 'd' at the same moment in Fig. 7(b), and will collide safely at 'n'. This also holds for reset signals generated on lines with an even number of lines between them, the collision will occur at one crossover's 'n' point.

For reset signals generated on lines separated by an odd number of lines, however, the very symmetry of our controlled pattern causes the collision to occur at a crossover's 'd' (or 'c') point. This is a problem if that crossover has a gate[4], since the head-on collision will cause the type of the signal to change from 7 to 4. To correct this we have added an extra component, a 'type 7' transformer shown in Fig. 7(b) at 'h'. When such a transformer receives its first signal (which must be a 7) it creates two gates on either side of the same point on a wire. These gates have the effect that any signal passing through this point will be changed into type 7, whatever its original type. The transformer at 'h' will be initialized along with the gate 'g' at the start of the machine's operation.

Would Codd's design for a decoder have worked without this extra component? If so then we would have to abandon the concept of keeping each line crossover identical, else the odd number situation above will arise. It is conceivable that some set of individual delays could then be tailored to guarantee that the entire set of collisions on all 8 columns over all 16 possible inputs would occur safely but this appears to be very difficult. The delays on the horizontal lines could not be too large, for example, else some gates would get reset before their read-out line had been traversed.

2.5 Control section

Having seen some of the components of Codd's machine it is now time to see how they fit together. Figure 8 shows a simplified overview of the machine. The control section ties everything together, taking a set of inputs and toggling the gates to cause the various actions of the machine to occur. The gates are all initialized to the closed state, so signals from the periodic emitter do not initially spread very far. The s0 signal enters the control section first and sets things in motion.

Inside the control section is simply a set of connections between the control paths in the main machine and the microprogram stack (see Fig. 9(a)). While simple to describe it turns out that this section is by far the largest part of Codd's machine, due to the number of crossovers required.

Codd's microprogram (Fig. 10) expands from 51 steps into 1090 commands, each being either a request to toggle a gate (e.g. g_7, A, E), a program label (e.g. s_3) or a 'goto' command (e.g. 'goto s_4'). In his design the commands are stacked vertically, with inputs and outputs on the right and delay lines on the left (Fig. 6.3 on page 94 in [5]). As implied by Fig. 9(a), numerous crossovers and merge units will be needed to reorganise and consolidate the 1090 commands into the 86 control paths on the right that connect to them.

There are many ways to go about connecting the two sides. One algorithm considered was the 'odd-even transposition sort' [12], a parallel-processing version of bubble sort that resembles a partitioning CA [34]. The paths are taken in pairs, and may be swapped or merged, with the pairing alternating between columns. An improvement over this however is the simpler approach of 'pulling down' a different command with each column: merging all entries of that type into a vertical line (Fig. 9(b)). By taking the most frequent command each time (working from the left), the number of crossovers required is reduced. The extracted commands are then stacked neatly at the bottom, and require no further crossovers - we simply choose to organise the main body of the machine such that the control paths are ordered by approximate frequency of appearance in the microprogram.

The delay lines on the far left of the control section (Fig. 9(b)) are there to ensure that consecutive output signals do not interfere with each other at crossovers as they travel rightwards to the main body of the machine, and do not arrive at the gates out of order. Through experiment we found that a delay of 24,000 timesteps is sufficient to meet these conditions for our implementation of the machine.

[4]One that had been blocked by a previous column, else it would have generated its own reset signal.

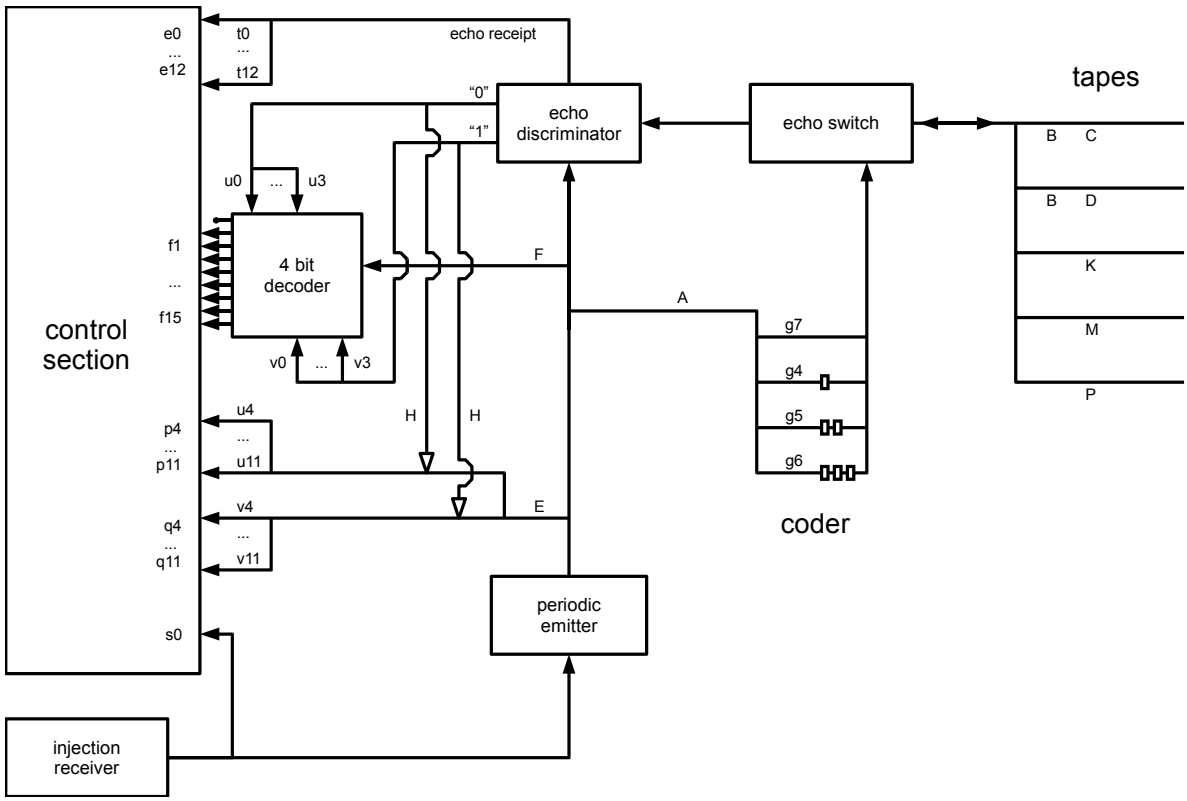
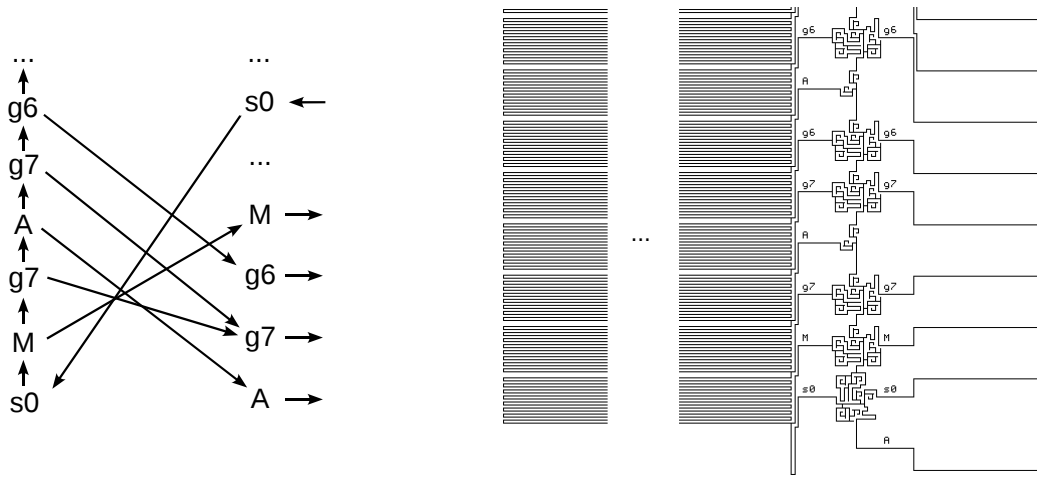


Figure 8: An overview of the components in Codd's machine. Letters beside wires indicate where those wires are gated. After initialization from the injection receiver, the periodic emitter sends regular pulses to the echo discriminator and other controls. These pulses are normally blocked since the gates (A, B, C, D, K, M, P, E, F, H, g_i, t_i, u_i, and v_i) are all closed. The control section directs the opening and closing of these gates (along wires not shown), allowing pulses from the periodic emitter to travel through various parts as required: to send commands and sense requests to the tapes, for example, and wait for responses. The echo that returns from the tapes is passed into the echo discriminator that sends a pulse down "0" or "1" depending on the echo's type. These echos can be stored in the 4-bit decoder, if one of u_0–u_3 or v_0–v_3 is open. If H is open, the echos can also toggle one of the two 'echo storage' gates leading to control inputs p_i and q_i that are shown as hollow arrowheads. Every echo also generates an echo receipt. The echo switch simply ensures that signals from the coder head towards the tapes, while signals from the tapes head towards the echo discriminator.



(a) A sketch of the control section. Each program entry point (e.g. s_0) triggers a sequence of outputs (M, g_7, A, g_7, etc.) by sending a signal that ascends the command list on the left.

(b) The left-most end of the control section in our implementation, showing the command list and the first of 86 columns. In this column the 'A' command is extracted. Between each command is a convoluted delay line, allowing each output time to take effect.

Figure 9: The control section of Codd's machine connects the 86 control lines to a microprogram of 1090 commands. The control section ends up being enormous because of the complexity of wiring so many channels (a) and the bulkiness of the crossovers, seen in (b).

2.6 Problem 3: Collision of sheathing signals

The wire sheathing in Codd's CA is created by the injection of a single signal of type 6. Where the sheathing signal reaches a junction it duplicates, and those duplicate signals will later collide and merge if the wires rejoin. Many collision cases are provided for by Codd's transition table but some are not. None of the cases shown in Fig. 11 will sheath correctly - they all break. The top row highlights the problem collision, while the bottom row shows a real-world example that causes the problem. While these examples are constructed for clarity, these problems were encountered for real in the control section when trying to sheath our implementation of the machine, resulting in breakage in over 30 separate places. In general, given a complex enough network of components every possible collision situation will occur, and there is no obvious method for avoiding the problem cases - introducing a delay in one place will produce different collisions in many other places.

Analysis of the transition table showed that the five situations in Fig. 11 are the only ones that are problematic.[5] We therefore correct for these five problem cases by introducing three new transitions: 006622, 066022, and 166606 (written as: centre, north, east, south, west, new state). The first two deal with the two-signal collision cases and their reflected versions, the last with the three-signal case. With these transitions included, the full machine sheathes correctly.[6] It should be noted that these extra transitions are rarely encountered and thus constitute a very minor change to Codd's ruleset.

There are still many ways for the sheathing to fail. If the wires are too close, or junctions are too close to corners, then the sheathing signals can collide incorrectly and break. However these cases are easily avoided by ensuring sufficient separation of the wires and junctions when designing the components. This constraint imposes a lower limit on the size of the crossover unit we showed in section 2.2, for example.

[5]Note that we don't need to consider 'odd distance' collisions (section 4.12 in [5]) because the sheathing starts at a single location. Four-way junctions are also forbidden in Codd's CA.

[6]The machine also sheathes correctly with just the two-way collision cases but we include the 3-way case for completeness.

s0	M, x, xr, t12, sw
e12	cs, rr, M, D, B, K, P, xr, P, x, K, B, D, P, H, w4, s4
s4	sw
e4	cs, E
p4	s4
q4	H, w4, rr, M, x, M, xr, P, s3
s3	P, w0, sw
e0	cs, rr, M, x, M, xr, w0, w1, sw
e1	cs, rr, M, x, M, xr, w1, w2, sw
e2	cs, rr, M, x, M, xr, w2, w3, sw
e3	cs, rr, M, x, M, xr, w3, P, F
f1	C, s3
f2	D, s3
f3	s3
f4	B, x, B, s3
f5	B, xl, B, s3
f6	B, xr, B, s3
f7	B, r, B, s3
f8	B, rl, B, s3
f9	B, rr, B, s3
f10	B, m, B, s3
f11	B, e, B, s3
f12	H, w5, B, sw
e5	cs, B, E
p5	w5, w6, P, s6
s6	sw
e6	cs, E
p6	w6, rr, M, x, M, xr, P, H, s3
q6	rr, M, x, M, xr, s6
q5	w5, w7, P, s7
s7	sw
e7	cs, E
p7	w7, rr, M, r, P, xr, w8, s8
q7	P, K, r, rr, x, xr, x, m, K, P, rr, M, x, M, xr, s7
s8	sw
e8	cs, E
p8	rr, P, r, P, xr, s8
q8	w8, rr, r, xr, w9, sw
e9	cs, rr, x, xr, E
p9	w9, w8, p8
q9	w9, M, s10
s10	w10, K, sw
e10	cs, E
p10	K, w10, H, M, rr, M, P, xr, P, s3
q10	e, rr, r, xr, x, K, w10, w11, M, s11
s11	rr, P, x, P, xr, sw
e11	cs, E
p11	s11
q11	w11, M, s10
f13	s3
f14	B, i, B, s3
f15	B, j, B, s3

(a) The microprogram, as published in [5], page 101. Where program steps appear on the right (e.g. s4 at the end of the e12 sequence) these indicate the next command to be executed; these are goto commands.

x	G7, G6
xl	G4, G4, G5, G6
xr	G5, G5, G4, G6
r	G4, G5, G6, G6
rl	G5, G6, G6, G6
rr	G4, G6, G6, G6
m	G7, G6, G4, G5, G7, G6
e	G6, G7, G4, G5, G6, G6
i	G7, G5, G6
j	G6, G7, G7, G6, G6
sw	G7, G7
cs	G4, G6
G4	g4, A, g4
G5	g5, A, g5
G6	g6, A, g6
G7	g7, A, g7
w_i	t_i, u_i, v_i

(b) Certain commands are written in short-hand and are expanded as listed here. The first 10 are program tape commands and are explained in section 2.8, while 'sw' stands for sense-and-wait and 'cs' for cap-after-sense.

Figure 10: Codd's microprogram lists a set of gates to be toggled on/off for each input signal received into the control section. For example, s0 is sent when the machine is initialized: it first opens gate M to the marker tape and then sends the command to extend (x). The extend command is expanded *(b)* into the gate sequence: g7, A, g7, g6, A, g6 - opening and closing gates g7 and g6 to allow a pulse of the given type to leave the coder (see Fig. 8). (This starting sequence can be seen in Fig. 9.) Gate A is subordinate-restored (the gate closes automatically after a signal passes through) and so does not need to be closed each time. Gates E, F, and the two echo storage gates (shown as hollow arrowheads in Fig. 8) are also subordinate-restored.

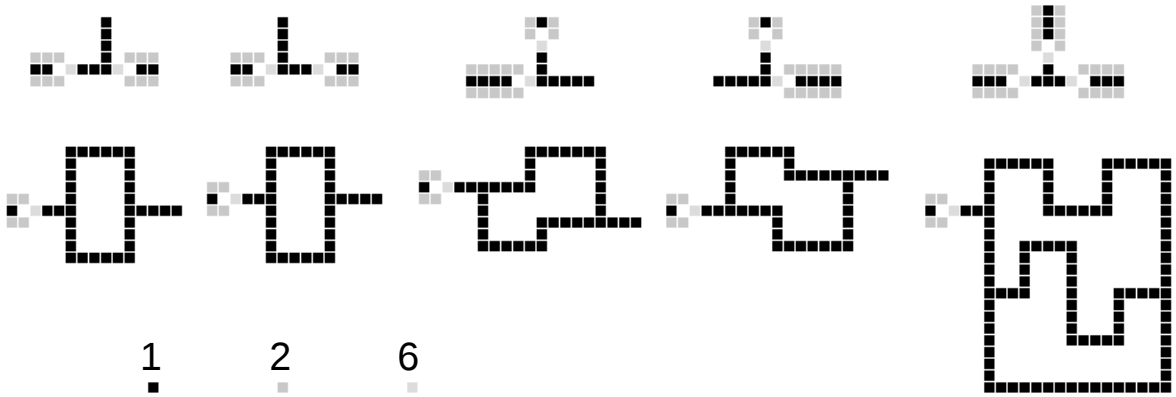


Figure 11: Some examples of sheathing. The sheathing signal (type 6) traverses the core wires (state 1), duplicating and merging as necessary to cover every section. The examples shown illustrate cases where new transitions are needed: without the extra transitions these examples will not sheath correctly.

2.7 Problem 4: Gate t_{12} left open

The gate t_{12} is opened by the first microprogram step, s_0 (Fig. 10). Opening t_{12} allows the first echo receipt to pass along control output line e_{12}, triggering the microprogram step of the same name.

However t_{12} never appears again in the microprogram - gate t_{12} is never closed. The problem with leaving gate t_{12} open is that all later echos will also trigger e_{12}. This causes microprogram steps to be executed out of order and the machine soon breaks. It seems to be a simple oversight (or a printing error) that t_{12} was not closed, since correcting it allows the machine to perform as expected.

There are two obvious solutions. One is to change the program (Fig. 10) to close t_{12} at the beginning of step e_{12}. The other is to make gate t_{12} be subordinate-restored from the e_{12} line it gates. In this implementation we have followed the latter option simply because we had already implemented the control section when this issue was discovered.

2.8 Programming the machine

Codd's machine has five paths (see Fig. 12) - the construction arm (C) and four tapes. The program, marker and counter tapes are used by the machine itself during execution of commands and as such are not available for use by the program. The data tape (D) is intended for use as a linear storage of bits along a half-ray but in fact its writing arm can be steered into any position just as with the construction arm.

There are 14 program commands available: c, d, x, xl, xr, r, rl, rr, m, e, jump_if_one:k, i, j and stop. The first two toggle gates to C and D, to direct the flow of generated signals. The next eight commands direct whichever path is open (either C or D or both) to execute the command given: to extend, extend left, extend right, retract, retract left, retract right, mark or erase. Command jump_if_one:k is a conditional jump: the cell under the current path (exactly one of C or D must be open) is sensed, and if it is 1 then the program control is transferred to the k'th command. If not then program control passes to the next command as normal. Commands i and j inject the sheathing and trigger signals into the new structure, while stop is available to cause the whole machine to halt.

While this is a very limited set of commands, these operations are equivalent to Wang's W-machine [37, 15] and thus Turing-complete.

For construction of the body of the daughter machine the process is simple. After opening the gate to C the right sequence of move and write instructions is sent, to scan the construction arm across the empty space writing cells as necessary.

If we try to copy the whole machine, including its tapes, with this technique however, we hit the

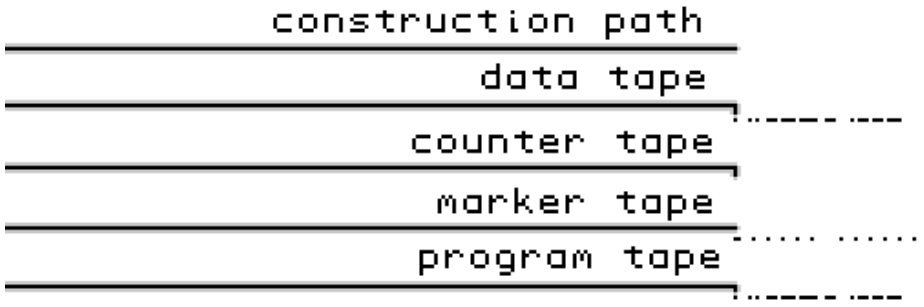


Figure 12: The five paths of Codd's machine, with example tapes. The marker tape (M) contains a mark at the start of each program command (for indexed access, since they are different lengths), with an additional mark at the beginning that is used for rewinding to the start during the conditional jump operation.

problem of self-reference - the tapes would have to contain instructions for constructing themselves and would therefore be longer than themselves. While Codd gives no explicit solution to this problem, the implied answer is to follow von Neumann (and later Langton) by using the tape twice: once as instructions and once as a passive sequence to be copied. Of course Codd doesn't need to provide a solution - his provision of a universal computer with two input/output tapes guarantees that a solution is possible and thus his existence proof is complete. For actually implementing the machine we are therefore free to choose a method.

The program tape, P (together with the marker tape, M) contains instructions to perform three tasks: to copy the data tape (D), to construct the contents of D, and to trigger the new machine. The data tape D contains encoding(B+P+M): an encoding of the body of the machine (B) plus the program and marker tapes. Before triggering, therefore, the new machine will have all the necessary parts: B, P, M and D. The data tape is encoded as in Table 1.

11	write a 1
10	write a 0
01	write N 0's
001	end of row
000	end of construction

Table 1: The data tape contains instructions for constructing the body of the machine, with the encoding listed here. Each row is written as a string of commands 11, 10 and 01, followed by 001 which marks the end of the line. After the last line the sequence 000 indicates the end of the data tape. This encoding scheme is chosen to allow blank areas to be skipped over quickly - a value of N=15 was found to minimise the length of the data tape.

Table 2 contains the program listing. Labels (e.g. retract_until) are used rather than command numbering, for clarity. The program begins by first moving the construction arm into position and copying the contents of D into a new location. We then retract C (but not D) and after moving C to a new location, begin decoding D, reading backwards from the far end. After the (near) end of D is reached (the 000 command is encountered) the program then sheathes and triggers the new machine, retracts from it and stops. The new machine is then capable of self-replication in turn. With further commands the original machine could be made to perform other tasks in addition to self-replication.

Temporary markers are used in several places to allow retracting of the construction arm: a single one above the start of the row being constructed, a double one above the start of the new copy of the data tape, and one to the left of the vertically-extending construction arm for retracting downwards back to the bottom-left corner of the new machine. Other temporary markers are used to turn the

conditional jump command into an unconditional jump (a goto) - the first jump command is of this type and its temporary marker is erased at the beginning of the 'extendup' subroutine. The program assumes that the space being written to is initially empty.

The program takes advantage of several features of the data tape encoding and other constraints. We know that the last two commands on the data tape will be 001 (end of row) and 000 (end of construction). We actually omit the last of these, leaving the last bit of the data tape (at the near end) as 1, which is assumed at the end of the 'extendup' subroutine. During construction, the data tape head will then read three cells past the near end of the data tape into the empty space (see Fig. 14b), reading 000 (end of construction). This trick allows us to be sure that the sequence 000 doesn't appear anywhere on the data tape, which is useful during the copying process for detecting when we have reached the far end. (We know that 10 (write a zero) can't be followed by 001 (end of row) because each row ends on the last non-zero cell, and no row is allowed to be empty.) We also know that the first bit of the data tape (at the far end) is 1 because we ensure that the injection receiver (a plus shape) is located in the lower-left corner of the machine body - so the first command is 10 (write a zero). Knowing this allows us to position the data tape head correctly for the start of construction.

We put the most heavily used parts of the program near the beginning for reasons of speed. Codd's machine uses absolute addressing for jumping to other parts of the program and so must rewind to the beginning and count along each time. Parts of the program that are used only infrequently are placed in subroutines at the end of the program tape. Devore's machine (see section 4.1) and the Perrier loop [20] use a similar programming language but avoid this issue by using relative addressing, with an extra bit used to indicate whether the jump is forwards or backwards.

3 Results

Our implementation of Codd's machine is shown in Fig. 13. It occupies an area of 22,254×55,601 cells, with 45,503,464 of those cells in state 1. The data tape necessary for self-replication is 208, 114, 972 cells long, with a program tape of length 346, 695. Sheathing the machine takes 136,606 timesteps, resulting in a non-zero-cell count of 133,868,322. The complete machine, together with the transition table and tape scripts can be found here: `http://ruletablerepository.googlecode.com/files/Codd-self-rep.zip`

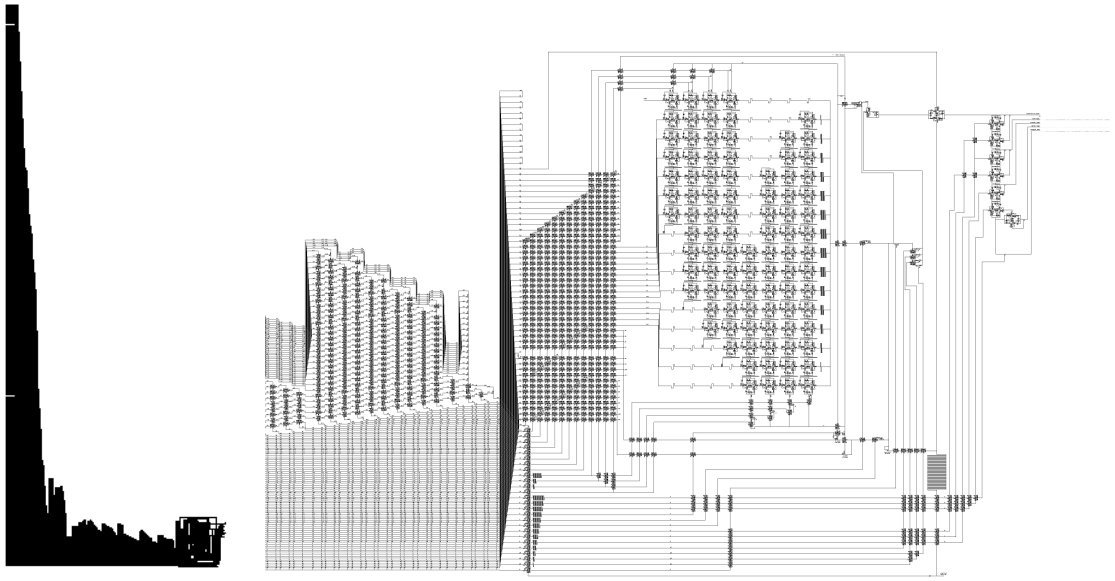
As a demonstration of the correct functioning of the machine, Fig. 14 shows a small pattern being constructed. The example has a data tape of 102 cells and constructs a pattern of 18 cells in 8.9×10^{11} timesteps. This pattern is approximately 2 millon times smaller than Codd's machine and a minimum estimate of the overall replication time is therefore 1.7×10^{18} timesteps. The actual figure is likely to be larger than this because of the time needed for signals to traverse the long tapes. (In the limit this form of replicator runs in $O(n^2)$, where n is the length of the tape but in this case the linear time factor appears to outweigh the quadratic time factor.)

On a 3 GHz PC with 2 GB RAM, the example above took around 5 hours to run in Golly - an average speed of 50 million timesteps per second. At this speed the full machine would take at least 1,000 years to replicate.

One optimisation that is specifically designed with Golly's hashlife implementation in mind is the period of the periodic emitter: we use 128 because as a power of 2 this greatly reduces the number of sub-patterns that must be stored in the hashtable. This reduces the wall-clock time by at least a factor of 10 compared to the worst case, while increasing the generation-count time somewhat.

```
# -- first we jump to a subroutine to prepare for tape copying
    c,m,jump_if_one:extendup,
# ----- retract until marker (used for tape *and* construction) ----
retract_until:
    e,xl,x,jump_if_one:done_retract,r,rl,m,jump_if_one:retract_until,
# ---------- data tape copying phase: ------------------------
# == 1 ==
write1:
    r,rr,d,c,xr,m,rr,d,x,c,xr,x,jump_if_one:write1,
# == 0 ==
    r,rr,c,x,c,xr,x,jump_if_one:write1,
# == 00 ==
    r,rr,c,x,c,xr,x,jump_if_one:write1,
# == 000 ==
    r,rr,c,r,r,d,m,jump_if_one:retract_until,
# ---------- construction phase: ------------------------
nextcolumn:
    e,x,c,d,r,xr,x,jump_if_one:1,
# == 0 ==
    r,rr,r,xr,x,jump_if_one:01,
# == 00 ==
    r,rr,r,xr,x,jump_if_one:001,
# == 000 ==
    m,jump_if_one:doneconstructing,
01:
    r,rr,d,
# -- write N 0s --
    c,[x]*N,m,jump_if_one:nextcolumn,
1:
    r,rr,r,xr,x,jump_if_one:11,
# == 10 ==
    r,rr,d,c,x,m,jump_if_one:nextcolumn,
11:
    r,rr,d,c,xr,m,rr,x,m,jump_if_one:nextcolumn,
001:
    r,rr,d,c,m,jump_if_one:retract_until,
# ------------------------ subroutines (here for speed): -----------------
retractcolumn:    # retract vertically after construction
    e,xl,x,jump_if_one:inject,r,rl,m,jump_if_one:retractcolumn,
done_retract:     # after retraction: decide whether was tape copying or construction row
    e,rl,r,xl,x,jump_if_one:retractdown,r,rl,r,r,rr,x,xr,x,x,x,xl,x,m,r,rl,m,jump_if_one:nextcolumn,
extendup:         # extend the C-arm up to copy the tape
    e,xl,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,x,
    xr,[x]*(X-3),xl,[x]*(Y+6),xr,x,x,x,xl,x,m,r,rl,x,xl,x,m,r,rl,
    c,d,jump_if_one:write1,
retractdown:      # pull the C-arm down after copying the tape
    e,rl,r,r,r,rr,[r]*(Y+6),rl,[r]*(X-3),rr,x,x,x,x,x,x,x,x,
    xl,x,m,r,rl,xr,x,x,x,xl,x,m,r,rl,m,jump_if_one:nextcolumn,
doneconstructing: # end of construction: prepare to retract vertically
    e,x,d,c,xl,x,e,rl,r,r,r,rr,m,jump_if_one:retractcolumn,
inject:           # done retracting vertically: sheath, trigger, retract and stop
    e,rl,r,r,xr,x,i,j,r,r,rr,r,r,r,stop
```

Table 2: The contents of the program tape in our implementation, before being compiled down to a binary sequence of cells. Entries following a # are comments. Entries ending in colons are labels, as jump destinations, and are replaced by command numbering in the final program. Three parameters are used: N=15 is defined in Table 1, while X=22252 and Y=4189 contain the position of the data tape of the new machine, relative to its bottom-left corner.



(a) The whole machine. The control section occupies the left three-quarters.

(b) A closeup of the main body of the machine. The decoder from Fig. 7(a) can be seen in the middle, and the last 17 columns of the control section on the left. The tapes are in the upper-right, while most of the other components are too small to see. Most visible components are crossovers or delay lines.

Figure 13: Our implementation of Codd's self-replicating machine. The machine consists of 45 million cells, requires a data tape of 208 million cells to self-replicate and would take at least 1.7×10^{18} timesteps to do so.

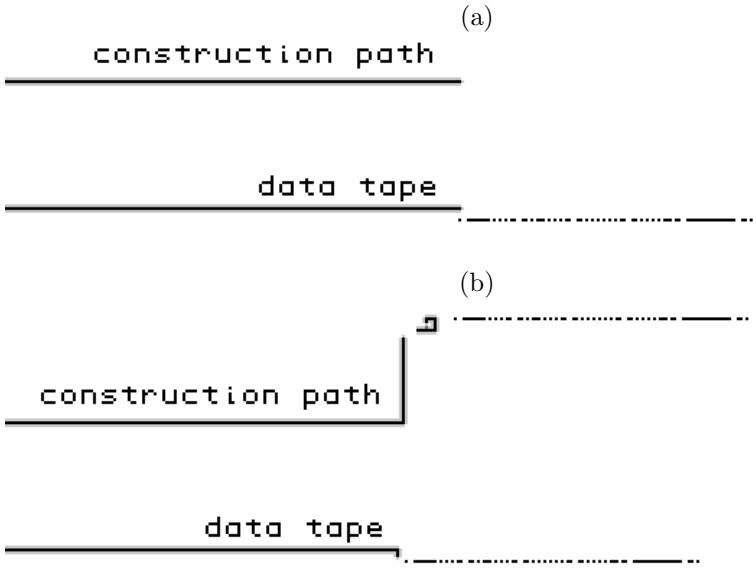


Figure 14: Before and after images of the construction and data tapes in a simple example. After 8.9×10^{11} timesteps *(b)* the data tape has been copied and a small structure has been constructed, sheathed and triggered.

4 Discussion

What is the point of implementing Codd's design? In a sense his design was never meant to be implemented - it was never intended to be a practical demonstration, and the machine has no intended purpose. His design is an existence proof, intended to show that the 8-state CA he creates does support the existence of a universal computer/constructor (UCC). However, without a functioning implementation doubts could linger over whether there was a mistake in his design, and therefore whether the CA space proposed did indeed support a UCC. As we have seen, there are indeed mistakes in the design but in every case these mistakes are correctable (and it is reasonable to think that Codd would have discovered and corrected them himself had more computing power been available at the time). By implementing his design we have therefore finally managed to confirm that his 8-state space is indeed UCC-supportive.

A different method of proof for this fact is available, instead of implementing Codd's complicated machine: show that his CA can simulate some other CA, one known to be construction- and computation-universal. Codd himself uses this approach to show that a 2-state two-dimensional CA with a cross-shaped neighbourhood of 85 cells can simulate his 8-state space and thus is itself UCC-supportive. However, in the absence of such a proof for Codd's 8-state CA itself, we must rely on his complicated machine functioning as intended, as this paper has confirmed is indeed true, given slight modifications.

It is also nice to have a full implementation of Codd's design for other reasons. His CA has been very influential on the field of Artificial Life, and many published self-replicators are derived from it [28]. Aside from any other reason then, making a publically-available working version of Codd's machine is of historical interest, akin to rebuilding the Colossus [25] or the Difference Engine [31] (although of course not as difficult or as significant as these).

As can be seen in Fig. 13(b), our implementation is not minimal or optimal. In layout we have followed Codd's diagrams where possible, to make the machine visually comprehensible when compared with Codd's specification. Therefore there is scope for many components to be rearranged and compacted, with a subsequent reduction in the overall size of the machine and its tapes, and in its replication time. Additionally there are many other optimisations that can be achieved: type 7 transformers are not needed in the decoder crossovers (Fig. 7(b)) and could be removed, and the delay lines (Fig. 9(b)) could be tailored to each command instead of all being the length of the longest required. A different algorithm for the control section wiring could also make a big difference. It is conceivable that these optimisations could reduce the overall size by half but even this would still leave the machine far from being practical to run to completion at the current time.

4.1 Related work

For completeness, we should mention some work that occurred shortly after the publication of Codd's CA. In the 1970's two groups of researchers made variants on Codd's rules, both aimed at reducing the size of the machine. A Hungarian team published a series of papers concerning their variant 'Codd-ICRA' [7, 32], which permitted, amongst other innovations, a 9-cell crossing organ for signals of type 6 and 7. This organ is shown in Fig. 15(a) and would enable the size of Codd's machine to be greatly reduced.

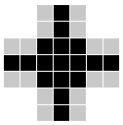
John Devore also worked on a variant, presented in 1973[7] and 1992 [8] but never appearing in a published paper until now it is believed. His design uses a modification of Codd's ruleset to allow tiny diodes[8] and triodes (Fig. 15(b)). His machine (Fig. 15(c)) occupies an area of only 365×258, with a non-zero population count of 19,105 in its unsheathed state. It takes 1.02×10^{11} timesteps to replicate.

John Devore also gave an estimate (reported in [13]) of the size of Codd's machine as at least 10,000×10,000. As a lower bound this is a good estimate, our implementation has now revealed.

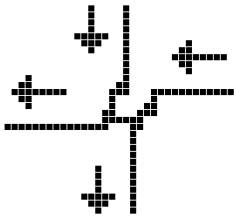
[7] John Devore, personal communication

[8] Suggested by Rod Bates (John Devore, personal communication).

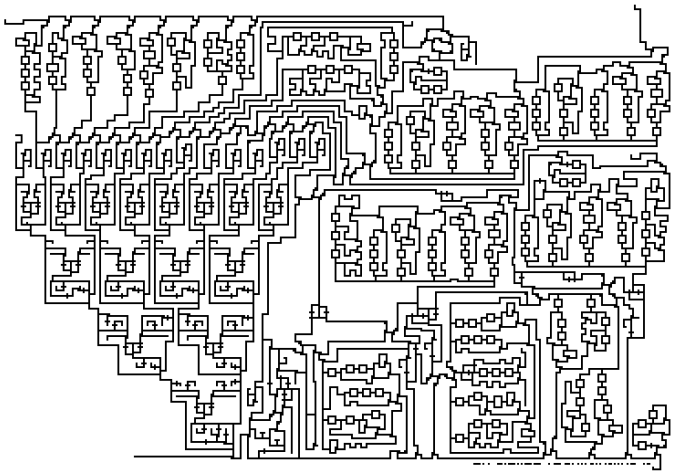
More recently, in 2009 Adam Goucher (Calcyman) showed that it is possible to make a small self-replicator within Codd's original CA (Fig. 15(d)) [11]. This machine is 398×815 and has 21,251 non-zero cells in its unsheathed state. Its program instruction set lacks a branch command and so the machine is not a universal computer but (because the underlying CA is computation-universal) it is capable of constructing one.



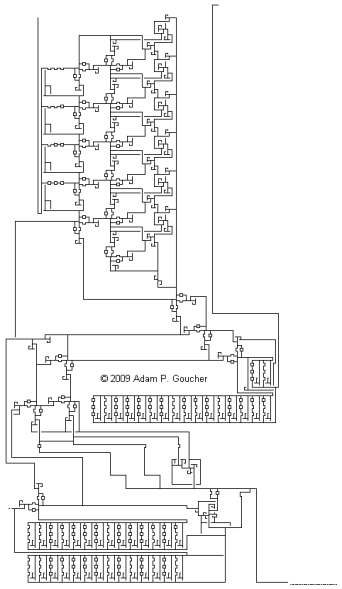
(a) The crossing organ of Dettai [7] for signals of type 6 and 7 on two bidirectional wires in the Codd-ICRA variant ruleset.



(b) The crossing organ of Devore, using 2 × 2 diodes and triodes, for signals of any type on two unidirectional paths in the Devore variant ruleset.



(c) The Devore variant of Codd's machine. A short tape is seen at the bottom-right. The construction arm is at the top left.



(d) Adam Goucher's self-replicating machine.

Figure 15: Two variants of Codd's transition rules were created in the 1970's, by the ICRA group and John Devore. Additionally a small self-replicator (but not a universal computer) has been created in Codd's original ruleset by Adam Goucher.

5 Conclusions

We have identified four separate problems that prevent Codd's machine from working as originally described. Solving these problems has meant making small changes to Codd's design and his transition table. These changes are sufficiently minor that we suggest that the resulting CA conforms to Codd's intentions, if not his exact specification. Our functioning implementation therefore provides confirmation of Codd's central thesis, that his 8-state CA is capable of supporting universal constructors and computers.

Our implementation of Codd's machine can safely be described as enormous, at 45 million non-zero cells in its unsheathed form, or 134 million while running. A full cycle of self-replication has never been achieved but the machine can be run sufficiently far to demonstrate that it is possible.

While the details of Codd's design may no longer be of practical relevance, his approach to implementing computing machines in cellular automata continues to influence Artificial Life research. Additionally, if the search for a new implementation of the evolutionary growth of complexity is to be successful (whether in cellular automata or other media) we will need to continue exploring the middle ground between the simplest self-replicators and those with higher functions as found in the designs of von Neumann and Codd.

6 Acknowledgments

Thanks to Ron Hightower and John Devore for their help in connection with the Devore variant machine. Following their approach of putting a compact encoding on the data tape led to a large improvement in the replication time for our implementation of Codd's machine over an earlier version.

References

[1] Banks, E. (1971). *Information processing and transmission in cellular automata*. PhD thesis, MIT, Department of Mechanical Engineering.

[2] Buckley, W. (2008). Signal crossing solutions in von Neumann self-replicating cellular automata. In A. Adamatzky, R. Alonso-Sanz, A. Lawniczak, G. Martinez, K. Morita, & T. Worsch (Eds.), *Automata 2008*. Luniver Press.

[3] Buckley, W., & Mukherjee, A. (2005). Constructibility of signal-crossing solutions in von Neumann 29-state cellular automata. In V. Sunderam, G. van Albada, P. Sloot, & J. Dongarra (Eds.), *International Conference on Computational Science (2)*, volume 3515 of *Lecture Notes in Computer Science*, (pp. 395–403). Springer.

[4] Chou, H., & Reggia, J. (1997). Emergence of self-replicating structures in a cellular automata space. *Physica D*, 110, 252–276.

[5] Codd, E. (1968). *Cellular automata*. New York: Academic Press.

[6] Cook, M. (2004). Universality in elementary cellular automata. *Complex Systems*, 15, 1–40.

[7] Dettai, I. (1974). Effectivity problems and suggestions concerning Codd-ICRA cellular space. Technical report, KGM ISZSZI. Reported in [32].

[8] Devore, J., & Hightower, R. (1992). The Devore variation of the Codd self-replicating computer. In *Third Workshop on Artificial Life, Santa Fe, New Mexico*. Original work carried out in the 1970s though apparently never published. Reported in [13].

[9] Dewdney, A. (1990). The cellular automata programs that create Wireworld, Rugworld and other diversions. *Scientific American*, 262, 146–149.

[10] Gosper, R. (1984). Exploiting regularities in large cellular spaces. *Physica D*, 10, 75–80.

[11] Goucher, A. Personal communication. Machine available as part of [35].

[12] Knuth, D. (1973). *The art of computer programming, vol. 3 - sorting and searching*. Addison-Wesley.

[13] Koza, J. R. (1994). Spontaneous emergence of self-replicating and evolutionarily self-improving computer programs. In *Artificial Life III*, (pp. 225–262). Addison-Wesley.

[14] Langton, C. (1984). Self-reproduction in cellular automata. *Physica D*, 10, 135–144.

[15] Lee, C. (1961). Categorizing automata by W-Machine programs. *Journal of the Association for Computing Machinery*, 8(3), 384–399.

[16] McMullin, B. (1992). *Artificial knowledge: An evolutionary approach.* PhD thesis, Ollscoil na h'Eireann, The National University of Ireland, University College Dublin, Department of Computer Science.

[17] McMullin, B. (1993). What *is* a universal constructor? Technical Report bmcm9301, School of Electronic Engineering, Dublin City University.

[18] McMullin, B. (2000). John von Neumann and the evolutionary growth of complexity: Looking backwards, looking forwards... *Artificial Life*, 6(4), 347–361.

[19] Nobili, R., & Pesavento, U. (1996). Generalised von Neumann's automata I: a revisitation. In E. Besussi & A. Cecchini (Eds.), *Artificial Worlds and Urban Studies*, Venezia: DAEST.

[20] Perrier, J.-Y., Sipper, M., & Zahnd, J. (1996). Toward a viable, self-reproducing universal computer. *Physica D*, 97, 335–352.

[21] Pesavento, U. (1995). An implementation of von Neumann's self-reproducing machine. *Artificial Life*, 2, 337–354.

[22] Reggia, J., Armentrout, S., Chou, H., & Peng, Y. (1993). Simple systems that exhibit template-directed replication. *Science*, 259, 1282–1287.

[23] Reggia, J., Chou, H.-H., & Lohn, J. (1998). Cellular automata models of self-replicating systems. *Advances in Computers*, 47, 141–183.

[24] Rokicki, T. (2006). An algorithm for compressing space and time. *Dr Dobb's Journal*, April.

[25] Sale, A. (2005). The rebuilding of Colossus at Bletchley Park. *IEEE Annals of the History of Computing*, 27(3), 61–69.

[26] Salzberg, C., Antony, A., & Sayama, H. (2004). Complex genetic evolution of self-replicating loops. In J. Pollack, M. Bedau, P. Husbands, T. Ikegami, & R. Watson (Eds.), *Ninth International Conference on Artificial Life*, (pp. 262–267).

[27] Sayama, H. (1999). A new structurally dissolvable self-reproducing loop evolving in a simple cellular automata space. *Artificial Life*, 5(4), 343–365.

[28] Sipper, M. (1998). Fifty years of research on self-replication: an overview. *Artificial Life*, 4, 237–257.

[29] Smith, A. (1969). *Cellular automata theory.* PhD thesis, Digital Systems Laboratory, Stanford University, Stanford, CA.

[30] Smith, A. (1992). Simple nontrivial self-reproducing machines. In C. Langton, C. Taylor, J. Farmer, & S. Rasmussen (Eds.), *Artificial life II*, (pp. 709–725), Redwood City, CA: Addison-Wesley.

[31] Swade, D. (2005). The construction of Charles Babbage's Difference Engine No. 2. *IEEE Annals of the History of Computing*, 27(3), 70–78.

[32] Takács, D. (1977). A maximum-selector design in Codd-ICRA cellular automaton. *Acta Cybernetica*, 3, 107–143.

[33] Tempesti, G. (1995). A new self-reproducing cellular automaton capable of construction and computation. In F. Morán, A. Moreno, J. Merelo, & P. Chacón (Eds.), *Proc. Third European Conference on Artificial Life*, (pp. 555–563). Springer-Verlag.

[34] Toffoli, T., & Margolus, N. (1987). *Cellular automata machines - a new environment for modelling.* Cambridge, MA: MIT Press.

[35] Trevorrow, A., & Rokicki, T. Golly - a Game of Life simulator. `http://golly.sourceforge.net`.

[36] von Neumann, J. (1966). *Theory of self-reproducing automata.* Urbana: University of Illinois Press. Edited and completed by Arthur Burks.

[37] Wang, H. (1957). A variant to Turing's theory of computing machines. *Journal of the Association for Computing Machinery*, 4, 63–92.

[38] Wolfram, S. (2002). *A new kind of science.* Wolfram Media, Inc.