On the Turing Completeness of Eeny, Meeny, Miny, Moe

Javier Lim

Unemployment Agency

Abstract. We demonstrate that an infinite number of people playing Eeny, Meeny, Miny, Moe (EMMM) is Turing Complete, via a reduction from the Rule 110 cellular automata. Specifically, we provide a scheme to simulate any finite elementary cellular automaton as a game of EMMM. For N cells to be simulated, $O(N)$ EMMM players are needed, and each generation takes $O(N)$ time to step.

Keywords: Eeny, Meeny, Miny, Moe · Tiger Catching · Turing Completeness

1 Introduction

Eeny, Meeny, Miny, Moe (EMMM) is a traditional counting-out rhyme used to pseudo-randomly select a person or object from a group. In the context of children's games, EMMM can be used to select the "it" person for a game of tag, or the seeker in hide-and-seek.

One person (termed the counter) points to elements from the set to be chosen while chanting the rhyme. Each time a syllable or word is said, the counter moves to the next element. The selected element is the one that is pointed to at the end of the rhyme. A common 20-word variation of the rhyme is as follows:

Eeny, meeny, miny, moe,
Catch a tiger by the toe.
If he hollers, let him go,
Eeny, meeny, miny, moe.

In this paper, we prove that EMMM is Turing Complete via a reduction from the Rule 110 cellular automata.

2 EMMM

2.1 Variations

Let us explore the variations of EMMM that exist. While there exist the modern 20-word version, there was also a 16-word version, from around 1815 [4]:

Hana, man, mona, mike.
Barcelona, bona, strike.
Hare, ware, frown, vanac.
Harrico, warico, we wo, wac.

These variations are important, as they highlight how language evolves over time. ~~Also, I couldn't figure out how to make EMMM Turing Complete without it.~~

2.2 Definition

Let us now formalise the game of EMMM that we shall use. Each player has 3 components:

1. A list of players in the group, also known as the cycle.
2. Which player of the group they are currently pointing to.
3. EMMM variation, the number of times the next counter should count.

For instance, Alice, Bob, and Charlie are choosing who between them to be the seeker in a game of hide and seek. Alice is designated the counter. Suppose she begins by pointing at herself and uses the common 20-word version of EMMM. She counts in the cycle: Alice, Bob, Charlie. In this situation, she will end on Charlie:

initial	Eeny	meeny	miny	moe		
Alice	Bob	Charlie	Alice	Bob		
Catch	a	tiger	by	the	toe	
Charlie	Alice	Bob	Charlie	Alice	Bob	
If	he	hollers	let	him	go	
Charlie	Alice	Bob	Charlie	Alice	Bob	
Eeny	meeny	miny	moe			
Charlie	Alice	Bob	Charlie			

In children's games, Alice could instruct Charlie to take on the role of "it", or the seeker. However, in this setup, there are no other games than EMMM. Hence, Alice instructs Charlie to become the next counter. Charlie begins a new game of EMMM, continuing the chain.

However, there are multiple variations of EMMM. Which should be used? This choice will be made by the previous counter, who not only instructs the player to start EMMM, but also how many times to count. Hence, Alice instructs Charlie to become the counter and count, say 20, times. Note that this means that depending on who was the previous counter, the same counter can count a different number of times.

One point to touch upon is that each person's pointer is maintained throughout the run of the whole game. For instance, Alice points to Charlie on the last count. If Alice were to become the counter again, she would begin with Charlie, not herself. 20 counts later, Bob becomes the next counter.

In this scenario, Charlie has no cycle, hence cannot play EMMM. This can be interpreted as the EMMM game halting at Charlie.

3 Elementary Cellular Automaton

An elementary cellular automaton is a one dimensional cellular automaton. That is, a row of cells, each in one of two states (0/1, off/on). On the next generation, all cells simultaneously change their state. The new state is solely determined by it's own state, and the state of its immediate neighbors [2].

For instance, if the rules are:

Neighborhood	Next State
111	_0_
110	_1_
101	_1_
100	_0_
011	_1_
010	_1_
001	_1_
000	_0_

And boundaries are assumed to be 0 padded, then the state "0100110" becomes "1101110" after one generation.

3.1 Rule 110

The above table describes Rule 110 (Wolfram's notation, decimal for 0b01101110). Interestingly, it can be shown to be computationally universal [1]. Hence, if we can simulate Rule 110 within a game of EMMM, we have shown that EMMM too, is universal, via a proof by reduction.

4 Reduction from Rule 110

4.1 Notations

We will use the term "signal" loosely to mean a game of EMMM occurring. For example, Alice passing the counter role to Charlie is a signal from Alice to Charlie.

Let us introduce the following diagram notation for specifying arrangements of EMMM players:

An unfilled circle represents a player. The current counter is represented by a filled circle. A number in a player represents how many times they will instruct the next counter to count. By default, we will assume it to be 1.

A colored number next to a player represents that the corresponding color's player has this player in their cycle, at that index. The index starts at 0. An arrow starting from a player, A, and ending on another player, B, represents that A is currently pointing at B.

When a counter counts, they move from the currently pointed index to $(current + count) \mod cycle\ length$.

For instance, the original Alice, Bob, Charlie situation could be represented as Figure 1.

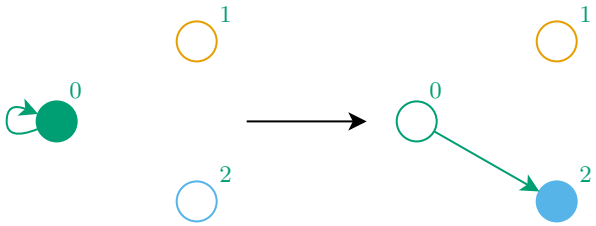

Fig. 1. Alice is the current counter, hence is a filled circle. She starts pointing at herself. Bob and Charlie are indexes 1 and 2 respectively. When Alice counts 20, it will land on $20 \mod 3 = 2$, Charlie. Charlie then becomes the next counter.

4.2 One Bit Memory (OBM)

Let us consider the arrangement of players in Figure 2.

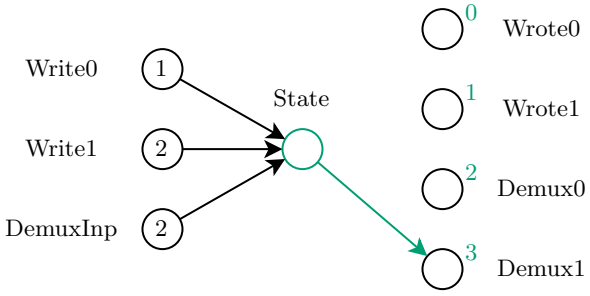

Fig. 2. An arrangement that acts as a single bit-memory. Write0, Write1 and DemuxInp only have State in their cycle, hence color and indexing are not needed.

It acts in a fashion similar to a single-bit memory. From the initial arrangement, if Write0 is the counter, it passes along the signal, ending at Wrote0. If DemuxInp later becomes the counter, the signal will be passed to Demux0.

Had Write1 been the first counter, the first signal would have ended at Wrote1. Then, When DemuxInp later becomes the counter, the signal ends at Demux1.

This arrangement allows the data (input at Write0 or Write1) to be stored without losing information (output at Wrote0 or Wrote1), then read out again later (output at Demux0 or Demux1). This "selection" is similar to demultiplexing in traditional circuits.

While an EMMM rhyme with 1 word seems unrealistic, there likely exists a 17 or 21 word rhyme, which is the same modulo the cycle length.

4.3 Resettable One Bit Memory (ROBM)

The main issue with the previous arrangement is that it is only able to be used once. The state counter will be moved to an unknown position (2 or 3), hence is difficult to work with. However, we may simply add a "Reset" circuit, as seen in Figure 3.

When State points to any of the OBM outputs (Wrote0, Wrote1, Demux0, Demux1), a signal at ResetInp will cause State to point to a corresponding reset player. This reset player simply instructs State to advance to ResetOut. Even if State is already pointing to ResetOut, it will go to RResetOut, which returns to ResetOut.

Hence, we can use ResetInp to reset the arrangement back into its original configuration. This is very useful, since it means that we can reuse the component.

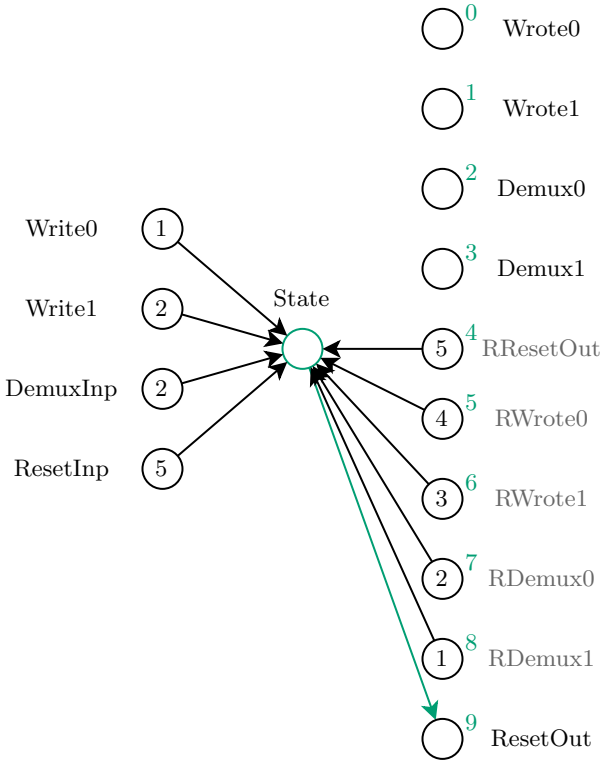

Fig. 3. A OBM with added reset circuit.

We can further abstract this arrangement. We represent the ROBM in a "chip" diagram, only caring about the exposed inputs (Write0, Write1, DemuxInp, ResetInp) and exposed outputs (Wrote0, Wrote1, Demux0, Demux1, ResetOut). We can further assume that signals coming in and out of the chip will use a count value of 1.

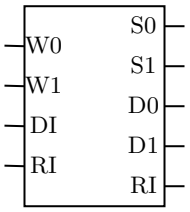

Fig. 5. Chip Representation of ROBM.

We may then use arrows from the output pins (S0, S1, D0, D1, RI) to other input pins (W0, W1, DI, RI) to indicate that the output pin player should pass on the counter role to the input pin player.

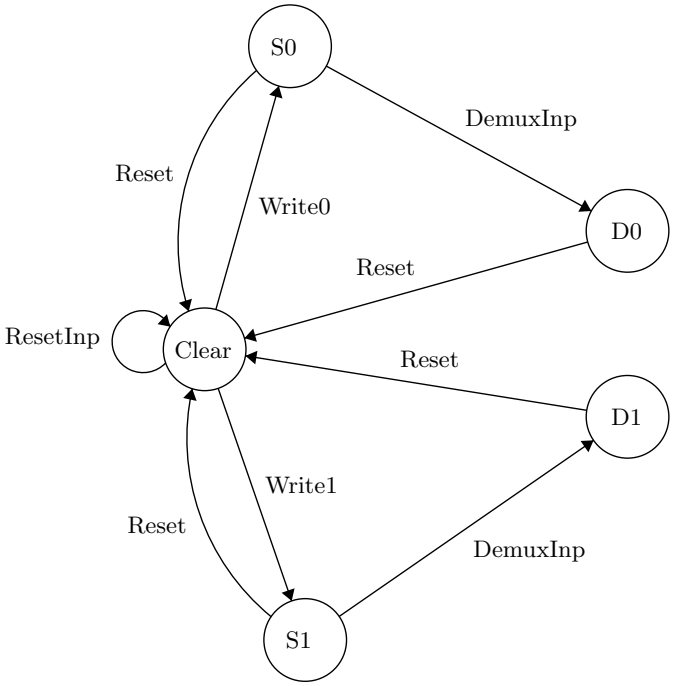

Fig. 4. ROBM Finite State Machine. Wrote is abbreviated to S, for "Store".

4.4 Elementary cellular automaton

Cells We may encode the state of the Rule 110 board as a line of ROBMs, each holding either a 0 or 1 (State player pointing at S0 or S1).

Neighbor Updates To perform neighbor updates, we can take one input query signal, and demultiplex it three times, once for each cell that influences the next state. Each of the 8 possible outputs has an associated next state, determined by the rule we are following. We may use this signal to write data accordingly.

Since the signal branches each time, we need multiple ROBMs per step. For instance, demultiplexing 3 times requires 1 ROBM for the first selection. Then, 2 ROBMs, one connected to each possible output. Then, 4 ROBMs for each of those possible outputs. This is illustrated in Figure 6.

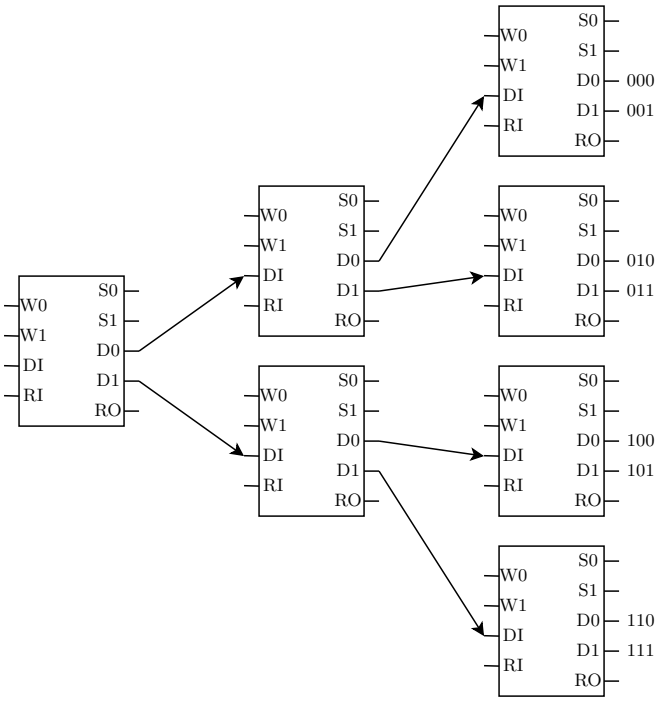

Fig. 6. Chained ROBMs to demultiplex 3 bits into one of 8 output paths. ROBMs in the same column contain the same data. Each column corresponds to the state of a cell.

Once the signal has been demultiplexed, we can apply the rules accordingly. For Rule 110, the 000 pin in figure 6 would be connected to the Write0 pin of the state ROBM, 001 to Write1, 010 to Write1, 011 to Write0, etc.

Hence, for each cell, we not only need the initial ROBM to store its state, but also 7 more ROBMs to perform selection of the rules. We can easily synchronise their states by linking their write and reset lines.

Full execution order With the state of the Rule 110 board in a line of ROBMs (the state ROBMs), and the block of chained ROBMs for demultiplexing (the rule ROBMs), the execution order is as follows:

1. Read from each state ROBMs, writing to the associated rule ROBM
2. Reset all state ROBMs
3. Perform selection on the rule ROBMs to determine the next state. Write to the state ROBMs
4. Reset all rule ROBMs
5. Go back to 1

Due to their large size, diagrams for each step are provided at the end of the section (Figures 7 to 10).

Crucially, across all steps, no pin is used more than once. There may be multiple arrows pointing into one input pin, but no output pin has more than one arrow coming out of it. This means that connections needed for all 5 steps can coexist, without the need for additional circuitry.

Edges The edges of the board require special handling, since they are missing a neighbor to read from. It is common to treat the boundaries as 0s, which is done by skipping one step of the demultiplexing. It is also possible to wrap the board around, which can be done by connecting the edge rule ROBMs together.

4.5 Analysis of execution

Each simulated Rule 110 cell uses 8 ROBM components, each of which use 15 EMMM players. Hence, for N cells, $120N \in O(N)$ EMMM players are needed.

The calculation of a new generation requires the signal to pass through each ROBM component at least twice (write, reset), and at most thrice (write, read, reset) . Hence, for N cells, $O(N)$ time is needed to step to the next generation.

4.6 Diagrams

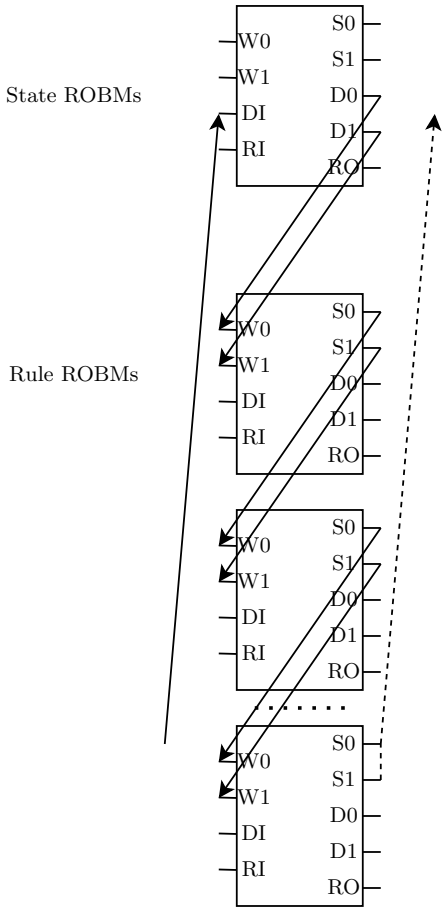

Fig. 7. Step 1: Read from the state ROBM, write into the 7 rule ROBMs, repeat for the next cell.

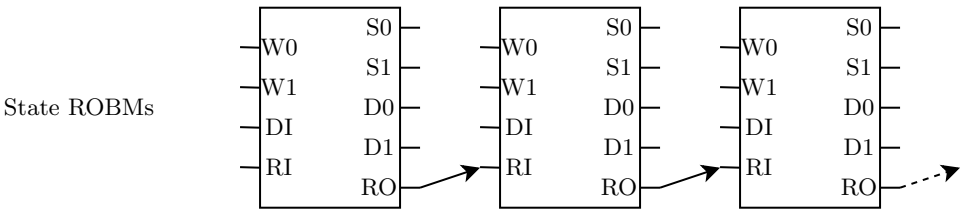

Fig. 8. Step 2: Once the last cell has been read from, and rule ROBMs written to, reset all state ROBMs.

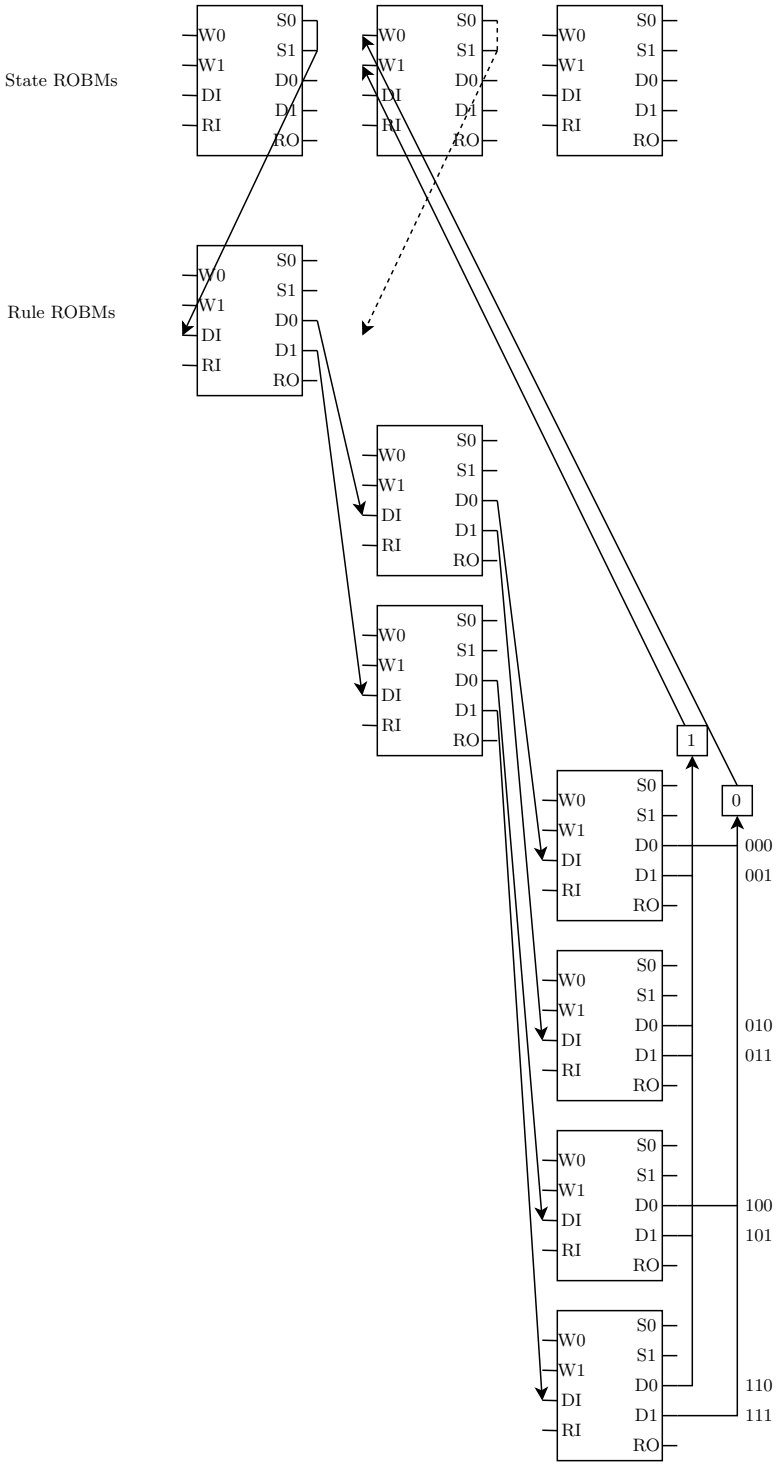

Fig. 9. Step 3: Once the last state ROBM has been reset, demultiplex to figure out what neighborhood each cell has. From there, write the new state as given by the ruleset, and repeat for the next cell.

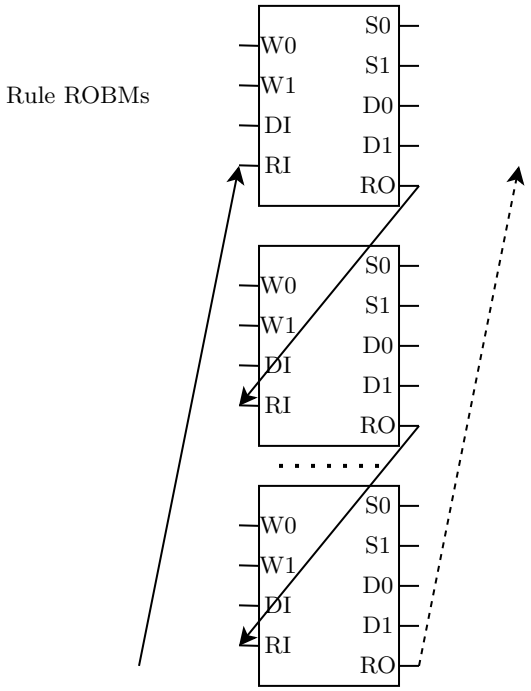

Fig. 10. Step 4: Once the new generation has been calculated, reset all rule ROBMs.

4.7 Implementation

A *very* messy implementation of Rule 110 simulated on EMMM players is available at `https://blog.javalim.com/2023-Eeny/`. It also contains a work in progress esolang, Eeny, designed for EMMM based computation.

5 Conclusion

In conclusion, we have demonstrated an arrangement of Eeny, Meeny, Miny, Moe players that is able to simulate the evolution of the Rule 110 cellular automaton. Rule 110 itself is universal, hence by reduction, EMMM too, is universal. The overhead needed to execute Rule 110 is linear.

qed. i think.

6 Implications and Future Work

6.1 Implications

EMMM is, cryptographically speaking, a bad pseudorandom number generator (PRNG). However, as it is Turing Complete, better PRNGs can be implemented,

such as the Mersenne twister [3], or as demonstrated, cellular automata based PRNGs. Millions of children can now rest easy, knowing that their game of tag was not match fixed.

6.2 Multithreading

Multithreading can be accomplished by having multiple starting counters. One thing to deal with would be what to do in the event that multiple counters instruct the same person at the same time. Perhaps another children's game, Rock-Paper-Scissors, can be used to resolve the conflict. This opens the door to non-deterministic multithreaded computation, which sounds pretty fancy indeed.

6.3 Other reductions

It might be possible to prove its Turing Completeness more directly, that is to say, simulate a Turing machine, not a model of computation known to be Turing complete. This would enable more efficient computation with EMMM.

If this has been done in the future, it will be updated in the Eeny repository.

7 Acknowledgements

Diagrams created with `https://diagrams.net` and `https://madebyevan.com/fsm/`.

References

1. Cook, Matthew. (2004). Universality in Elementary Cellular Automata. Complex Systems. 15.
2. Wolfram, S. (1983). Statistical mechanics of cellular automata. Rev. Mod. Phys., 55(3), 601–644. https://doi.org/10.1103/RevModPhys.55.601
3. Matsumoto, M., & Nishimura, T. (1998). Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Trans. Model. Comput. Simul., 8(1), 3–30. https://doi.org/10.1145/272991.272995
4. I. Opie and P. Opie, The Oxford Dictionary of Nursery Rhymes (Oxford: Oxford University Press, 1951, 2nd edn., 1997), pp. 156-8.