Self-Implementing Rules for Cellular Automata

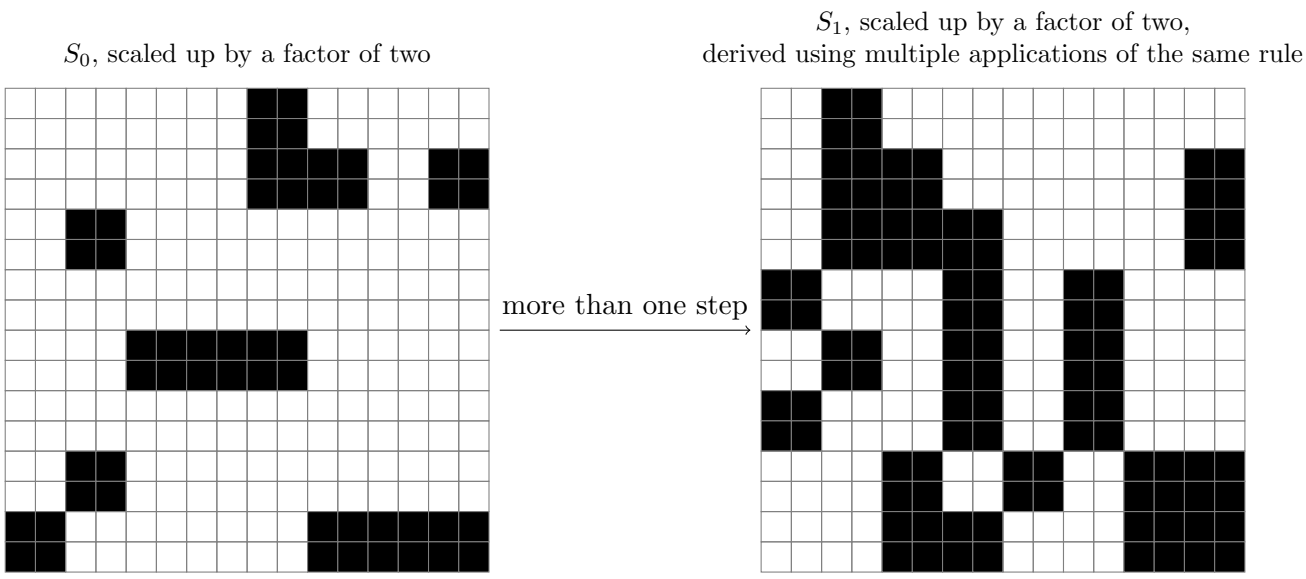
Samuel B Reid

2025

A few months ago, I came up with the idea of a certain class of cellular automata with intriguing behavior. I was not sure if this class had any nontrivial members, but I did eventually discover that it does. Here I will describe the problem and the process that lead me to its solution.

The Problem

I wanted to find a cellular automaton rule on the square grid with the property that, for arbitrary starting states, evolution of the state in accordance with the rule would proceed identically (with some time delay) if the entire starting state were scaled up by a factor of two. This problem description is probably not very clear, so here is a graphic illustrating what I mean.



Motivation

I liked this problem because it had a pleasant, self-referential quality to it in that the cellular automaton implements its own rule at a higher scale. Another nice and perhaps not immediately obvious thing about cellular automata of this type is that being able to scale up by a factor of two in the way described above also unlocks the ability to scale up by any power of two while maintaining the same sort of behavior. This can be proven inductively; If some rule perfectly implements itself at a higher scale then the higher scale implementation will also perfectly implement the higher scale version of itself. So arbitrarily large self-simulation is available.

Trying Approaches

The first thing I considered was brute-force search. However, for a two-state cellular automaton on the Moore neighborhood, there are 512 possible states so 2^{512} possible rules. This space is quite a bit larger than what my laptop (or any computer humanity might ever construct) can reasonably search exhaustively. So I tried to make the space artificially smaller by enforcing various symmetries (for instance, that the rule has to be the same for all states of the Moore neighborhood that are equivalent under reflection or rotation). This still did not make the size of the problem small enough for brute force search. I was able to search the entire space of two-state rules on the smaller Von Neumann neighborhood, but I didn't find anything nontrivial (trivial rules with the desired property include those where the state never changes or where all starting states proceed immediately to the same terminal state).

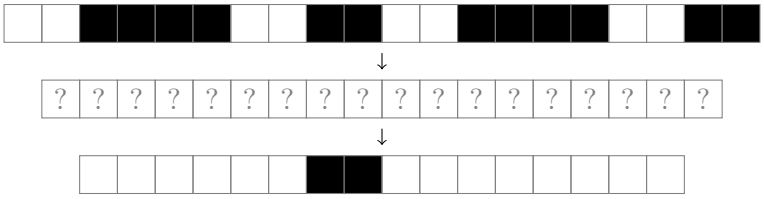
I then realized that starting with a two-dimensional cellular automaton was probably overkill. One-dimensional cellular automata can also have interesting properties, so I decided that I should try to find one-dimensional cellular automata with the property I desired and then see if this clarified things for the two-dimensional case.

I started off by considering two-state cellular automata in which a cell in the next generation determines its state from its previous state and from the previous states of its neighbors. There are only 8 possible parent states so only 256 possible cellular automata of this type. One could try out all of them by hand, if one were only moderately insane. I didn't actually search all of these cases (computationally or manually) because I had an interesting (and ultimately fruitful) idea that I was able to explore using pencil and paper.

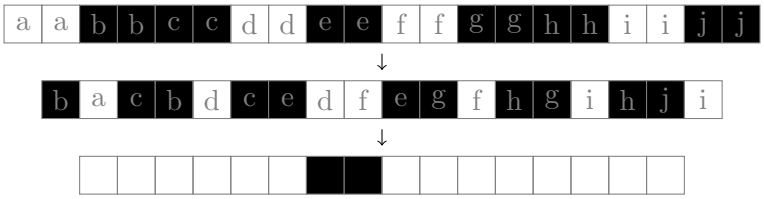
I began thinking about what a scaled up version of a starting state looks like locally. That is to say, what neighborhoods might be found in a scaled up state and what neighborhoods might not be. For the one dimensional case, six of the eight neighborhoods might be found in a scaled up initial state.

Two of the eight neighborhoods are never found in a scaled up initial state.

Then, keeping the above groupings of six and two in mind, I began thinking about some arbitrary scaled up initial configuration and about what intermediate representation was required in order for the correct next scaled up state to be achieved. I started with the assumption that only one intermediate generation was required (this assumption ended up being correct).



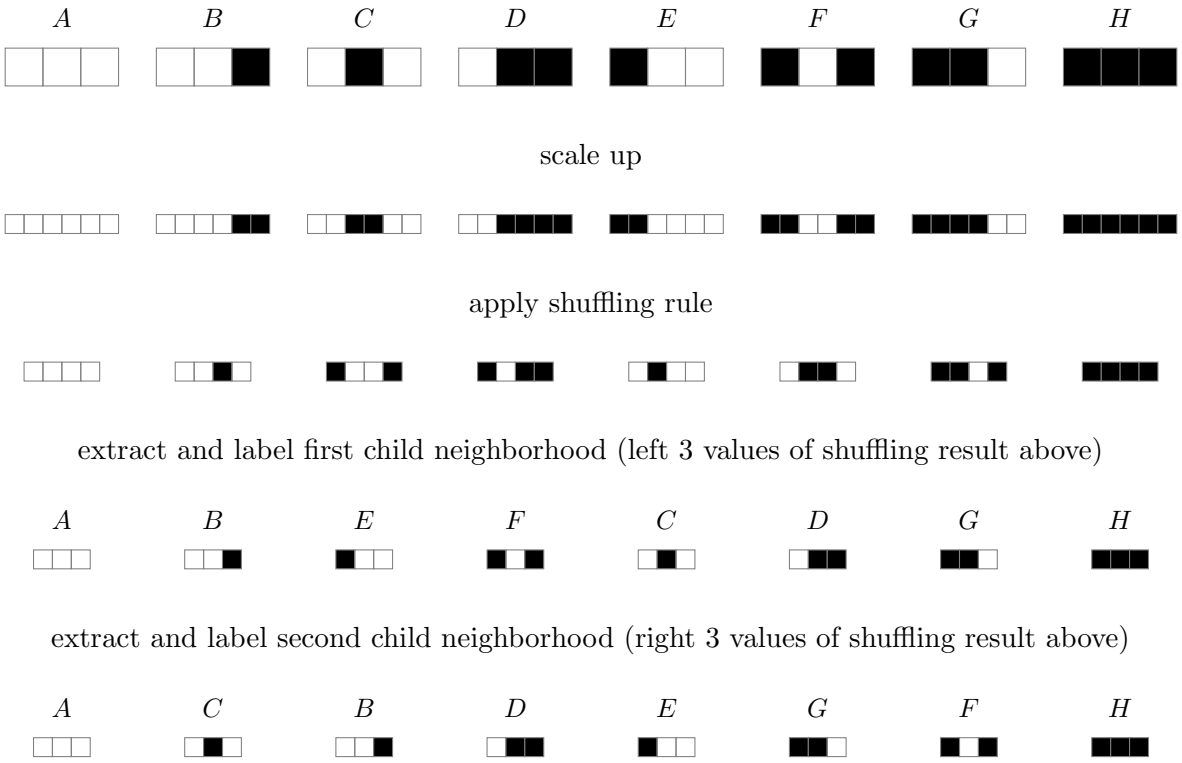
I thought the most useful intermediate representation would be one that spread out the information from the initial state as much as possible without destroying any of it. Given these constraints, there was one possible intermediate representation that jumped out at me. The approach was that if a cell's neighborhood might be part of a scaled up state (if it is part of the group of six states listed above), then it should take the value of the scaled up cell of which it is not a component. Here is a graphic illustrating this concept. I've found that adding labels here makes the nature of the transformation clearer.



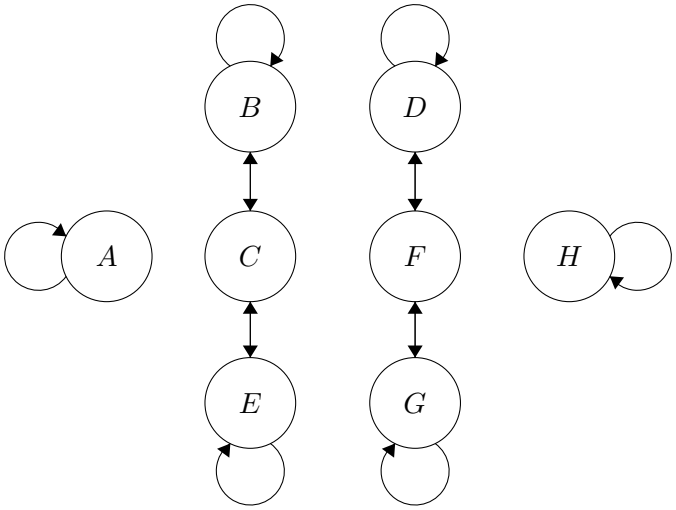
So the intermediate state is just a shuffling of the scaled up state. Explicitly, this shuffling rule for the six potentially-scaled-up neighborhoods listed above is this:

So, what exactly is achieved by this partial, one-dimensional rule given above? Well, as I said, it takes any state that has been scaled by a factor of two and, without loss of information, spreads out its values in the next generation of cells. This has a nice consequence that probably isn't immediately obvious. Namely, the values in the scaled up neighborhood of a scaled up starting state are now in the local neighborhood of both child cells of said scaled up starting state. To give an explicit example, look at the beginning of the diagram with the alphabetically labeled cells. The pattern starts off *aabbcc* and then produces the child pattern of *bacb*. So the cells previously labeled *bb* have in their respective neighborhoods all values originally in the scaled up neighborhood. In other words *aabbcc* gives rise to local neighborhoods *bac* and *acb*.

The key here is now to begin thinking of this as a graph. Neighborhood *abc* can be scaled up to produce *aabbcc* and then, under the intermediate transformation described above, results in two child neighborhoods *bac* and *acb*. These two neighborhoods can then each be scaled up again to give *bbaacc* and *aaccbb* respectively. To these scaled up patterns, the shuffling operation can again be applied to give us *abca* and *cabc* respectively (the four central child values). These can be broken into *abc* and *bca* in the first case and *cab* and *abc* in the second case. This can be represented as a graph which I will derive and then show here for the one-dimensional, two-state example currently under discussion.



From this information, we can create a graph depicting connections between parent and child states.



This graph has four connected components. The key is to realize that if all neighborhoods within a single connected component can be assigned the same output state, the result will be a one-dimensional, self-implementing rule of the type originally desired. Some of the output states (in fact, most of them) are already constrained by the partial shuffling rule that was defined for the six states that look like components of scaled up starting states. The components containing only A and H respectively can obviously be given consistent outputs of white and black respectively. B and E are both constrained to have a black output state. C is not initially constrained, but it is in

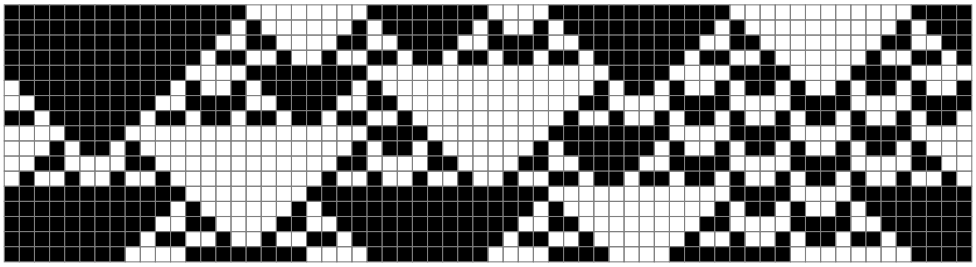
the component with B and E and so must also have a black output state. For isomorphic reasons, D, G, and F must all have a white output state.

It is perhaps good to note that it is not simply luck that it is possible to give the connected components consistent states. The shuffling transformation that lead to the creation of the graph is actually exactly the same as counting the states in the neighborhood and producing as output the state whose count has an odd parity (this is perhaps an overly convoluted way to say "one or three", but it generalizes better later). By design, the shuffling operation produces the same states as were present in the original larger pattern, so parities of the counts are maintained.

For the sake of completeness and for the sake of fun, here is the complete, one-dimensional, self-implementing rule and a random example of its application. It is not a particularly interesting rule, but I think it rises above the level of triviality.

the rule

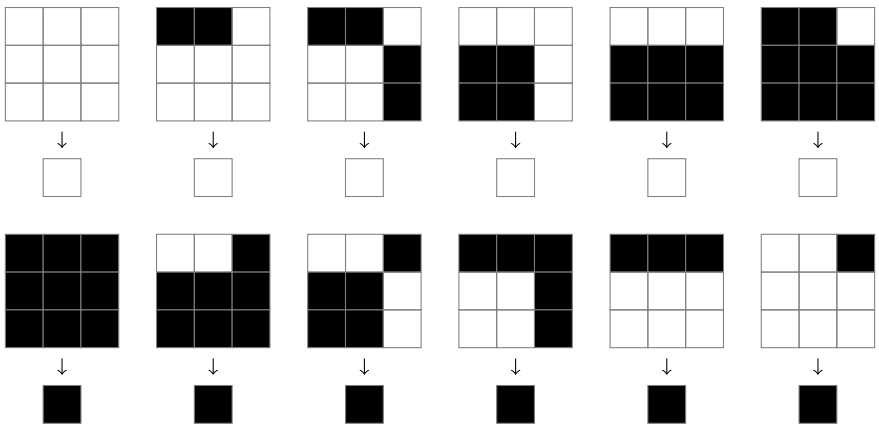
the rule applied to a random, cyclic, initial state that has been scaled up by a factor of four



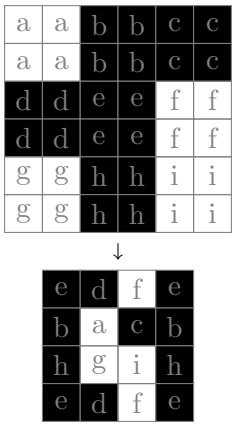
Note how every fourth generation has values in contiguous groups of four and every other generation has values in contiguous groups of two. This is the expected behavior given the self-implementing nature of the rule.

The Two-Dimensional Case

The two-dimensional case is actually not that difficult to handle using the tools and intuitions developed while finding and exploring the one-dimensional case. Again, we are going to focus on a two-state cellular automaton (though the approach will actually generalize easily to higher numbers of states). I covered the one-dimensional case pretty extensively because it's difficult to visually depict similarly exhaustive representations of the two-dimensional case; there are just too many different possible states. However, I will try to at least give a representative sample of the process in the two-dimensional case. The first step is to find the two-dimensional equivalent of the shuffling rule, a rule that will spread out information about the scaled up state. Here is the correct rule:

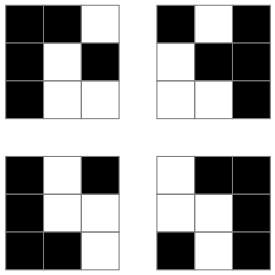


The above shows how the shuffling rule works in two dimensions. Again, it only initially needs to be defined on neighborhoods that look like they might be part of scaled up patterns. Also again, it is equivalent to selecting the state that occurs an odd number of times in the neighborhood. The rationale is that the next state is the value of the scaled up cell that is diagonally across from the corner of the scaled up cell of which the current central state is a member. This strategy should, like with the one-dimensional case, spread out information into the next generation without destroying any of it. Next, I will show a labeled example similar to the one I gave in the one-dimensional case.

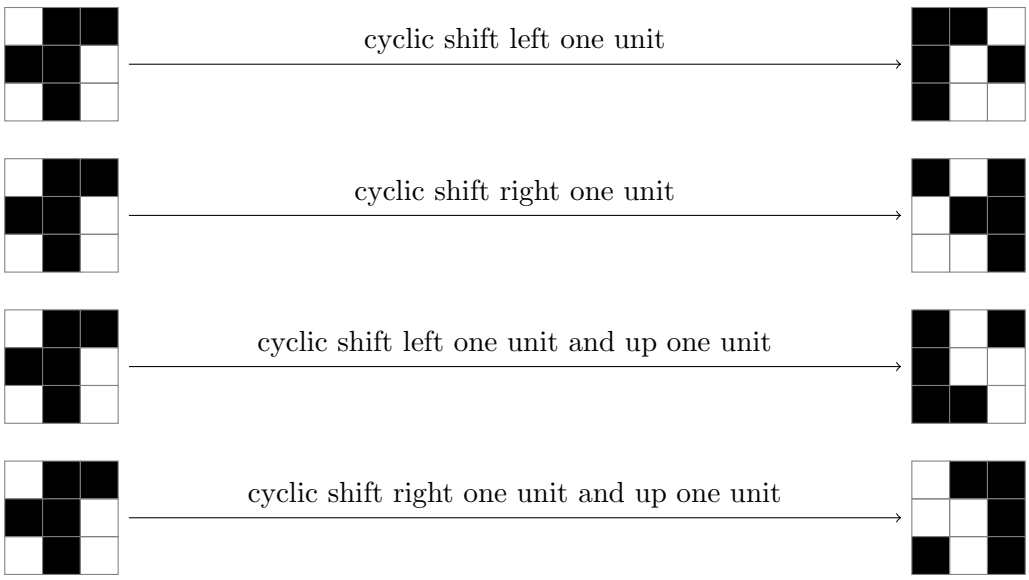


The output state above depicts the next states for the center region of the input state. The labels show where the information from the input state has gone.

Here are the four 3x3 regions from the output state shown above:



These are child states that must have the same output value as the scaled up original state from which they were derived in order for the rule to be self-implementing. One other interesting thing to note about these is that they are all cyclic shifts of the original input state:



This observation about the cyclic shifting can actually allow us to somewhat shortcut the exhaustive parent and child graph creation process that we used for the one-dimensional case. We can just say that all neighborhoods that are equal under cyclic shifting are in the same connected component of the graph and that if a particular connected component has any neighborhood that can be interpreted as a portion of a scaled up pattern, then whatever value the shuffling rule dictates for that neighborhood has to be the output value for every neighborhood in the connected component. One might worry that a single connected components might contain multiple patterns for which the shuffling rule dictates different values, but this is actually not possible. As stated earlier, the shuffling rule is equivalent to taking the value that occurs an odd number of times in the neighborhood. Because child neighborhoods have the same states as their parents (cyclic shifting would obviously maintain this property) there can never be conflicting predetermined values within connected components. Finally, any connected component that does not contain a neighborhood already constrained by the shuffling rule is free to take any value.

Here is some C code that computes the next state of a simple version of this cellular automaton in which all neighborhoods in unconstrained connected components are made to output zero. The *state* input value to this function should be single-dimensional array containing the 9 values of the neighborhood read from left to right and top to bottom.

```
uint8_t find_next(uint8_t *state) {
  int32_t next_x[] = {1, 2, 0, 4, 5, 3, 7, 8, 6};
  int32_t next_y[] = {3, 4, 5, 6, 7, 8, 0, 1, 2};
  for (int32_t i = 0; i < 9; i++) {
    if (state[i] == state[next_x[i]] &&
        state[i] == state[next_y[i]] &&
        state[i] == state[next_x[next_y[i]]]) {
      int32_t corner_index = next_x[next_x[next_y[next_y[i]]]];
      if (state[next_x[corner_index]] == state[next_x[next_x[corner_index]]] &&
          state[next_y[corner_index]] == state[next_y[next_y[corner_index]]]) {
        return state[corner_index];
      }
    }
```

```
  }
  return 0;
}
```

The code works by exhaustively checking if the input state can be interpreted as some cyclically shifted region of a scaled up pattern. If such an interpretation is found, the value dictated by the shuffling rule is returned. If no such interpretation exists, the function returns zero. This function is actually quite general and also implements valid behavior for self-implementing cellular automata with more than two states.

And finally, Here is a short example of the actual behavior of the two-dimensional, self-implementing rule as specified by the C program. I've included 17 generations of the evolution of a toroidal 256 by 256 board with a 16 by 16 scaled starting pattern. Personally, I think it looks quite nice.