**Abstract**

**Introduction**

**The Stranded Cellular Automata Model**

Cellular automata are mathematical models that consist of a grid of cells evolving through discrete steps in time. [1] As the name implies, they consist of cells with states that are “neighbors” to each other and change their states based on the states of their neighbors. The set of all neighbors that influence a cell’s state is defined as the cell’s “landscape”. //define “initial condition/starting generation”, “generation” in the introduction

In the case of the Stranded Cellular Automata (SCA) created by Dr. Holden [2], each cell has 8 possible states and a landscape of 2 neighbor cells that determine its state. Each cell is generated based on a set of rules applied to its landscape. Figure 1 shows the landscape cells in highlighted in red and the resulting new cell in highlighted in blue.

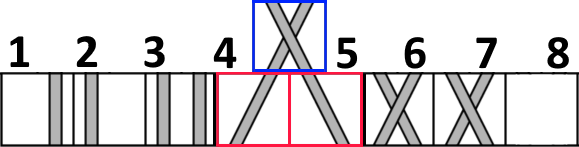


Figure : All 8 cell states, with an example neighbor pair generating a new cell.

In order to distinguish between the two types of crossings, we will refer to the crossing with the strand on top resembling the slant in the letter Z as a “z-cross” and the opposite crossing with the strand on top resembling the slant in the letter S as a “s-cross”.



Figure : The letter S next to a s-cross, and the letter Z next to a z-cross. The relevant sections of each are highlighted.

The calculation of each cell’s state based on its landscape is split into two different rules: the “turning rule”, which dictates whether or not strands will slant, and the “crossing rule”, which dictates which strand goes over the other in the case of a cross. Instead of covering every single case, each rule deals with a more general set of cases, where multiple cell states are equivalent to each other if they exhibit the same features. The turning rule cases include straight, slanted, and absent cells, and the crossing rule cases include z-cross, s-cross, and no cross cells.

Because each landscape consists of two cells, the number of possible landscapes comprised of the 3 different cases would be 3\*3 = 9 different landscapes. Each one of these 9 landscapes controls the status of the new cell. For the turning rule, every landscape determines whether the new cell has straight strands or slanted strands. For the crossing rule, every landscape determines whether the new cell has a s-cross or z-cross. We can label each of these landscapes with a number ranging from 0 to 8, and based on the status it determines we can assign it a 0 or a 1. The number 0 corresponds to straight strands for the turning rule and s-crosses for the crossing rule, while the number 1 corresponds to slanted strands for the turning rule and z-crosses for the crossing rule. A visual example can be seen in Figure 3 for the turning rule and Figure 4 for the crossing rule.

Since each of these bits is labeled 0-8, it is possible to write out each rule in decimal notation. For example, instead of writing turning rule 101000100, it is more concise to write turning rule 324 (the equivalent base 10 number)

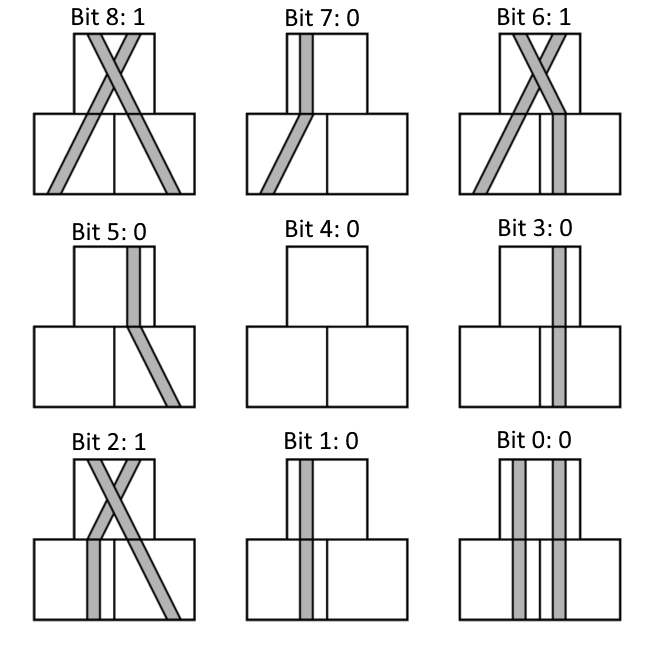


Figure : Turning Rule 324 (Binary 101000100)

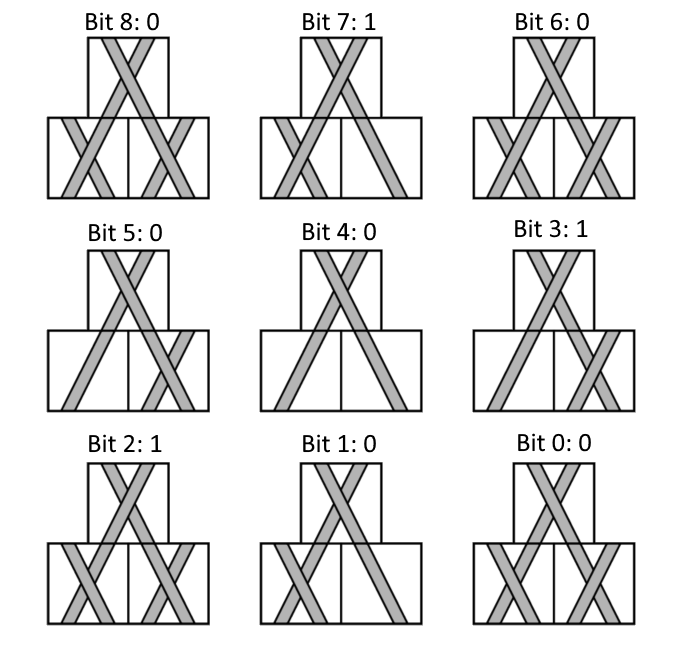


Figure : Crossing Rule 140 (Binary 010001100)

**Representing Braids with Stranded Cellular Automata**

We can use Stranded Cellular Automata to model various types of braids with different numbers of strands. Braids, unlike weaves, have finite width because they reuse the same strands. This means that there is no need to let the border cells “wrap around” as Hao Yang defined the border cells in his work with weaves. Instead, our border cells will act as no strand cells that are not drawn in the figures below.

We started off by constructing physical models of the braids to analyze. We then transcribed the crossings and strands as their corresponding cell states in a Stranded Cellular Automata. Upon checking the output of each neighbor pairing, we were able to derive an initial condition, turning rule, and crossing rule that generated a braid identical to the model.

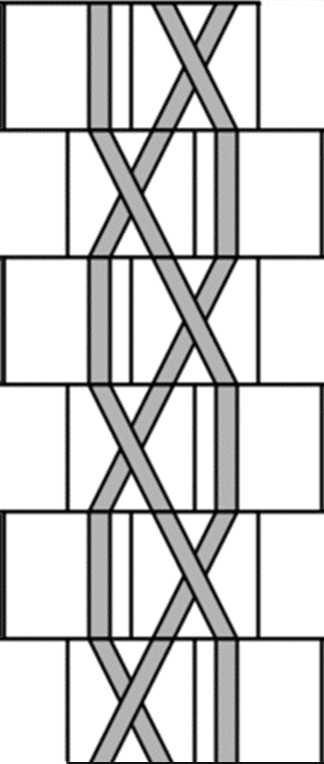
 

Figure 5: 3-Strand Braid and its SCA counterpart, Turning Rule 68, Crossing Rule 32 (68, 32)

After analyzing the simple 3-strand braid and finding no issues with converting it into an SCA with Turning Rule 68 and Crossing Rule 32, we decided to add another strand to add to the complexity. We found two 4-strand braids that were representable by SCA, a “flat” and “square” pair of braids that both used the same turning rule but different crossing rules.

|  |  |
| --- | --- |
| Figure : Flat 4-Strand Braid with SCA counterpart | Figure : Square 4-Strand Braid with SCA counterpart |
| (Turning Rule 324, Crossing Rule 4) | (Turning Rule 324, Crossing Rule 140) |

An interesting observation made when comparing 3-strand braids to 4-strand braids was the “backwards compatibility” of the turning rule shared by the two 4-strand braids we analyzed.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bit Number | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | Decimal |
| 3-Strand Turning Rule | 0 | **0** | **1** | **0** | 0 | 0 | **1** | 0 | 0 | 68 |
| 4-Strand Turning Rule | **1** | **0** | **1** | **0** | 0 | 0 | **1** | 0 | 0 | 324 |

Figure : Turning Rule Comparison, the underlined/bolded bits are the bits relevant to generating the braid's behavior.

Since the case that bit 8 governs in the turning rule does not appear in the 3-strand braid, the value of bit 8 is irrelevant in choosing a turning rule to represent the 3-strand braid. Therefore, it is possible to reuse the turning rule from the 4-strand braids to generate a 3-strand braid identical to the original. See Figure 8 for details.

For the case of braids with 5 strands, there was a lot more room for experimentation as different combinations of cells that previously could not be represented with only 3 or 4 strands emerged. To start, we applied the ruleset of the flat 4-strand braid to 5 strands. The result was that the braid became no longer flat as the number of s-crosses outnumbered the number of z-cross and made the braid start to twist. We observed that each generation of this braid had two crossings, so we altered the crossings of the braid to have equal amounts of s-crosses and z-crosses. We accomplished this in two different ways. First, we had the crossings alternate between 2 z-crosses and 2 s-crosses. Because each generation contained 2 slanting strands that alternated every generation, we referred to it as the “double slant” braid.

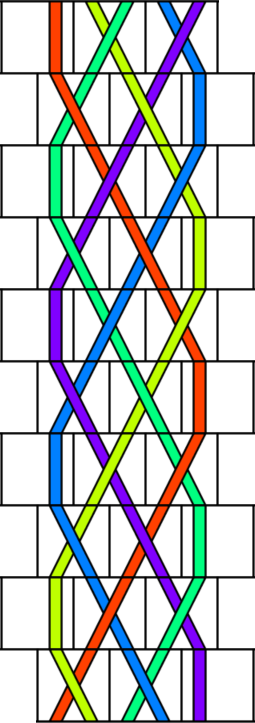


Figure : Double slant 5-strand braid with SCA counterpart (Turning Rule 324, Crossing Rule 6)

Building off the previous braid that had generations that alternated between 2 z-crosses and 2 s-crosses, we attempted to construct a braid that had the same crossings for each generation without twisting. We decided upon having each generation contain a single s-cross adjacent to a single z-cross, resulting in a braid with the top strands exhibiting a “v-shaped” pattern as shown in Figure 10.

|  |  |
| --- | --- |
| Figure : V-shaped 5-strand braid with SCA counterpart | Figure : Zoomed-in view of the first 3 generations. S-crosses are highlighted red and z-crosses are highlighted blue. Note how the red-blue pairs generates different crossing types. |

When analyzing the v-shaped braid, we encountered an issue with finding a crossing rule to represent the crossings. As shown in Figure 11, identical landscapes were generating different output crossings. Distinctly coloring each strand was the first idea we tried but would require adding a lot of complexity to the rulesets that govern them. It is important to note that since there were no conflicts with the turning rule representing the braid, we will only look at solutions that affect the crossing rule only.



Figure 12: Conversion of a colorless no cross cell into 5 color variations

Because the crossing rule is composed of 4 landscapes with 4 strands, 4 landscapes 3 strands, and 1 landscape with 2 strands, the formula for calculating the number of bits needed to represent a crossing rule for a n-strand braid with n colors is:

Plugging in n=5 for our 5 strands gets us 740 as the number of bits needed to represent a distinctly colored turning rule. Since each of the bits can be either on or off, there are possible crossing rules which is several orders of magnitude larger than the original possible crossing rules for the non-color model. This is unreasonably large of a number to deal with when analyzing and deriving rules from. Additionally, the model created by this new ruleset would only work for braids with 5 or fewer strands. To model 6 or more strands a new model would need to be created.

To decrease the number of bits needed to represent the rules and make the model expandable past 5 strands, we decided to try coloring the strands with only two different colors based on whether they were odd or even. Because repeated color strands were now possible, the formula for the number of bits needed to represent a crossing rule with number of colors n, where n is less than the number of strands in the braid it represents is:

Plugging in n=2 for our odd-even coloring scheme gets us 100 as the number of bits needed to represent a odd-even colored turning rule. The number of possible crossing rules for this method, is still not feasible for analysis. The even-odd coloring method also failed to resolve all the landscape conflicts which further invalidates its usefulness.

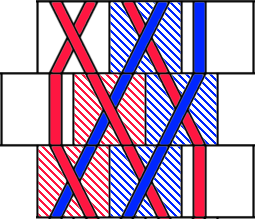


Figure 13: Section of the braid that contains the conflict; the middle generation here is the initial generation, the conflict occurs when the braid repeats

Pinpointing that the crossing rule conflict occurred because generation 1 generated generation 2 and vice-versa, we sought to add a “hold state” generation consisting of straight, non-crossing strands sandwiched between the two generations. Because the strands in the hold state generation do not cross, adding the hold state creates an equivalent braid. This would make generation 1 generate the hold state instead of generation 2, and the hold state generation would generate generation 2.

|  |  |
| --- | --- |
| Generation 1 | Figure : The staggering of the grid causing some strands to disconnect and generations to shift in the opposite direction due to the extra offset hold state generation |
| ^  Generation 2 |
| ^  Hold State |
| ^ Generation 1 |

However, due to the staggering of the grid that the cells are generated in, we could not connect the two braid generations with a single hold state. When we added more hold states we encountered the same issues with crossing rule conflicts since the hold state could not generate both the additional hold state and generation 2.

Taking a step back, we observed that all the s-cross/z-cross neighbor pairs that produced s-crosses were located on the left side of the braid, and the s-cross/z-cross neighbor pairs that produced z-crosses were located on the right side of the braid. If we were to draw a zipper-like line through the middle of the braid, it would be possible to assign a different ruleset to each side of the line.

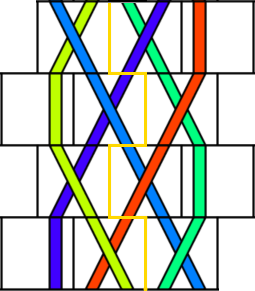
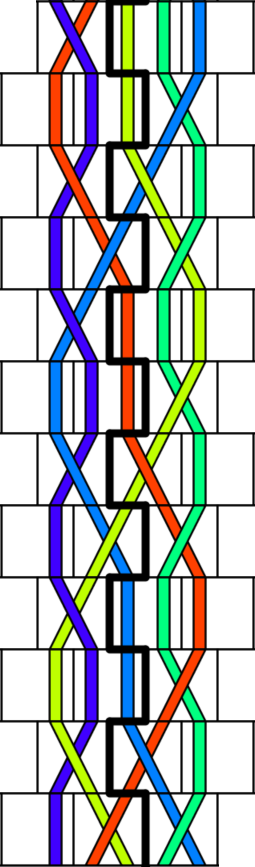


Figure : Zipper-shaped line dividing braid into two parts each with different rulesets

The ruleset used to generate a cell is based on the side of the zipper line that the new cell is on. For example, the bottom generation’s middle and rightmost cell generate a cell that is to the right of the zipper line, so the righthand ruleset is used to calculate the crossing of the new cell. To represent the v-shaped braid, we used

* Over only 3+2 braid, used space-varying rulesets [ (69, 2) , (321, 18) ]



* Over under 3+2 braid – see the ruleset grouping doc for rulesets, time varying only

Start time-varying section here:

**References**

[2] Holden, J. & Holden, L. (2016). “Modeling Braids, Cables, and Weaves with Stranded Cellular Automata.” Proceedings of Bridges 2016: Mathematics, Music, Art, Architecture, Culture, 127-134. Tessellations Publishing.