**Abstract**

**Introduction**

**The Stranded Cellular Automata Model**

Cellular automata are mathematical models that represent an initial condition changing over time. 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. In the case of the Stranded Cellular Automata(SCA) created by Dr. Holden, each cell has 8 possible states and 2 neighbor cells that determine its state.

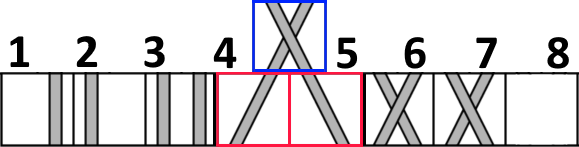


Figure : All 8 cell states, with an example neighbor pair generating a new cell. (Red cells are the neighbors, and the resulting generated cell is blue)

In order to distinguish the two types of crossings, we will refer to the crossing with the strand on top running like the slant in the letter Z as a “z-cross” and the opposite crossing with the strand on top running like 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 neighbor pair is split into two different rules: the “turning rule”, which governs whether or not strands will slant/cross, 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 cells are equivalent to each other given 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.

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)

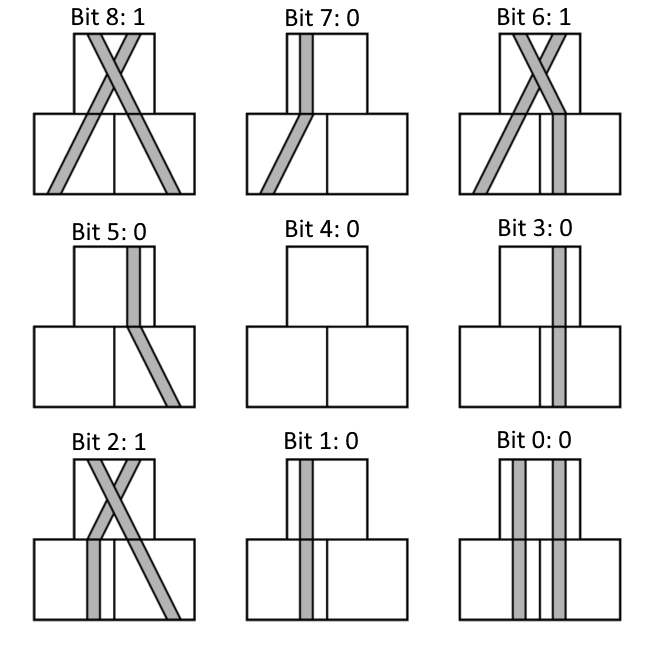


Figure : Turning Rule 324 (Binary 101000100)

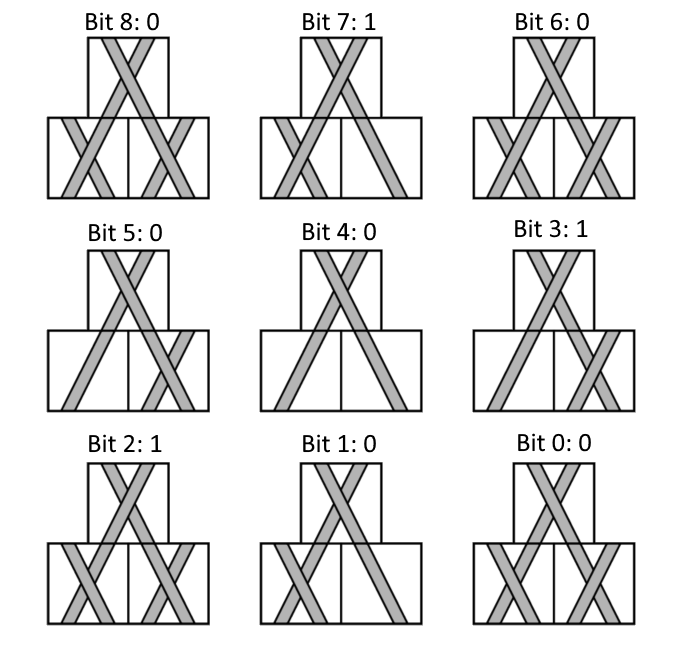


Figure : Crossing Rule 140 (Binary 010001100)

**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 them in his work with weaves. In a similar vein, our turning rule for representing braids will not be fixed due to the nature of braids containing both slanted and upright cells.

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.

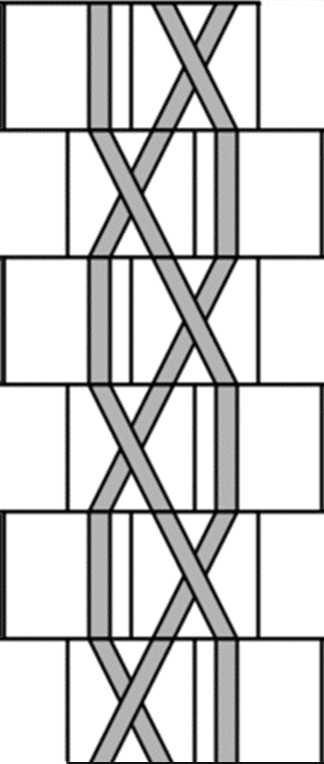
 

Figure 5: 3-Strand Braid and its SCA counterpart (Turning Rule 68, Crossing Rule 32)

After analyzing the simple 3-strand braid and finding no issues with converting it into an SCA, 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. However, the case that bit 8 governs in the turning rule does appear in both 4-strand braids using the turning rule of the 3-strand braid would not generate the same braids.

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. To start, we took the idea of the backwards compatibility of the turning rule 324 and used it to prototype new braids by varying the crossings. The result of this was a braid whose generations alternated between having 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.

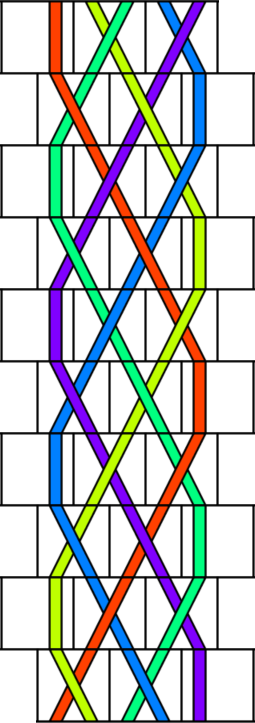


Figure : Double slant 5-strand braid with SCA counterpart (Turning Rule 324, Crossing Rule 6)

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. 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.

|  |  |
| --- | --- |
| Figure 10: 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(10/11, depending on how I choose to display it), identical neighbor pairs were generating different crossings. Initially, we explored several methods of representing the braid. Giving each strand a distinct color would require adding a lot of complexity to the rulesets that govern them. Specifically, for the crossing rules, the “no cross” neighbor cell would need to have 5 variations, and both crossing neighbor cells would need to have 5 \* 4 variations for each pair of colors.



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

To have a rule that works with each strand having a distinct color, it must have more cases for every single combination of colored strands. Because each of these cases needs one bit to represent its output, just adding the cases to handle two no cross cells would require 5\*4 = 20 new cases, more than doubling the number of bits required to represent the rules. Given that the case of two no cross cells is the simplest to update since it only has 2 strands to color, updating the cases with a crossing and a no cross(3 strands to color) or 2 crossings(4 strands to color) would require several more bits than the case with two no cross cells.

We then tried to limit the number of colors by numbering all the strands from 1-5 and coloring the even and odd strands differently. Although doing so did resolve some neighbor pair conflicts, a few remained and continued to make the braid unrepresentable.

\*graphic – make colorless at first and recolor with paint.net, even = blue, red = odd\*

\*do the same as above and highlight where conflicts occur\*

Pinpointing that the crossing rule conflict occurred because generation 1 and generation 2 of the braid kept repeating and generating each other, we sought to add a “hold state” generation consisting of straight, non-crossing strands sandwiched between the two generations.

\*\*The repeat would then be generation 1> hold state > generation 2 > generation 1.\*\*

|  |  |
| --- | --- |
| Generation 1 |  |
| ^  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, and when we added more hold states we encountered the same issues with crossing rule conflicts.

\*graphic – take original v-shaped braid and spread the generations apart to fit the “hold state”\*

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.

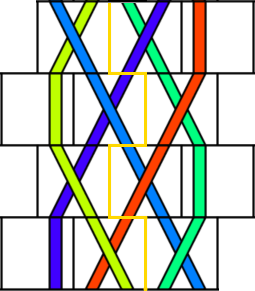
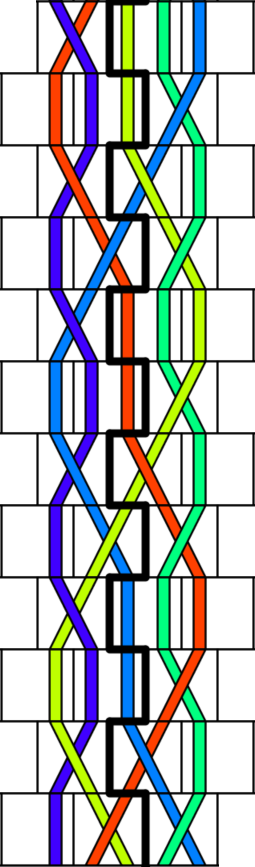


Figure : Zipper-shaped line dividing braid into two parts each with different rulesets

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.

* 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

Programming TO-DO

* Add labels for space-varying rulesets to the sides of the SCA ✔
* Add selector buttons to indicate which side of SCA you are loading rulesets into ✔
* Add indicator for where space-varying rulesets differ ✔
* Show confirmation of loading rulesets ✔