Fall 9-2-2018

Stranded Cellular Automaton and Weaving Products

Hao Yang
Rose-Hulman Institute of Technology, yangh4@rose-hulman.edu

Stranded Cellular Automaton and Weaving Products

Student Investigator: Hao Yang

Faculty Mentor: Joshua Holden

Date: September 2nd, 2018

Abstract

In order to analyze weaving products mathematically and find out valid weaving products, it is natural to relate them to Cellular Automaton. They are both generated based on specific rules and some initial conditions. Holden and Holden have created a Stranded Cellular Automaton that can represent common weaving and braiding products. Based on their previous findings, we were able to construct a Java program and analyze various aspects of the automaton they created. This paper will discuss the complexity of the Stranded Cellular Automaton, how to determine whether a weaving product holds together or not based on the automaton and the Java program we used for investigation.

Contents

1. Introduction

Weaving products, which are made of two distinct sets of threads that interlace each other, are crucial to people's lives nowadays. For example, most of our blankets, clothes, scarfs are made from them. Even though the items are simple, we should not underestimate the amount of information contained in the weaving products. The patterns of the weaving products are not only beautiful but also complex. With the same strands, different weaving rules can result in distinct weaving products. Some products might hold together firmly and have simple patterns, while others can be fancy but cannot hold together at all. In order to study the behavior of weaving products mathematically, we used the Stranded Cellular Automaton (abbreviated as SCA below) [1] to simulate the weaving process.

The Cellular Automaton [2] is a discrete math model, just like the Finite State Machine and the Pushdown Automaton. It consists of a grid of cells, each in one of a finite number of states. Each cell will have several neighbors that decide its state. For different Cellular Automata, the grid can be in different numbers of dimensions and the generating progress will be different. In our model, the grid is in two dimensions and each cell has two neighbors. To simulate the weaving process, we will generate the grid from bottom to the top based on an initial row and a given set of rules. A Java program (available at https://github.com/HaoAndersonYang/StrandedCellularAutomaton) was implemented by us to simulate the SCA and print out grids.

With the SCA, we are able to analyze the behavior and various aspects of some weaving products. In this paper, we will discuss the complexity of them by analyzing the maximum cycle length. In addition, we will talk about the holding-together property of cycles of the weaving products. We will investigate the following questions for the SCAs that can produce weaving products which have holding together cyclic parts and do not have any upright strands under Turning Rule 511 and any crossing rule:

1. For each width, how many distinct cyclic parts can be formed?
2. For each width, what is the maximum cycle length of our SCA?

2. Representing the Weaving and Braiding Products

Unlike an Elementary Cellular Automaton [3], which only has two possible values for each cell (0 or 1); an SCA has eight states for each cell, as shown in Figure 1. When two strands cross, we will identify which strand is on the top and record it. We can represent most of the common weaving and braiding products with the eight states provided since they take care of the direction of strands and the way they cross over each other.

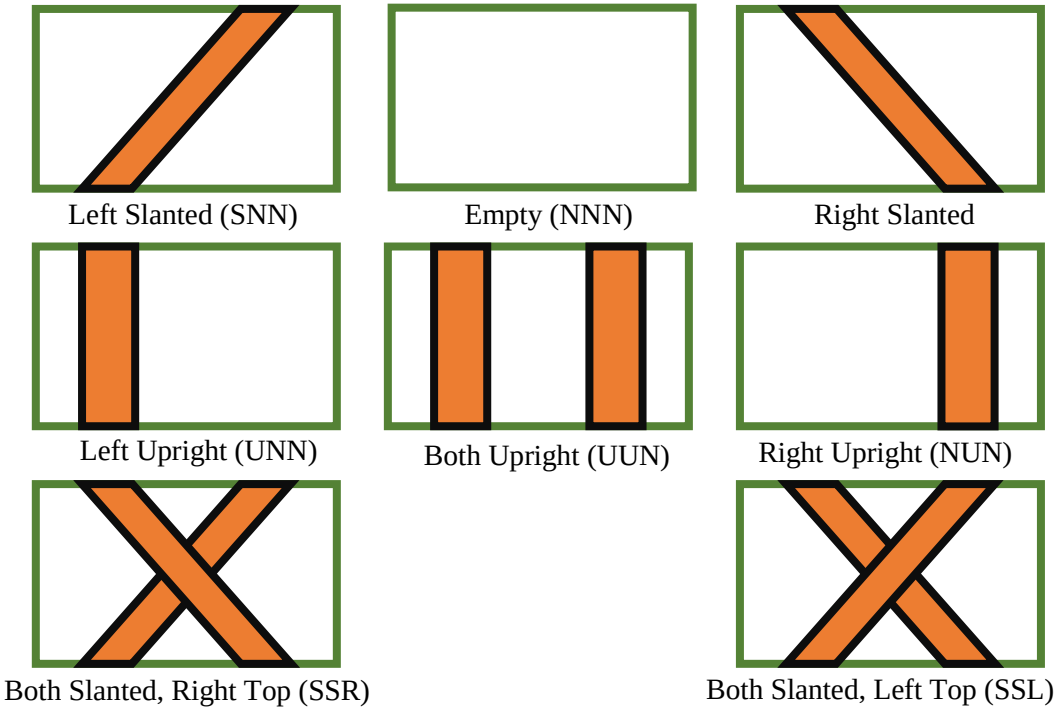

Figure 1: The eight valid states of a Stranded Cellular Automaton.

When we are analyzing a weaving product, we need to rotate it by 45 degrees clockwise, as shown in Figure 2.a. Therefore, the slanted strands will correspond to warps (the vertical threads in original product) and wefts (the horizontal threads in the original product) of common weaving products. To keep coherence with the weaving process, each cell only has two neighbors (as shown in Figure 2.b) in an SCA. Because the weaving process is from the bottom to the top, we represent the time in our model by moving from the bottom of the picture to the top. As demonstrated in the Figure 2.b, the states of two neighbors below each cell contribute to its state.

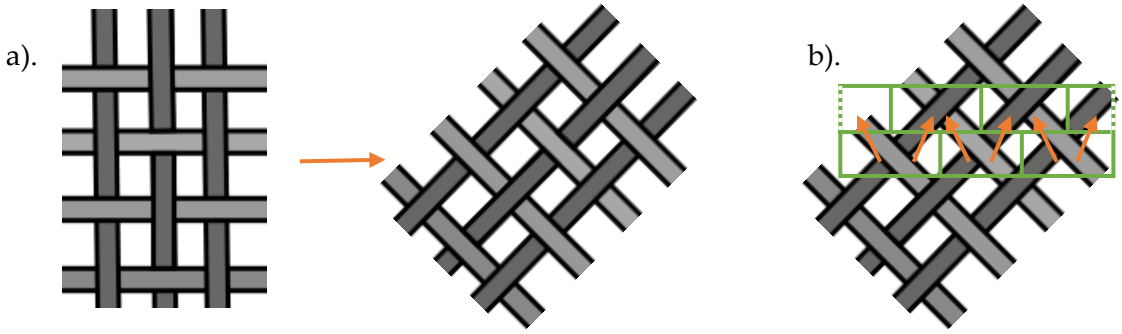

Figure 2: Rotated weaving product (2.a) and the demonstration of neighbors (2.b).

Because we are dealing with the direction of each strand and how they cross over each other, to build a practical model, we will use two separate sets of rules: turning rules and crossing rules. As their names suggest, the turning rule will decide the turning status of strands (upright or slanted) based on the neighbors, while the crossing rule will decide which strand is on the top (left top or right top) when a crossing happens.

Notice that there are nine possible combinations for both turning status and crossing status, we can encode the rules as 9-bit binary numbers. For the turning rules, "1" indicates that the output turning status is slanted while "0" means the output turning status is upright. For crossing rules, "1" means the crossing status is left top and "0" means crossing status is right top. Notice that since there are 9 bits for each rule, the total number of rules is $2^9 = 512$. However, in this paper, we will only discuss weaving products, which means that there will be no upright strands in the cell. Hence, we only need to care about one turning rule, which is turning tule 511.

For reference, Figure 3.a shows Turning Rule 7 (binary 000000111) and Figure 3.b shows Crossing Rule 77 (binary 001001101).

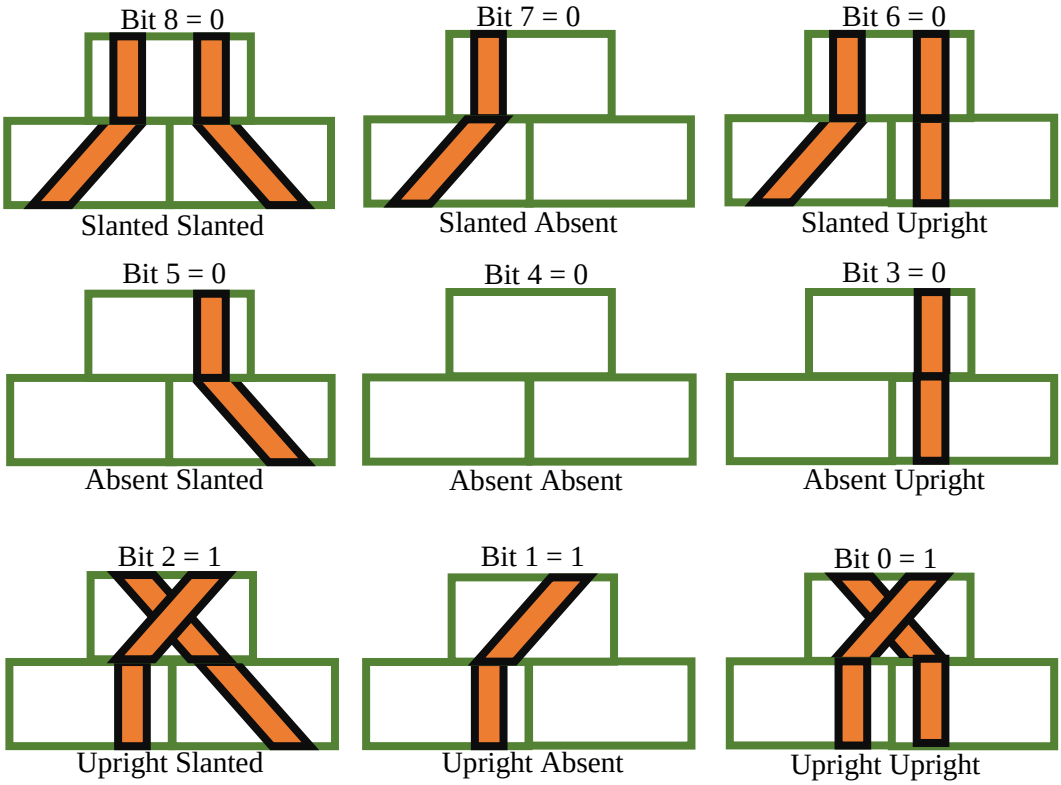

Figure 3.a: Turning Rule 7. The turning status of neighbors is shown below each bit (left first).

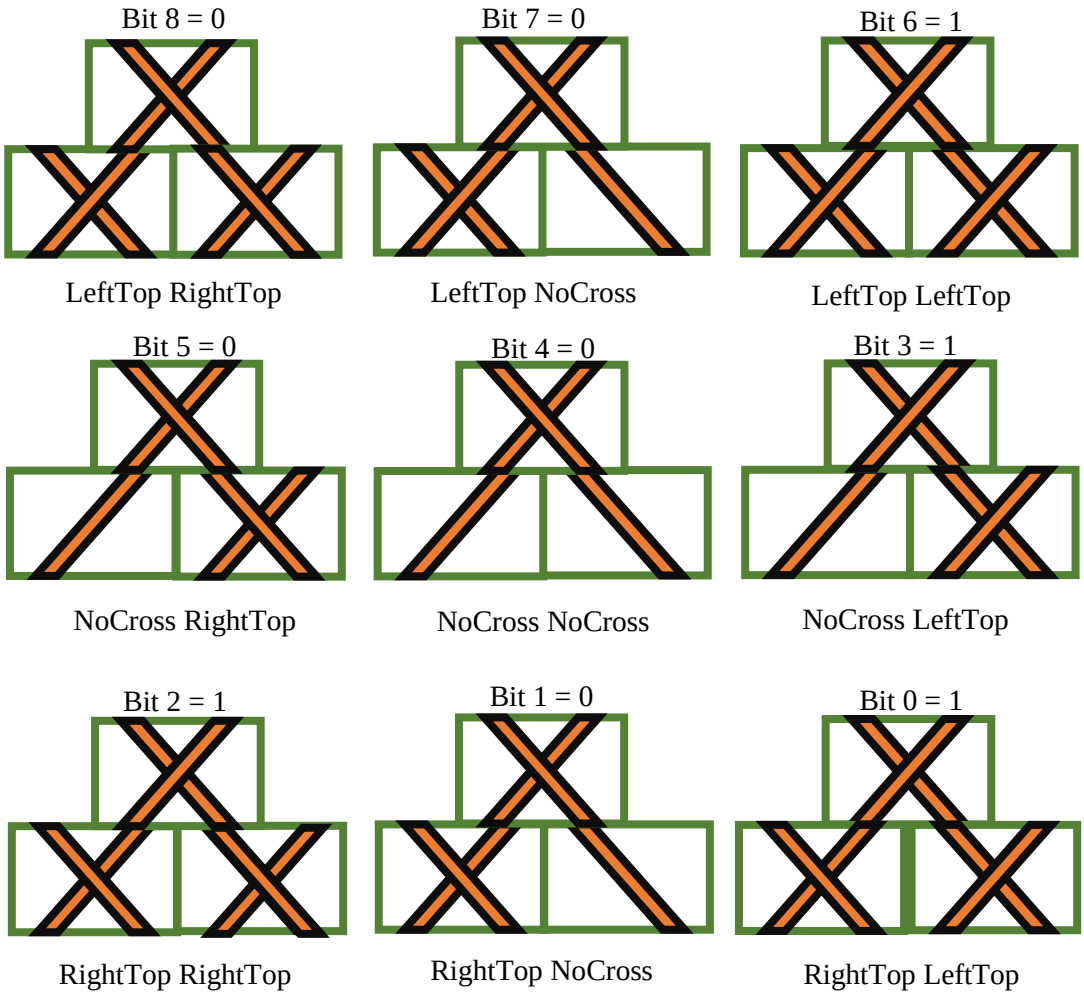

Figure 3.b: Crossing Rule 77. The crossing status of neighbors is shown below each bit (left first).

3. The Complexity of Stranded Cellular Automata

If we consider the weaving products in the real world, many of them are of a cylindrical shape (collar, cuff, clothes and so on). Therefore, we let the cells "wrap around" [1]. That is, on the rows that are odd, the cells are shifted by a half so that the leftmost half cell and rightmost half cell are actually part of the same cell. If we print out the grid with width w, there will be $w - 1$ complete cells and two half cells on the rows that are odd. We can imagine the grid to be cylindrical. This gives an SCA a finite-width grid. With a finite-width grid and a finite number of states, we can easily conclude that the result of an SCA will repeat at some time based on the Pigeonhole Principle. The question is how long can the cycle length be.

For future reference, if the cycle length is k, then when the rows starts to repeat themselves, row i will be exactly the same as row $i + k$ in all aspects, including the thread each cell contains, the state each cell is in, and whether the row is wrapping around or not. In the following paper, we will refer to the k repeated rows as "cyclic part".

Since we are only considering weaving products, we will fix our turning rule to be 511 (that is, no matter what states neighbors are in, the new cell will be in the slanted state) and do not include any upright strands. By modifying the crossing rule and initial conditions, we will be able to find the maximum cycle length for each width.

Due to the numerous amount of possible combinations of initial conditions and crossing rules, this paper will only discuss the maximum cycle length of one of the special cases. In that case, all cells in the initial row will have two strands, which means that there will only be cells in states "SSR" or "SSL". Since we fixed our turning rule to be 511, the only possible states in the grid will be those two.

In order to represent the grid in a more algebraic way, we assign binary value 1 to crossing state right top (R) and binary value 0 to crossing state left top (L). Then, we define a crossing rule to be "additive" if the output state will always possess the value that is the sum modulo 2 of the value of two input states [4]. For example, when the input crossing states are R and L, the output crossing state will be R because R + L = 1+0=1. If at least one of the input crossing states is no cross (N), we do not care about its output since the case we are analyzing will not have any cell with crossing status N.

Crossing Rule 291, for example, is one of the additive crossing rules. The binary encoding of Crossing Rule 291 is 100100011, and the output of it is demonstrated in chart 1 on the next page. In the chart, input states are in white cells and output states are in grey cells. The number beside the states indicates their binary encoding (for crossing state N, we do not care so its value is X). For those outputs that decide whether the crossing rule is an additive or not, there are expressions besides them to demonstrate that Crossing Rule 291 is a valid additive crossing rule.

We now defined additive crossing rule and showed that Crossing Rule 291 is one of them. With the definition, we can now analyze our special case.

R=1=0+1		L=0		L=0=0+0	
L=0	R=1	L=0	N=X	L=0	L=0
R=1		L=0		L=0	
N=X	R=1	N=X	N=X	N=X	L=0
L=0=1+1		R=1		R=1=1+0	
R=1	R=1	R=1	N=X	R=1	L=0

Chart 1: *Outputs of Crossing Rule 291. Notice that it is an additive crossing rule.*

Proposition 1. *Under Turning Rule 511 and any additive crossing rule, an SCA that has a power of two grid width (*w*) and an initial condition with all states that can be represented with 0's and 1's will have its rows filled up with all 0's after row* w.

Proof. For the following proof, we set the first row of the grid and Pascal Triangle to be row 0 and we will assume that the width is a power of two.

Lemma 1 [4]. *In this special case, we are able to "add up" initial conditions. If initial condition C can be represented by the bitwise modulo 2 sum of initial conditions A and B, every cell in the grid produced by initial condition C will have the value which is the sum of corresponding cells in grids produced by initial condition A and B.*

Proof of Lemma 1: Because our crossing rule is additive, the state of new cell will be based on the sum of its two neighbors. Assume that we are adding two grids (Grid I and Grid II) together to form a new grid (Grid III). For any certain cell position that is not in the first row, we can have the following:

In Grid I, the value for the cell is a and its neighbors have values b and c and $a = (b + c) \bmod 2$.

In Grid II, the value for the cell is d and its neighbors have values e and f and $d = (e + f) \bmod 2$.

Now, for the cell in the corresponding position in the Grid III, the value of it will be $(a + d) \bmod 2$. Since $a = b + c$ and $d = e + f$ and $\big((b + e) \bmod 2\big) + \big((c + f) \bmod 2\big) = (b + c + e + f) \bmod 2$ [5], we have $(a + d) \bmod 2 = \big((b + e) \bmod 2\big) + \big((c + f) \bmod 2\big)$. Thus, the additive rule is correctly followed in the new grid. Therefore, we can "add up" two grids and obtain a valid new grid.

After being able to add up two grids, we can narrow down the initial conditions we need to prove.

Lemma 2. *We only need to prove our proposition is true when the initial conditions consist of all 0's but a 1 at any position (that is, all SSL but one SSR).*

Proof of Lemma 2: Based on Lemma 1, by adding up the rows bitwise, we can form any initial condition based on a combination of rows with all 0's but a 1. The readers should notice that our proposition states the grid will end up with all 0's when rows begin to repeat themselves. In addition, adding up rows with all 0's will result in a row with all 0's. Therefore, if we can show the proposition is true under the initial conditions with all 0's but a 1, we know that the proposition is true in all cases.

Because the grid is wrapping around, we can always shift our initial row around. Hence, we only need to show the proposition is true for one case. Therefore, we can set the position of the cell with crossing state SSR to be at position $w/2$. This makes the cell to be at the center of the grid.

Lemma 3. *In this special case, the grid is equivalent to an upside-down modulo* 2 *Pascal Triangle until row* $w - 1$. *The row* i *in our grid will be the same as row* i *in the transformed Pascal Triangle if we fill all empty spots with* 0*'s.*

Proof of Lemma 3: Because we are producing the grid based on an additive crossing rule, each cell in the new row will be the sum of its two neighbors. If we fill positions that are empty in Pascal Triangle (with row number less than $w - 1$) to have value 0, we know that the Pascal Triangle under row $w - 1$ is also generating its position in the new row based on the sum of the position's two neighbors [6]. Therefore, we can conclude that the Pascal Triangle has exactly the same generation rule as the SCA with additive crossing rule.

Since $(a + b) \bmod 2 = (a \bmod 2 + b \bmod 2) \bmod 2$, we can conclude that our grid is indeed an upside-down modulo 2 Pascal Triangle. Since row i of Pascal Triangle will have $i + 1$ items, at row $w - 1$ the grid will be filled with row $w - 1$ of Pascal Triangle. However, for rows above it, the result will not prevail since the grid's width is smaller than the width of Pascal Triangle.

Because the width is a power of two, we know that there exists an integer n such that $2^n = w$. Notice that for $w - 1 = 2^n - 1$, its binary representation consists of all 1's. Since the binary form of $2^n - 1$ has n bits, it means that there will be n 1's in the binary representation of it. It has been proved that in Pascal Triangle, the number of odd numbers in row $w - 1$ is equal to 2^k, where k is the number of 1's in w's binary representation [7]. Hence, the row $2^n - 1$ in Pascal Triangle will consist of 2^n odd numbers, which will fill the row. Therefore, based on Lemma 3, in our SCA grid, row $2^n - 1$ is going to be filled by 1's (because all odd numbers mod 2 will result in 1). Consequently, row 2^n will be filled by 0's since all input states are 1 and adding up 2 1's will result in a 0 for the output state. For any row that is beyond row 2^n, it is going to consist of all 0's since adding two 0's together will also provide you with a 0.

Now, based on Lemma 2, since we proved the proposition is true with the initial conditions consists of all 0's but a 1 at any position, we are able to state that our proposition is true for all initial conditions.

Remark 1. Even though the case we are analyzing in this paper is special, it should be able to inspire readers to think about the problem in a new aspect. Utilizing the idea of converting the SCA grid into the modified Pascal Triangle can potentially be helpful for analyzing other aspects of the SCA we created. Unfortunately, due to the limit of time, we were not able to develop more details of the complexity.

4. Whether the Product Holds Together

Another important aspect of a valid weaving product is that the strands must hold together. When we try to remove a thread, we will have to remove all other threads that crossed over it. If the product holds together, one cannot lift up several strands of the weaving product while leaving the rest untouched. Therefore, for a valid weaving product, any attempts to remove a single thread will result in lifting up the whole product.

Different valid weaving products will result in products with various behavior. The main idea is that the strands in the product will stick together. Readers should notice that the weaving product is valid if and only if its cyclic part (introduced in Section 3) holds together. Otherwise, when the weaving proceeds, most of the product will fall apart. Hence, we only care about the holding-together property within the cycle and ignore the rows before the cycle when analyzing this property. In order to find the possible weaving products generated by an SCA, we need to generate a graph and check it to make sure that the strands held together.

It is hard to tell whether the strands hold together or not simply by looking at the weaving product. Therefore, we need to convert it to a directed graph and take advantage of graph theory. To convert the weaving (or braiding) product to a directed graph, let each strand be a node. Then, we say there is a directed edge from Node B to Node A if strand A is on top of strand B at some cell. Notice that due to the property of the weaving products, it is possible to have a directed edge from Node A to Node B and from Node B to Node A. Figure 4 shows a sample conversion from the result generated by Turning Rule 333 and Crossing Rule 39.

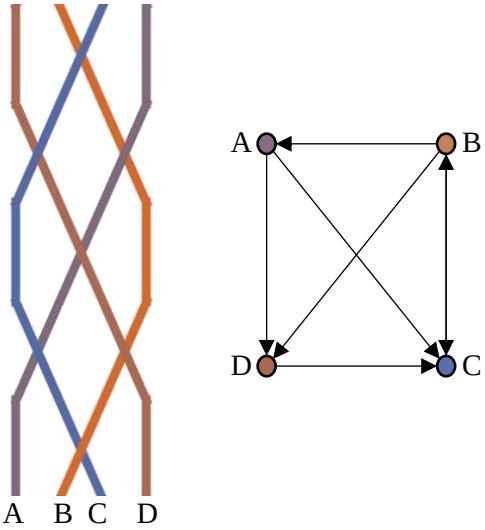

Figure 4: Sample conversion from an SCA result to a directed graph.

Proposition 2 [8]. *The converted graph is strongly connected if and only if the weaving (or braiding) product holds together.*

Proof. By the construction of the graph, being able to reach from Node A to Node B means if we want to remove thread A, we will eventually need to remove thread B. The converse is also true.

If a weaving product "holds together", removing any single thread (call it strand A), will result in removing the threads that crossed over it. This means that from Node A, we can reach all other nodes in

the graph. Since the statement is true for any thread, we know that we can reach every node in the graph from any starting point. Therefore, the graph is strongly connected.

On the other hand, if the graph is strongly connected, we can reach every node in the map from any starting point. That is, removing any thread will eventually remove all other threads. This means that the weaving product holds together.

Remark 2. With the proposition, we can convert the weaving product to a directed graph and determine its properties. Applying the Kosaraju–Sharir Algorithm, which will be discussed in the latter part of this paper, can check whether a graph is strongly connected or not easily.

The conversion makes it easier for us to check the validity of weaving products. However, it is tedious to search through all initial conditions. Even though we ignore the upright strands, there are still five available states for each cell. When the width grows bigger, the difficulty of using a brute force search increases. Fortunately, we can narrow down the initial conditions using symmetry.

Proposition 3. *For any combination of initial condition and crossing rule, we can construct a mirrored combination of initial condition and crossing rule so that the result of them will behave the same in terms of holding together*.

Proof. We can obtain the mirror of a certain row by flipping the original row 180° around a vertical axis. A sample is shown in Figure 5.a. Similarly, the mirror of a set of crossing rule can be obtained by flipping the crossing rule 180° vertically. The reader should notice that because we flip not only the output cell but also two input cells, we could not simply interchange 0's to 1's in the output. For example, flipping input (NoCross, RightTop) will give us (LeftTop, NoCross). Sample conversion is shown in Figure 5.b.

Hence mirroring both the initial condition and the crossing rule will result in a weaving product that is flipped 180° vertically. If strand A was on top of strand B in the original weaving product, strand B will be above strand A in the mirrored product. The mirrored product is basically the same product as before, and hence its property of holding together isn't changed.

Remark 3. The proposition can also be proved using graph theory. Converting both the products to directed graphs will result in two graphs with a reversed path direction. Since reversing all paths will not change the strongly connected property of a directed graph, the mirrored products behave the same in terms of holding together.

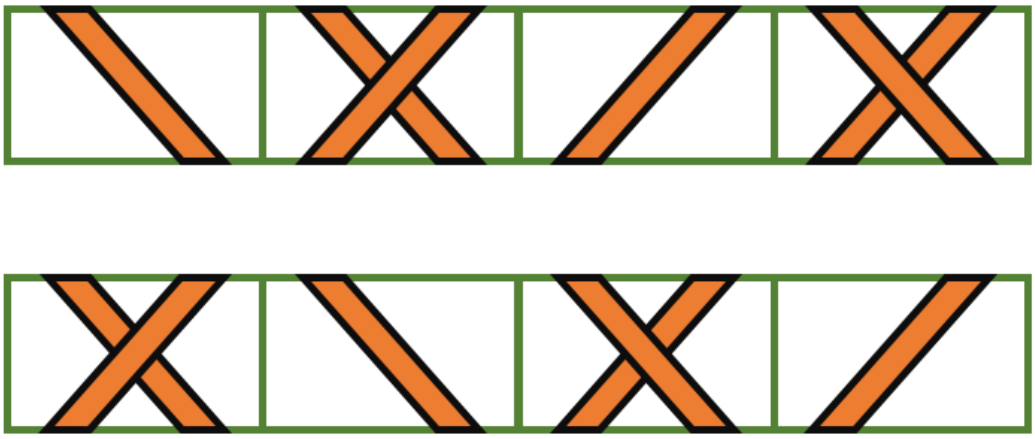

Figure 5.a: Mirrored row (the top row is the original row and the bottom is the mirrored one)

Figure 5.b: Sample conversion of crossing rule (the left set is the original crossing rule and the right set is the mirrored crossing rule).

Now, with the proposition, we can decrease the number of initial conditions that we need to check. Since the property of holding together of initial condition A with crossing rule B will be analyzed with mirrored initial condition A' and crossing rule B', we can search through all crossing rules while only checking half of the initial conditions. However, since there must exist at least two crossings in a product that holds together, the number of initial conditions to check can be further decreased.

Proposition 4. *Assume all strands are slanted. One only needs to check all initial conditions that start with a cell in the SSR state (both slanted and right top) to find all cyclic parts that hold together.*

Proof. Because our SCA has a cylindrical, wrapping-around grid, shifting every row left or right will not interfere with the final result. Hence, as long as there is a cell in the SSR state in the initial row, we can shift the initial row around and make it a row that begins with a cell in the SSR state.

If there is not a cell in the SSR state in the initial row, there are three cases:

i. There is at least one cell in the SSL state in an initial row.
For this case, we can create a mirror of the row (then the cell in the SSL state will become a cell in the SSR state) and shift it around. Eventually, we can build a row starting with a cell in the SSR state.

ii. All cells in the initial row are single-sided or empty (that is, SNN, NSN or NNN). However, when the automaton proceeds, there will exist a row (call it Row X) that has at least one cell whose crossing status is not NoCross.
In this case, we know that Row X must have at least one cell in state SSL or SSR. We can shift (and mirror if necessary) the row and make it a row beginning with a cell in the SSR state. Since the result of an SCA will repeat eventually, we can take this row as the initial row and construct a product where the cyclic part has the same holds-together property.

iii. All cells in the initial row are single-sided or empty (that is, SNN, NSN or NNN). In addition, when the automaton proceeds, there will **not** exist a row that has at least one cell whose crossing status is not NoCross.
In the last case, since there will not be a row where some strands will cross, the weaving product is not going to hold together at all. Thus, we can skip these cases safely.

Remark 4. With the proposition, we actually decrease the number of initial conditions to one-fifth of the original number. The reason is that for the first cell in the row, we now only need to check one possibility instead of five.

5. Java Program Implementation

In this section, we will discuss the implementation of the computer program we created to help with the study. We wrote the program with Java SE 8. The program has two main features:

1. Generate the weaving product:
   Given a valid turning rule, a valid crossing rule, the width and height of the grid, whether to detect the cycle or not and the initial condition, the program generates the grid and outputs the picture of the weaving product, the maximum cycle length and the holding-together property of the cyclic part of the product based on the input.
2. Brute-force testing:
   Given a width, the program performs a brute-force search through all crossing rules and the initial conditions that we need to check for the holding-together property (proved in Section 4).

The following will demonstrate how the key features are implemented.

5.1 Cycle Detection

In order to find out the maximum cycle length of the weaving product, a cycle detection technique is required. Because the program has to generate the grid from the bottom to the top based on the initial row, we can integrate the cycle detection algorithm into the grid generation routine. Therefore, we will be able to find out the cycle length when filling out the grid. Notice that for all given inputs, the generated weaving product must have a cycle because of the Pigeonhole Principle. Hence, we do not have to consider the case that our program runs into an infinite loop.

After considering various choices, we chose to implement Brent's Algorithm [9]. There are two main reasons for our decision. The first one is that this algorithm will output the cycle length directly and the program does not need to go into any subsequent stage for it. The algorithm is time efficient when compared with its fellows [9]. According to its creator, the algorithm performs 36 percent faster than Floyd's algorithm on average.

The Java implementation of the algorithm is shown in Appendix-I. The reader should notice that only the first stage of Brent's Algorithm is implemented and the displayed code is slightly different from the actual code in the program. The actual method used in the program considers conditions when the user does not want to detect the cycle and when the program finds the cycle before it fills out the grid.

5.2 Generating Initial Conditions

To perform brute force testing on different combinations of crossing rules and initial conditions, we have to find a way to generate all such combinations without duplication. Otherwise, the program can waste a lot of time on generating repeated results. Because we always have to check through all possible crossing rule, we can simply create a loop for it (start with Crossing Rule 0 and end with Crossing Rule 511). However, for the initial condition, we need to adopt a different method.

After an investigation, we decided that a Generalized Gray Code (described in detail in [10]) could be helpful to the program. The reason is that by enumerating all n-ary Gray Codes of length k, the program is able to generate k^n distinct arrays of length k with minimal change, each of them corresponding to one of the k^n possible initial conditions. Since the bit in each entry of the array

represents a state, the array can be easily converted into initial conditions that can be used in latter part of the program. Because we are only analyzing weaving products, our program does not have to consider upright strands. This reduces the number of possible states to five (SSR, SSL, SNN, NSN, NNN). In addition, since we can fix the first cell to be state "SSR" (as proved in Section 4), we only need to enumerate an n-ary Gray Code of length $(w - 1)$ for an SCA that has a grid with width w.

The Java implementation of Generalized Gray Code enumeration is shown in Appendix-II. Guan developed and analyzed the enumeration algorithm (as shown in [10]) used in the code.

5.3 Strongly Connected Check

Eventually, to figure out whether the cyclic parts holds together or not, the program has to be able to generate a directed graph and check its connectivity. However, because the SCA is able to generate rows infinitely, we need to determine when our program should stop generating cells and start analyzing the product. Fortunately, the answer is straightforward.

Proposition 5. *The program should stop generating cells and start analyzing the directed graph when it finds out the cycle length.*

Proof. Because the program used Brent's Algorithm for cycle length detection when generating the cells, the SCA must have already generated some repeated rows. Then, generating any more rows will simply repeat the behavior within the cycle. This means that we will not be able to create any more directed edges based on the new rows. Hence, the holding-together property will not change after the first repeat occurs. The program can safely stop at the place where it finishes computing the cycle length and it can start to analyze the weaving product.

Remark 5. Since the program is generating the rows from bottom to top, we will need to keep track of the threads and update the directed graph whenever we generate a new row. In our code, each cell stores the threads that are in it. We assign numbers to the threads when the grid is initiated. Since the grid width is w and each cell can contain up to two threads, we number the threads from 0 to $2w$ in order of left to right. If the thread is in cell number i, it will either be numbered as $2i$ or $2i + 1$ based on its position in the cell (left or right). When a new cell is generated, the program will assign up to two threads to it based on its neighbors and the directed graph (stored as an adjacency matrix in our program) will be updated. Hence, the directed graph will always be up-to-date. In addition, by integrating Brent's Algorithm with the cell generation routine, we make sure that the directed graph will be ready for the strongly connected check whenever the cycle is found.

Eventually, in order to check whether the directed graph is strongly connected or not, we implemented the Kosaraju–Sharir Algorithm (described in [11]) in our code. It will take the directed adjacency matrix generated by the program as an input and perform two deep-first searches to check the graph's connectivity. In the end, it will output the number of connected components of the graph. If the number is one, then we know that the graph is strongly connected and hence the weaving product holds together.

In Appendix-III, the Java implementation of the Kosaraju–Sharir Algorithm is shown. The code displayed in the paper does not output any information about the connected components of the graph. In the actual program, that information will be printed out before the method "stronglyconnectedCheck()" returns the result.

6. Usage of Program and Examples

In this section, we will demonstrate some sample usage of the program and the output of it. There will be an example of each feature described in Section 5.

6.1 Generate weaving product:

Text-based interface and inputs are shown in Figure 6 below. The input texts are shown in green italic.

```
Input the Turning Rule (0~511)
68
Input the Crossing Rule (0~511)
0
Input the height of the grid
9
Input the width of the grid (not less than 2)
9
Input the initial configuration in the following form:
<Turning Left><Turning Right><Crossing> <Turning Left><Turning Right><Crossing>...
For the turning rules, please use U for Upright. S for Slanted. N for No Strand
For the crossing rules, please use N for NoCross. L for LeftTop. R for RightTop
UUN UUN UUN UUN SSR UUN UUN UUN UUN
If you want to decide whether the pattern holds together based on a full cycle enter
1. Otherwise enter 0.
1
```

Figure 6: The sample input for generating weaving product

The text output is shown in Figure 7 below. The picture output is shown in Figure 8 on the next page.

```
Length of the cycle is 98
The pattern does not hold together and it has 2 separated parts.
The components are:
Thread#0 Thread#1 Thread#2 Thread#3 Thread#4 Thread#5 Thread#6 Thread#7 Thread#9
Thread#8 Thread#10 Thread#11 Thread#12 Thread#13 Thread#14 Thread#15 Thread#16
Thread#17
Displaying the graph in the pop up window. Closing the window will exit the program
```

Figure 7: The sample output for generating weaving product (text output)

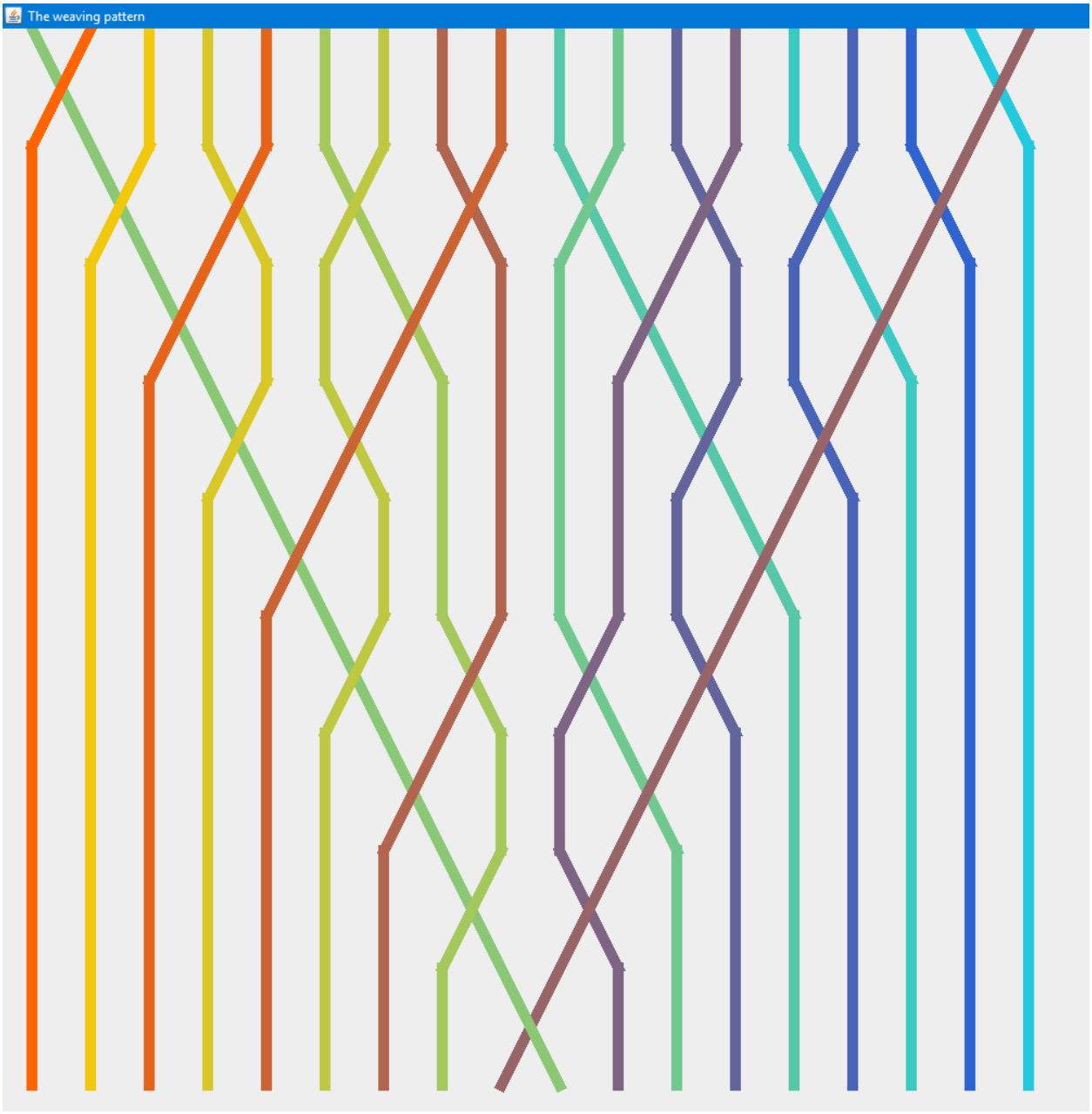

Figure 8: The sample output for generating weaving product (picture output)

6.2 Brute-force testing:

The sample input and output are displayed in Figure 9 and 10. Notice that readers can choose whether they want every cells have two threads or not. The input texts are shown in green italic.

Please enter the width:

2

If you want to only check the case where all cells will have two strands, enter 1. Otherwise enter 0

1

Progress: ||>>>>>>>>>>>>>>>>|| 100%

For width 2 when each cell always has two strands we have:

The maximum cycle is 4.

It is reached with initial condition SSR SSL and crossing rule #0.

The average number of distinct pattern for each crossing rule is 0.0078125.

Figure 9: The sample input and output for brute force testing (each cell has two threads)

Please enter the width:

2

If you want to only check the case where all cells will have two strands, enter 1. Otherwise enter 0

0

Progress: ||>>>>>>>>>>>>>>>>|| 100%

For width 2 with all possibilities we have:

The maximum cycle is 4.

It is reached with initial condition SSR SSL and crossing rule #0.

The average number of distinct pattern for each crossing rule is 0.25.

Figure 10: The sample input and output for brute force testing (all possibilities)

The combinations of crossing rules and initial conditions that will produce a valid weaving product are stored in a text file. Due to the size of the file, it is not displayed here. Readers can download the program from GitHub and run the program. The text file for width = 2 to 10 will be provided alongside with code.

7. Conclusion

After investigation, we figured out the maximum cycle length for width from 2 to 10. The results are shown in Chart 2 below.

Width (even)	Length	Width (odd)	Length
2	4	3	12
4	8	5	40
6	24	7	56
8	16	9	144
10	80		

Chart 2: *Maximum cycle length for width from 2 to 10.*

For the number of distinct cyclic parts, we were not able to come up with a conclusion based on our research. Instead, we found the average number of distinct cyclic parts for each crossing rules under a given width from 2 to 10 (shown in Chart 3). The numbers are rounded to nearest hundredth. Notice that different crossing rules can generate same cyclic parts so the exact number of distinct cyclic parts for each width cannot be obtained by multiplying the average number by the number of crossing rules.

Width (even)	Average number of distinct cyclic parts	Width (odd)	Average number of distinct cyclic parts
2	0.25	3	1.63
4	2.32	5	7.82
6	20.97	7	72.23
8	237.42	9	862.91
10	3116.84		

Chart 3: *Average number of distinct cyclic parts for each crossing rules for width from 2 to 10.*

Clearly, we are still at the beginning of everything. In this paper, we only analyzed the complexity of the SCA when the crossing rule is additive and the initial condition consists of only cells in states SSR and SSL. For the aspect of whether a weaving product holds together or not, we also did not go further than width equals 10. There is a lot of work to be done before we can fully understand how weaving products work mathematically.

However, our findings should be helpful for people who are interested in this area. Even though the complexity of the SCA we investigated is simply one of the many cases, the result and the way of approaching it should be helpful for readers who want to move on and analyze other possible combinations of rules and initial conditions. In addition, since the Java program we wrote is on GitHub, people who are interested in it can download it and utilize it for their research in related areas.

In the future, people can work on multiple aspects, including but not limited to the aspects we discussed in this paper. We so far only discussed the complexity in regards of how long will the cycle length be. It is also interesting to find out the maximum time it took for the cycle to appear (that is, the number of rows between the initial row and the row where the cycle starts). Another interesting question is how many unique weaving products the SCA can generate. By defining "unique" differently, the researchers should be able to obtain various results. Furthermore, the researchers can try determining the combinations that can produce products that hold together firmly or hold together loosely. Researchers should notice that they will need to define "firmly" and "loosely" themselves before the study begins.

Appendix.

I. Implementation of Brent's Algorithm

```
public int generateCell() throws Exception {
    // Power records the current power of two, lam is cycle length, row and checking
is row number for current row (Hare) and checking row (Tortoise)
    int power = 1, lam = 1, row = 1, checking = 0;
    // Row 0 is the bottom of the grid, we already generated row 0 and row 1 in
previous part of code.
    // Array currentRow contains information for row 1 and Array checkingRow contains
information of row 0.
    while (!(checking % 2 == row % 2 && compareRow(currentRow, checkingRow))) {
    // Compare currentRow and the checkingRow to see if they are equal.
    // If so, we break.
    // If only one of the two rows are wrapping around, we consider them to be
different rows.
        if (power == lam) { // If we need to start a new power of two
            checkingRow = currentRow; // Set checking row to current row
            checking = row; // Update the checking position for future use
            power *= 2; // Update the power of two
            lam = 0; // Reset lam
        }
        currentRow = nextRow(currentRow, row + 1);
        row++; // If we don't need to start a new power of two, we move on
        lam++;
    }
    return lam; // Return cycle length
}
```

II. Implementation of Generalized Gray Code

```
int N = 5; // Number of possible states in each cell
int[] n = new int[width]; // Stores the maximum value of each array entry
int[] g = new int[width]; // Stores N-ary Gray code of width width-1
int[] u = new int[width]; // Stores whether we decrement or increment current bit
int i, k; // i is the bit of the Gray code we are enumerating, k is the value of the
code
for (i = 0; i < width; i++) {
    g[i] = 0;
    u[i] = 1;
    n[i] = N;
} // Initialize each array
while (true) {
    String[] input = new String[width];
    input[0] = "SSR"; // Fix the leftmost cell to be in state “SSR”
    for (int j = 1; j < width; j++) {
        input[j] = cellStatusHasshMap.get(g[j - 1]);
    } // Convert the current Gray code to initial condition
    initialConditionMap.put(input, false);
    i = 0; // Enumerate next Gray code
    k = g[0] + u[0]; // Reset k
    while (k >= n[i] || k < 0) { //Do we need to switch to next bit
        u[i] = -u[i]; // Mark the bit to perform opposite operation next routine
        i++; // Move on to next bit
        k = g[i] + u[i]; // Update k
    }
    g[i] = k; //Set new Gray code
    if (g[width - 1] != 0) { // Check if we generated every Gray code
        break;
    }
}
```

Detailed analysis and explanation of the code are presented in [10].

III. Implementation of the Kosaraju–Sharir Algorithm:

```
public boolean stronglyConnectedCheck() {
    int size = availableThreads.length; //AvailableThreads contains all threads
    connectivityChecklist = new ArrayList<>(); //Initialize checklist
    componentsList = new ArrayList[size]; //Stores component (array of lists)
    visited = new int[size]; //Stores if the node is visited
    assigned = new int[size]; //Stores if the node is assigned to a component
    for (int i = 0; i < size; i++) {
        if (availableThreads[i] != 0) {
            visit(i); //Perform Depth First Search on the node
        }
    }
    for (int node : connectivityChecklist) {
        assign(node, node); //Assign the node to a new component
    }
    int componentsCount = 0;
    for (int i = 0; i < size; i++) { //Loop and count components
        if (componentsList[i] != null && availableThreads[i] != 0) {
            componentsCount++;
        }
    }
    return componentsCount == 1; //If there's only one component, the graph is
}                                //Strongly connected

public void visit(int node) {
    if (visited[node] == 0) {
        //Mark the node as visited if it hasn't been visited
        visited[node] = 1;
        //Depth First Search on the nodes connected by current node
        for (int i = 0; i < availableThreads.length; i++) {
            //Is there a path from current node to Node i?
            if (threadsMap[node][i] == 1) {
                visit(i);
            }
        }
        //Prepend the node to checklist
        connectivityChecklist.add(0, node);
    }
}

public void assign(int node, int root) {
    if (assigned[node] == 0) { //Is node assigned to a component?
        assigned[node] = 1; //Mark the node as assigned
        if (componentsList[root] == null) { //Initiate a new component if needed
            componentsList[root] = new ArrayList<Integer>();
        }
        componentsList[root].add(node); //Add node to component
        for (int i = 0; i < availableThreads.length; i++) {
            //Perform the reverse Graph Depth First Search.
            //Is there a path from node i to current node?
            if (threadsMap[i][node] == 1) {
                assign(i, root);
            }
        }
    }
}
```

References

[1] 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.

[2] Weisstein, Eric W. "Cellular Automaton." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/CellularAutomaton.html

[3] Weisstein, Eric W. “Elementary Cellular Automaton.” From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/ElementaryCellularAutomaton.html

[4] Rowland, Todd and Weisstein, Eric W. "Additive Cellular Automaton." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/AdditiveCellularAutomaton.html

[5] Wikipedia contributors. (2018, August 12). “Modular arithmetic.” In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Modular_arithmetic&oldid=854652063

[6] Wikipedia contributors. (2018, July 29). “Pascal's triangle.” In Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Pascal%27s_triangle&oldid=852439588

[7] Su, Francis E., et al. “Odd Numbers in Pascal's Triangle.” Math Fun Facts. https://www.math.hmc.edu/funfacts/ffiles/30001.4-5.shtml

[8] Enns, T. C. (1984). “An efficient algorithm determining when a fabric hangs together.” Geometriae Dedicata, 15(3), 259-260. doi:10.1007/BF00147648

[9] Brent, R. P. (1980), “An improved Monte Carlo factorization algorithm.” BIT Numerical Mathematics, 20(2), 176–184. doi:10.1007/BF01933190

[10] Guan, D. (1998). “Generalized Gray Codes with Applications.” Proceedings of the National Science Council, Republic of China, 22(6), 841-848.

[11] Sharir, M. (1981). “A strong-connectivity algorithm and its applications in data flow analysis.” Computers & Mathematics with Applications, 7(1), 67-72. doi:10.1016/0898-1221(81)90008-0