Cellular Automata and the Separation Problem

Matthew Timm
Department of Computer Science
University of New Mexico
Albuquerque, United States
mtimm1984@unm.edu

Sean Timm
Department of Computer Science
University of New Mexico
Albuquerque, United States
stimm48@unm.edu

***Abstract*—In a cellular automata, each cell determines its next state by looking at the state of its neighbors within a certain distance. Cellular automata are commonly used as an idea model to understand certain complex systems. Here, we attempt to solve the separation problem, in which the automata attempt to slip an array into two areas, one of entirely living cells, and one of entirely dead cells. We implement a genetic algorithm to attempt to solve this problem in 1 and 2 dimensions. We found that the genetic algorithm was able to solve the 1 dimensional case easily, but was unable to solve the 2 dimensional case. Instead, we solved this problem by hand. Furthermore, we discuss what this means for higher dimensional spaces and more states.**

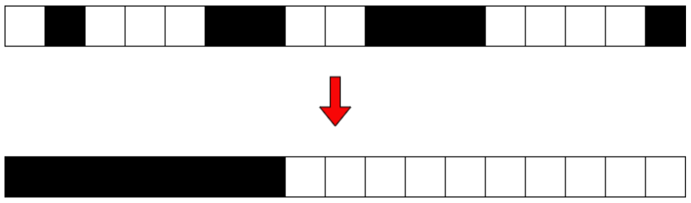

Fig. 1: An example of a solution to the separation problem. the top case is an example of an input, and below we have an example of an output, where all of the living cells (black) are clustered on one side of the array

I. Introduction

The world of cellular automata (CA) is a rather interesting one. In a CA, each cell determines its next state only by looking at the state of its neighborhood, usually between two states. A neighborhood in this case is a number of cells $k/2$ away from the starting cell. For example, Wolfram looked at cellular automata with a neighborhood size of 3 and Mitchell looked at automata with a neighborhood size of 7, both in one dimension [1]. Famously, Conway's Game of life is a Cellular automata that exists in two dimensions and follows a basic rule set.

Cellular automata are a common area of study, as they can model the spreading of a virus through a population or the population dynamics of simple organisms. However, we are instead looking at a particular problem and trying to determine if there exists a CA which can solve such a problem. A famous problem in the field is the majority classification problem, which looks at if cells in the population can all go to either white or black based on what the majority of all of the cells are. We look at a problem adjacent to this. Can cells in a Cellular automata divide themselves into two regions, one of cells that are entirely alive (black) and one which is entirely dead (white), without changing the number of dead and alive cells. Unlike the majority classification problem, we assume the population has borders, instead of being on a torus. To solve this problem, we use a genetic algorithm to evolve the rule sets for our CA's. However, as you'll see in the rest of the paper, this generally went poorly for us. To achieve better results, we ended up creating the correct rules by hand.

A. Discussion of the Cellular automata

Throughout the paper below, we focus on the movement of the 1s between cells as a good way to discuss the problem, rather than discussing the state of a particular cell. We find it's far easier for the reader to understand the movement of one of the colors, rather than discussing population dynamics. For example, it's simpler to say a 1 was passed from one side of the array to the other, rather than discussing how each cell knows what color to be by the rules. With that being said, discussing the state of cells is still presented.

B. Genetic Algorithms

A genetic algorithm is an iterative approach to solving a problem. Using cellular automata as a lens, we start by creating a population of rules for the automata at random. These rules likely won't be close to solving the problem, but are a starting point. Each iteration, the rules are compared against the test cases and scored. Those with the best score are replicated more in the next population. As well, the rules are mutated randomly and are often crossed with other rules to produce better results.

II. Methods

We used the python library DEAP [1] to create our genetic algorithm. We feed in rules as binary arrays, where each cell represents the rule for cells that add up to its index, plus some offsets. An example of this is as follows. Let's say that we have the neighborhood 101. This can be converted to a number viewing it as binary, $4 * 1 + 2 * 0 + 1 = 5$. Thus, the fifth spot in the array would dictate the transition for the middle of the array. The offsets are caused by the borders, which follow slightly different rules as they have less cells to check.

At every step, the population of new rules would run on our test arrays. For the one dimensional case, tests contained an input array, the desired output array, and the absolute limit on the number of steps it may take. The limit on the number of steps is to avoid cells running forever without making progress. For the two dimensional case, the tests contained an input matrix, the allowable number of steps, and the dimensions. We created a program which could solve if the matrix was partitioned correctly for the two dimensional case, as there are many more cases and matrices to test. In both cases, if the population was ever the same after a step, we ended the test and checked if the output was correct.

To choose the next population, we used a tournament selection algorithm, which takes sets of the rules and runs a tournament with the best being moved on to the next generation based on there fitness. The fitness was defined as follows: for the 1 dimensional case, the fitness was the inverse of the Hamming distance [3] from each of the desired outputs provided. For the two dimensional case, the inverse of the number of incorrect cases was the fitness. While the one dimensional rule leads to faster progress and is generally a better metric than simply looking at the correct cases, for the two dimension problem there are many solutions to the problem, as opposed to the two solutions for the one dimensional case (all of the black cells on the left or all on the right). Afterwords the population would be mutated through random bit flips and rules would be crossed for better results.

III. 1 Dimensional Results.

An adept reader will probably be able to quickly tell that a possible rule to solve this problem exists. In fact, the solution only requires a neighborhood size of 3 and the rule itself is only 16 bits long. The rule which solves the problem is 0111010001110001. The first four bits represent the rules for the left edge of the array and the last four bits represent the right edge of the array. The rest dictate the middle of the array. Below is a helpful aid for the reader to understand the rule:

```
LeftEdgeCase:
0   1   1   1
00  01  10  11

MiddleCase:
0    1    0    0    0    1    1    1
000  001  010  011  100  101  110  111

RightEdgeCase:
0   0   0   1
00  01  10  11
```

What we can see is that this is a conservative rule when it comes to switching colors. A cell only switches colors if it knows that one of its neighbors will switch to its color. For example, we see that 110 leads to a 1 being in our cell, whereas 010 leads to a zero. This is because the first rule can't know if the bit next to it will become zero and thus must wait until this occurs to pass its value, whereas the value already being zero means the second rule can pass its value. This makes sense. To preserve the number of both white and black cells, the rule needs to make the automata careful as to not lose a 1 or 0. With a r sized neighborhood, it's likely cells could move bits faster across the array, however, the same caution would still apply. Initially, due to the small size of this rule, we first solved this by hand. Then we wrote the genetic algorithm to see how long it would take to solve the problem. The algorithm found the solution after only 6 iterations. With that being said, we consider this to be a proof of concept. Below we detail our work on the more challenging 2D rule.

IV. The 2D rule

Let us first consider the size of a basic 2 dimensional rule. Counting itself, without edge cases the rule would be 512 bits long, as each cell has 9 neighbors and two possibilities. That is already quite a long rule, and unfortunately we have far more edge cases than before. There are four corners, each with 4 bits and thus 16 possibilities per corner. We also have four edges each one requiring 6 bits totaling with 64 possible rules per edge. In total, this means that each rule is 832 bits long. This leads to an very large search space. Our first attempt was to simply allow the genetic algorithm to work with rules of this length to see if it would make progress. The algorithm would find rules to solve our simple cases quickly, but was unable to generate anything to solve even slightly complicated matrices. This is unsurprising; the computer we used was not a super computer and using python works against us in terms of efficient computation. It was at this point that we shifted our approach.

A. *Using the 1D rule in two dimensions.*

To simplify the problem and give the genetic algorithm a chance, we decided to use the 1D rule in two dimensions. What we mean by this was of any of the cells not on the left edge of the matrix, we simply use the 1D rule with its neighbors on the left and right. The solver would instead try to find a rule that would solve the left edge and would move cells up to connect the regions of black cells. The solver was unable to produce this solution as well, which surprised us. The rule was only 56 bits long, which is far smaller than the previous rule.However, after writing our best interpretation of the rule, we realized we had given the genetic algorithm an impossible task. The issue can be seen in the image below.

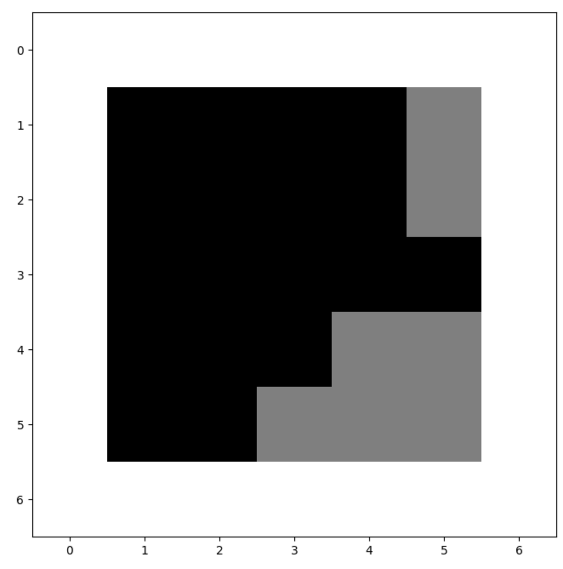

Fig. 2: The case that provides the issue for the cellular if we use the 1D rules for the 2D case. Even if we allow cells to move up on the left edge, the algorithm gets stuck here.

We can clearly see that in this case the algorithm is stuck. The rules on the left edge can no longer move the bits around, the rules for anywhere but the left edge only move values left and right. The edge is complete, and thus the rules must maintain that. Without the ability to move values up and down, the rule can only guarantee that one of the regions will be separated.

In order to find a solution, it became clear that we had to find a way to reduce the problem size. To accomplish this, the first thing to notice is that not all of the bits in the two dimensional case are required to solve the problem.

$$\begin{Bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{Bmatrix}$$

Looking at the matrix of values above and only considering moving the living values to the left or up (not diagonally), it becomes clear that the cell 0 and 8 provide almost nothing. We can't move our value into cell 0, as we have banned diagonal movement, and because the rules are conservative, we'll never move our value from cell 4 into cell 1 unless it already contains a 0. Thus we can ignore cell 0. Cell 8 is similar, as there is no diagonal movement, whatever value it values only affects our neighbors 7 and 5, but to the cell it can safely be ignored. If we use this reduction to all of the rules, we are left with a rule size of 304, however, we had already solved for the left edges rules. Thus, we gave the computer the problem of solving a 248 bit rule.

It was unsuccessful. To give it a better chance, we once again reduced the problem, solving all of the edge cases by hand and leaving it with a 128 bit rule to solve, the exact same size as the rules in the majority classification problem as done by Melanie Mitchell. This time, the genetic algorithm stopped after a little over 150 iterations, claiming to have found a rule to solve this case. However, as is often the problem with genetic algorithms, this was only due to it getting closer to solving the test cases. 3 of our test cases were random, and it's likely they were fairly easy to solve, as when that rule was tested against a few other 5 by 5 matrices, it failed spectacularly. The rule not only didn't solve them, it didn't even maintain the number of alive and dead cells.

B. A solution to the 2D problem

We could have created more test cases in an attempt to solve. However, now that we had reduced the rule to something that a human could understand, we decided to simply hand craft the rule. This was also made easier by that fact that we had a way to check the correctness not only of the ending solution, but also of each step, at least in terms of the number of alive and dead cells. We augmented it with a stopping feature to return which matrix failed and why (whether it failed due to an incorrect number of alive and dead cells or where the cells weren't clustered. The former being the more common issue). This rule and the edge cases are provided below. Please note the numbers which follow each rule, which dictate the cells used in that rule as in the matrix above.

```
TopRightRule (3 4 6 7)
0    0    0    1    0    0    0    0    0    0    0    1    1    1    1    1
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

BottomRightRule(1 3 4)
0   0   0   0   0   0   0   1
000 001 010 011 100 101 110 111

RightEdgeRule(1 3 4 6 7)
0     0     0     1     0     0     0     0     0     0     0     1
00000 00001 00010 00011 00100 00101 00110 00111 01000 01001 01010 01011

0     0     0     0     0     0     0     1     0     0     0     0
01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111

0     0     0     1     1     1     1     1
11000 11001 11010 11011 11100 11101 11110 11111

TopEdgeRule(3 4 5 6 7)
0     0     0     1     1     1     1     1     0     0     0     0
00000 00001 00010 00011 00100 00101 00110 00111 01000 01001 01010 01011

0     0     0     0     0     0     0     1     1     1     1     1
01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111

1     1     1     1     1     1     1     1
11000 11001 11010 11011 11100 11101 11110 11111


BottomEdgeRule(1 2 3 4 5)
0     1     0     0     0     1     0     0     0     1     0     0
00000 00001 00010 00011 00100 00101 00110 00111 01000 01001 01010 01011

0     1     1     1     0     1     0     0     0     1     1     1
01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111

0     1     0     0     0     1     1     1
11000 11001 11010 11011 11100 11101 11110 11111
```

```
LeftEdgeRule(1 2 4 5 7)
0     1     1     1     0     0     0     0     0     1     1     1
00000 00001 00010 00011 00100 00101 00110 00111 01000 01001 01010 01011

1     1     1     1     0     1     1     1     1     1     1     1
01100 01101 01110 01111 10000 10001 10010 10011 10100 10101 10110 10111

0     1     1     1     1     1     1     1
11000 11001 11010 11011 11100 11101 11110 11111

TopLeftRule(4 5 7)
0   1   1   1   1   1   1   1
000 001 010 011 100 101 110 111

BottomLeftRule(1 2 4 5)
0    1    0    0    0    1    1    1    0    1    1    1    0    1    1    1
0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

MiddleRule(1 2 3 4 5 6 7)
0       0       0       1       1       1       1       1       0       0       0
0000000 0000001 0000010 0000011 0000100 0000101 0000110 0000111 0001000 0001001 0001010
0       0       0       0       0       0       0       0       1       1       1
0001011 0001100 0001101 0001110 0001111 0010000 0010001 0010010 0010011 0010100 0010101
1       1       0       0       0       0       0       0       0       0       0
0010110 0010111 0011000 0011001 0011010 0011011 0011100 0011101 0011110 0011111 0100000
0       0       1       1       1       1       1       0       0       0       0
0100001 0100010 0100011 0100100 0100101 0100110 0100111 0101000 0101001 0101010 0101011
0       0       0       0       0       0       0       1       1       1       1
0101100 0101101 0101110 0101111 0110000 0110001 0110010 0110011 0110100 0110101 0110110
1       1       1       1       1       1       1       1       1       0       0
0110111 0111000 0111001 0111010 0111011 0111100 0111101 0111110 0111111 1000000 1000001
0       1       1       1       1       1       0       0       0       0       0
1000010 1000011 1000100 1000101 1000110 1000111 1001000 1001001 1001010 1001011 1001100
0       0       0       0       0       0       1       1       1       1       1
1001101 1001110 1001111 1010000 1010001 1010010 1010011 1010100 1010101 1010110 1010111
1       1       1       1       1       1       1       1       0       0       0
1011000 1011001 1011010 1011011 1011100 1011101 1011110 1011111 1100000 1100001 1100010
1       1       1       1       1       0       0       0       0       0       0
1100011 1100100 1100101 1100110 1100111 1101000 1101001 1101010 1101011 1101100 1101101
0       0       0       0       0       1       1       1       1       1       1
1101110 1101111 1110000 1110001 1110010 1110011 1110100 1110101 1110110 1110111 1111000
1       1       1       1       1       1       1
1111001 1111010 1111011 1111100 1111101 1111110 1111111
```

Similar to the 1D rule, this rule works conservatively on it's neighbors, being careful to avoid losing bits and only moving when it's clear that a value won't be lost. We gave preference to 1s moving from right to left. A 1 moves up only if and only if it's the case that it cannot pass the 1 forward and the cell to the right of the cell above us doesn't have a 1 to pass. This preserves the number of 1s and 0s, but allows the 1s to flow to the other side of the array.

C. More detail on the 2 Dimensional Rules

This section is intended to provide more detail on each case. The average reader can skip this. each rule is broken up into four seconds: When it will become a 1 (go from 0 to 1), when it will become a zero (go from 1 to 0), when it will stay zero and when it will stay 1.

1) Rules for the Left Edge: Rules on the left edge tend to become ones.

The top left rule only uses 3 bits. The only information needed is whether the top right has a 1 , at which case it will always remain 1, or if the bit below of right is a 1, in which case it becomes a 1. It will only stay a zero if all of the bits are zero and it will never become a zero if it is a 1.

The bottom left rule's cell will become a 1 if it's the case that it is currently zero and the bit on the right is a 1. It will switches from a 1 to a zero if neither of the bits above it have a one. It will stay a zero if the bit on the right of it is a zero, regardless of the bits above. It will stay a 1 if any of the bits above of it are a 1.

For the cells on the left edge, they will become a 1 if it's the case that either the bit on the right is a 1 or the bit below is a 1. They will become a zero if it's the case that they have a one and both of the bits above have a zero. They will stay a 1 if either of the bits above is a 1. They will stay a zero if the cells below and on the left are zeroes. Finally, they will stay a 1 if either of the bits above is a 1.

2) Rule for the Right Edge: Rules on the right edge tend to become zero.

The bottom right rule is similar to the inverse of the top left rule. It only cares above the bits directly above and directly on the left. If all of those bits are 1, then it will stay a 1. otherwise, it will either become a zero or stay a zero.

The top right rule will switch from a zero to a one only when the two bits below it are 1s. It will switch from 1 to a zero if the the cell on the left is a zero. It will stay a 1 if the bit on the left is a 1 and it will stay a zero if either of the bits below are a zero.

The right edge rule will become a zero if the bit on the left is a zero or the bit above is a zero. It will become a 1 if the two bits below are 1s. It will stay a zero if either of the bits below are zero. It will stay a 1 if the bits above and to the left are also 1.

3) Rules for the top and bottom edges: The top edge will tend towards 1.

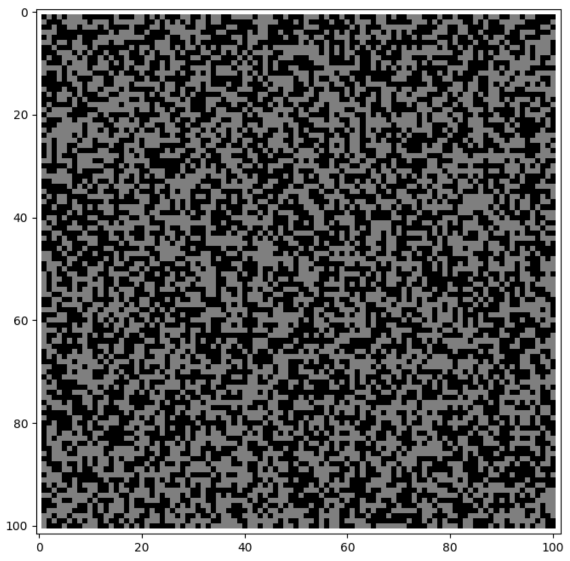

Fig. 3: A random 2D array of alive (black) and dead(grey) cells. Here each cell had an equal probability of being alive or dead.

It will remain a 1 if it's the case that the left bit is a 1. It will become a zero if it has a 1 and the cell on the left is a zero. It will become a 1 if it's the case that its neighbors in the bottom left corner and directly below are 1 or if the bit on the right is a 1. In every other case, it will stay a zero.

The bottom edge cells will remain a 1 if the bit on the left is a one and if either the bit directly above or top right neighbor are a 1. It will become a 1 if it's the case that the bit on the right is a 1. It will become a zero if the bit on the left is a 0 or if the bit directly above and the bit in on the top right are zeroes. In other cases it will remain zero.

4) The Center Rule: This is the most complex of the rules. It will become a 1 if it's the case that the bit on the right is a 1, or the bit directly below and on the bottom left are a 1. It will become a zero if the bit on the left is a one or the bit above is a zero and the bit on the top right is a zero. It will stay zero if it's the case that the bit on the right is a zero and the bit below or the bottom left bit are zero. It will become a one in all other cases.

V. 3 Dimensions and beyond

We can see from the 1 and 2 dimensional cases that a 3 dimensional case is possible as well. What we could do is define the following ordering for passing 1s. First, pass to the left if it's possible, then pass up if it's safe and you cannot pass left, finally, pass although the "z" axis if it's safe and both up and left are unavailable. To have a CA produce two separate regions of alive and dead cells, it's enough to define preferences among the possible directions a 1 can travel. This

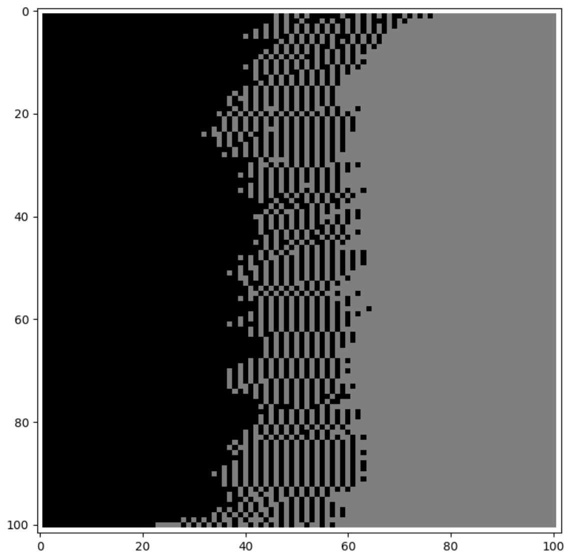

Fig. 4: The array beginning to separate into the two regions. Because we give a preference for the alive cells in the array to move left before moving up, we see the regions form on the left and right sides

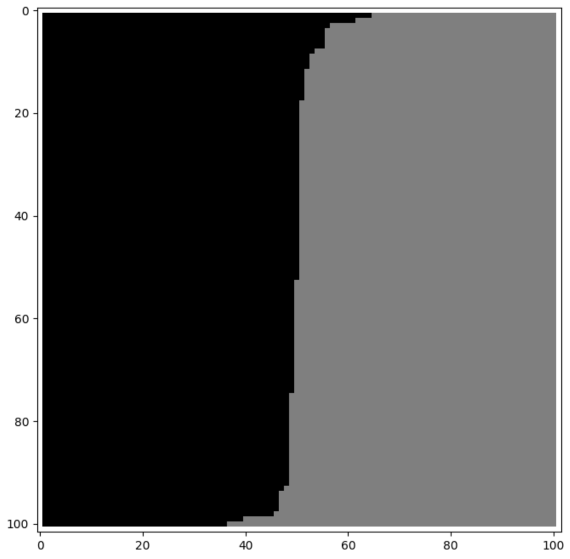

Fig. 5: The ending result of the array being partitioned. We noticed that the alive cells tend to cluster more towards the top, leaving a gap at the bottom. This is because when cells can no longer move left, they instead move up, leaving gaps at the bottom.

same principle could be applied in any number of dimensions. In all cases, the radius only needs to be 1. The size of these rules would be 3^n, where n is the number of dimensions, not counting the number of edges cases, nor excluding the bits which aren't useful to the automata. However, we find that with our genetic algorithm that generating rules above 128 bits is likely beyond our means. It's possible an algorithm could be used to generate these rules, possibly done by augmenting the two dimensional rule with a third dimension and associated changes, but that is a subject of future work.

VI. 3 Colors in 1D

We have found that 3 colors in one dimension cannot be solved with a neighborhood size of 3. After allowing our genetic algorithm to attack the problem, we found that it could only find solutions to problems that don't have all three colors present in one rule. This makes sense. While we can say that the first color (0) should be on the right, the second on the left (1), and the third in the middle (2), the following case makes a correct answer impossible: 021. We can see that this lacks a rule which all three neighbors could agree on. The 0 would want to exchange with the 2, and the 1 would want to exchange with the 2. This means that whatever the 2 picks, information will be lost. It's possible, maybe even likely, that a neighborhood size of 5 could possibly find a solution, but we lack the computing power and the time to attempt to create such a rule.

VII. Conclusions

We have shown that a solution to the separation problem for cellular automata in both 1 and 2 dimensions only requiring a radius of 1. We also present a case for solutions of radius 1 existing in any number of dimensions. The 1 Dimensional rule is a simple problem, which only requires a 16 bit rule and can be solved by the genetic algorithm in only 6 iterations. The 2 Dimensional rule is quite a bit larger at 304 bits, and we were unable to get the genetic algorithm to generate a solution, but were able to write out a solution by hand. The key to solutions in higher dimensions is to give a preference order for each direction you want to move the ones in the problem.

A. Genetic Algorithms and the Separation problem

Something we discovered part of the way through this project is that attempting to solve these problems via a genetic algorithm is actually a generally poor idea. Separation problems for cellular automata have a small set of solutions which fully solve the problem. A genetic algorithm tends to be a better solution to a problem that may lack such a solution and improvement is a better step. The majority classification problem is a good example of this. There is no known solution, but there are a lot of ways for the automata to improve. Cases are also easily generated.

Our attempted genetic algorithm is also a poor choice. Most of these solutions have long strings of ones and zeroes. We randomly flip the bits in the rules, but a better algorithm would focus more on moving longer sequences of bits and finding a correct ”pattern”.

B. *Genetic algorithms in python*

Throughout our experiments, we dealt with an issue in the speed of our algorithm. Evolution steps took quite a long time, and this is likely do to our use of python. We would suggest to anyone who wants to continue research on this problem to switch a different library. C++ has many such libraries. These would likely run faster and would allow the user to test larger cases at a faster rate.

C. *Beyond Cellular Automata*

What can this problem say beyond cellular automata. What it clearly shows is that communication of data can be completed with relatively simple rules and only local communication. One could imagine applying the same reasoning to a network where we may want to transfer data from one side of the network to the other. This shows that we need only need a very small amount of memory and knowledge of our neighboring nodes. It is important to recognize that we only worked in 1 and 2 dimensions, so the network would have to have a particular structure. Nevertheless, it would be interesting to attempt to solve this problem on multiple types of graphs, which provide a better approximation for many networks.

References

[1] DEAP - Distributed Evolutionary Algorithms in Python. https://github.com/deap/deap

[2] M. Mitchell, Complexity: A guided tour. Oxford: Oxford University Press, 2009.

[3] B. Waggener, W. Waggener, W. M. Waggener, Pulse Code Modulation Techniques. New York, United States: Springer Publishing, 1995.

VIII. Contribution Statement

A. *Matthew Timm*

Matthew wrote the code for the cellular automata and the genetic algorithm. He hand solved the rules for 1 dimensional and 2 dimensional cases. Matt wrote code to make a GIF of the 2 dimensional case. He also wrote the the methods and results sections of the paper.

B. *Sean Timm*

Sean wrote the code to display the 1 dimensional and 2 dimensional cases. He wrote the introduction, the abstract, and the conclusion sections. He handled most of the proof reading and the bibliography.