Columbus State University

CSU ePress

Theses and Dissertations

Student Publications

5-2016

An Exploration of Rule Clustering in Cellular Automata Rule Spaces

Jordon Huffman

Follow this and additional works at: https://csuepress.columbusstate.edu/theses_dissertations

Part of the Computer Engineering Commons

Recommended Citation

Huffman, Jordon, "An Exploration of Rule Clustering in Cellular Automata Rule Spaces" (2016). *Theses and Dissertations*. 243.
https://csuepress.columbusstate.edu/theses_dissertations/243

This Thesis is brought to you for free and open access by the Student Publications at CSU ePress. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of CSU ePress.

AN EXPLORATION OF RULE CLUSTERING IN CELLULAR AUTOMATA RULE SPACES

Jordon Huffman

Columbus State University

D. Abbot Turner College of Business

The Graduate Program in Applied Computer Science

An Exploration of Rule Clustering in Cellular Automata Rule Spaces

A Thesis in

Applied Computer Science

By

Jordon Huffman

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2016

©2016 by Jordon Huffman

I have submitted this thesis in partial fulfillment of the requirements for the degree of Master of Science.

5/10/2016 Date	Rodrigo Obando Associate Chair / Associate Professor Thesis Advisor
5/9/2016 Date	Rania Hodhod Assistant Professor
5/12/2016 Date	Eugene Ionascu Professor
5/11/2016 Date	Shamim Khan Professor
5/10/2016 Date	Hoda Mehrpouyan Assistant Professor
5/9/2016 Date	Yeşem Peker Assistant Professor

Table of Contents

Table of Figures 4

Abstract 5

1 Introduction 6

2 Cellular Automata 7

3 Related Work 9

4 Current Research 10

4.1 Initial Findings 10

4.2 Defining a Similarity Measure 13

4.3 Dynamic Bits 16

4.3.1 Recursive Algorithm 16

4.3.2 PrintSimilarBits[] 17

4.3.3 FindSimilarRules[] 19

5 Results 21

5.1 Verification 21

5.2 Validation 22

6 Future Work 26

7 Limitations 27

8 Conclusion 28

Code Appendix 29

References 35

Table of Figures

Figure 1 – From Left to Right: Class 1 and Class 2 (above) 7
Figure 2 – Rule 30 Evaluation 7
Figure 3– 3-tuple Cube 12
Figure 4– Similar Sierpiski Triangles 12
Figure 5– Computer Vision Comparison 14
Figure 6– Key Bits Positions of Standard Sierpinski Triangle 17
Figure 7– Key Bits Positions of Neighboring Sierpinski Triangle 17
Figure 8– Navigator on Standard Rule 18
Figure 9– Navigator on Neighboring Rule 19
Figure 10– Cluster Id 19
Figure 11– Similar Sierpinski Triangle Rules 20
Figure 12– Equivalent Class Tests 22
Figure 13– FindSimilarRules[] Contains All Recursively Identified Rules 22
Figure 14– Small Set of Similar Rules from Beginning 24
Figure 15- Small Set of Similar Rules from Ending 25
Figure 16– And Operator Across Recursively Found Rules 25

Abstract

The study of complex systems examines the global behavior of a system and how the individual parts of the system affect that behavior [1]. The study of complex systems spans across many fields of science like biology, physics, engineering, and computer science. One area of complex systems that has not been fully explored is cellular automata. Since its discovery by John von Neumann, there have been no consistent ways of categorizing similarities between cellular automata rules or collecting similar rules for observation. This thesis introduces an approach to identifying clusters of similar rules and extracting rules from that cluster. Several similarity measures were developed to establish similarity between rules. All similarity measure approaches are outlined in this thesis, but only one was selected for determining similarity in this approach. Based on a partitioning of the rule space, this approach uses λ_0 and λ_1 with their inherent primitives p_0 and p_1 to obtain a cluster identification string [5]. The cluster Id. is determined by the output of the surrounding neighbors of any rule in the cluster. This cluster Id. can be used to produce a set of rules, all yielding the same or similar output.

1 Introduction

Cellular automata can be classified as simple systems with deterministic input. However, these systems can generate complexity in their output. Cellular automata can appear in several dimensions depending on initial input. The formula for 1-dimensional cellular automata rule space creation is $k^{k^{(2r+1)}}$ where k is the number of states and r is the radius of cells to examine on both sides. The elementary cellular automata rule space uses $k = 2$ (binary: 0, 1) and $r = 1$ (three cells to use: central, left, and right). The elementary rule space is $2^{2^{(2(1)+1)}} = 256$. By moving to the next radius size of 2, the rule space jumps from 256 rules to $2^{2^{(2(2)+1)}} = 2^{2^5} = 4{,}294{,}967{,}296$ rules. With such a large space, traversal and examination have been limited to random sampling. Initially, cellular automata were thought to have random distribution on the space; however, recent work has shown that partitioning the space into a partially ordered system creates clusters of similar rules [2] [5]. To the best of our knowledge, no application or methodology for collecting similar rules exists. This thesis uses the foundation of rule partitioning [5] to develop a framework for collecting similar rules. This framework allows any similarity measure to be employed for determining rule similarity on the premise that the chosen similarity measure for this thesis can be improved upon, or replaced completely, to insure a more accurate similarity measure.

2 Cellular Automata

Wolfram classified cellular automata rules into one of four classes: Class 1, 2, 3, and 4 [9]. Class 1 is a uniform or homogenous state. Class 2 is a semi-uniform state where localized patterns may appear, yet they are simplistic. Class 3 is a random state of disordered output. Class 4 is a complex state. Class 4 can be seen as a mixture of both Class 2 and Class 3, where there are seemingly semi-uniform patterns and random patterns. Wolfram describes Class 4 as an "intermediate phenomenon" [8]. Below are examples of these classes:

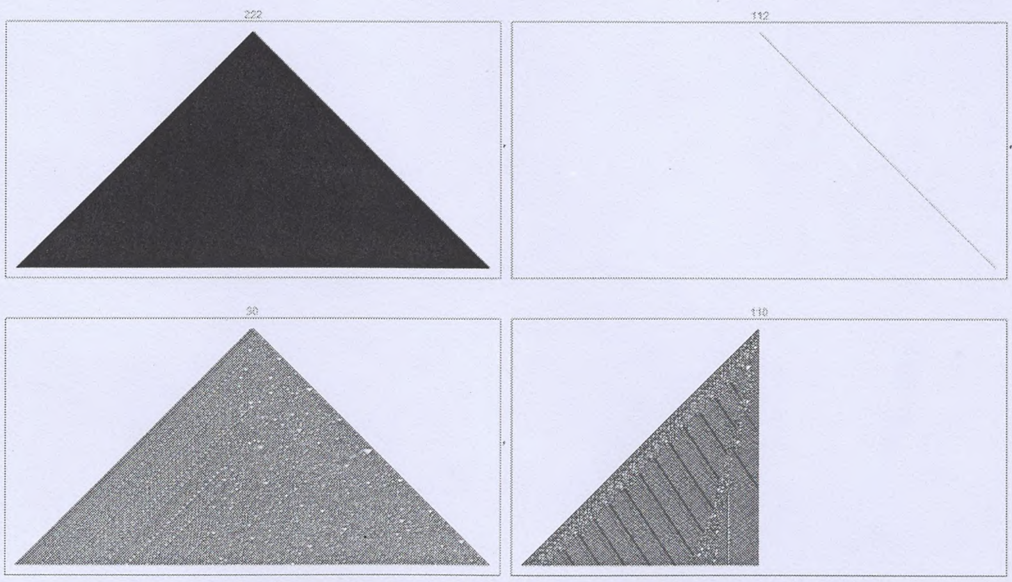

Figure 1 – From Left to Right: Class 1 and Class 2 (above)

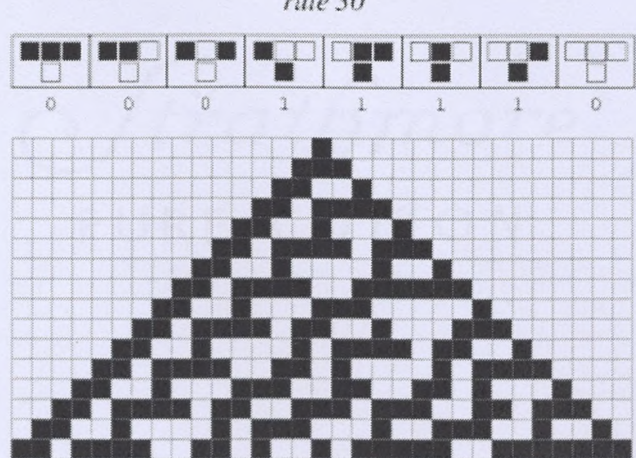

Figure 2 – Rule 30 Evaluation

Cellular automata evaluation begin with the rule number. Figure 2 shows the cell evaluation parameters for a two state cellular automaton at radius 1 (k = 2 and r = 1). The first row is the initial condition, where the middle bit is set to 1 (black) and all other bits are 0 (white). All rules are converted into their respective binary representations. The seond row is evaluated according to the radius. In Figure 2, all cells in the first row are matched to the corresponding cell evaluation parameter to produce the cell in the next row. The cell evaluation applies to each consecutive row to produce the output over a given number of steps. This procedure shows an evaluation over time.

3 Related Work

Many areas of cellular automata are currently being researched. Imre et al [3] show how quantum-dot cellular automata can be used as nanomagnets for logic functionality at the near subatomic level. Sipper has completed research in non-uniform cellular automata [7]. These automata use a different rule per cell to evaluate each row with a genetic algorithm to progressively become more fit. Recently, Obando partitioned the rule space using a new set of parameters [5]. He defines them as λ_0 and λ_1. Where Langston defines λ to be the ratio of ones in the rule to the total number of possible ones [4], Obando defines λ_0 as the number of ones in primitive zero and λ_1 as the number of zeros in primitive one, giving primitives p_0 and p_1 [5].

To the best of our knowledge, the contents of this thesis, with all of its new ideas, algorithms, and code, do not exist anywhere else. Wolfram acknowledges that "find[ing] families of cellular automaton rules with closely related behaviour" should be possible, but no known work has conclusively done so [8]. Also, please note that the approach outlined in this thesis stems directly from the work presented by Obando. Many functions used in the development, testing, and execution stages of our experiments make use of his Mathematica package. It was through his guidance and cooperation that this thesis was written.

4 Current Research

4.1 Initial Findings

Many researchers focus on specific cellular automaton rules, often finding that individual behavioral patterns can be applicable to other fields of science. This thesis focuses on the general behavior of the entire space. We examine the global behavior of the entire rule space to find patterns. Through the partitioning of the rule space [5], it can now be defined as a partially ordered system based on the binary primitives of the rules. In previous work, traversal of the rule space is achieved using a navigational tool [Huffman]. This tool shows the primitives of the rule in question as well as its neighboring rules and their primitives. Investigation into the space shows the likelihood of rule “clusters”. These clusters appear evenly throughout the rule space. As the navigator approaches the edge of the cluster, rules belonging to neighboring clusters appear. One can transition into the approaching cluster, revealing more rules belonging to that cluster. The existence of clusters in this cellular automata system gives rise to several questions that will be answered throughout this thesis. Firstly, what defines a cluster? Secondly, is each cluster unique? If so, what is the cluster identifier? Thirdly, can a rule pertaining to a specific cluster give information about the global behavior of that cluster?

When examining the primitives of each rule and its neighbors, each cluster seems to be ‘activated’ by flipping certain bits in the primitives. By definition of the partially ordered system, each neighbor of the central rule only differs by one bit. Hovering over the rules’ images allows one to see the single bit difference per rule via tooltip. All other bits in that local neighborhood are the same. Take Figure 3 for instance. The rule 001 is adjacent (neighbors) with 000, 101, and 011. These numbers have a single bit of difference. Also, the numbers 001, 000, and 101 all have a ‘0’ as the middle bit. These are the same types of observations seen in the 2^{32} rule space.

Because each rule in the space has thirty-two bits, many similar bits are present between neighbors. Take Figure 4, which is rule 2,147,483,649. The primitives are 1 and 32,768 for p_0 and p_1, respectively. The base 2 forms of each are represented by the boxes above the image, with white being 0 and black being 1.

Noticeably different about rule 2,147,483,649 and rule 2,281,701,377 is the first bit position from left to right in p_0 of rule 2,281,701,377. This change does not alter the rule, causing no deviation from the central rule. These two rules can be considered similar, but similarity will be discussed later. According to Obando [5], rule primitives are defined as

λ_0 = number of 1s in binary representation of the primitive

λ_1 = number of 0s in binary representation of the primitive

Let us take all the bit positions in p_0 that are 1 and all the bit positions in p_1 that are 0. In Table 1, the p_0 for both rules are exactly the same, while the p_1 of rule 2,281,701,377 includes the first bit position from the left. From here, we created a "rule crawler" that takes an initial seed (rule number). The crawler begins to collect similar rules based strictly on output; in other words, a bit-by-bit comparison per row. This similarity measure will be discussed in detail later. The sample space returned had 1,160 similar rules at Hamming distance 3. We use a separate algorithm to run a tally of all occurring position for both primitives. This algorithm takes the complement of each rule from the sample against a standard, in this case, rule 2,147,483,649. The result of this algorithm gave us $p_0 = \{16\}$ and $p_1 = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$. Running the same algorithm with the same sample space on rule 2,281,701,377 gave us the exact same primitive bit positions: $p_0 = \{16\}$ and $p_1 = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$. Matching primitive bit positions for visually similar rules implies that there exists a

correlation between particular cellular automata output (images) and the primitive bit positions for those images. We will call this the cluster identification.

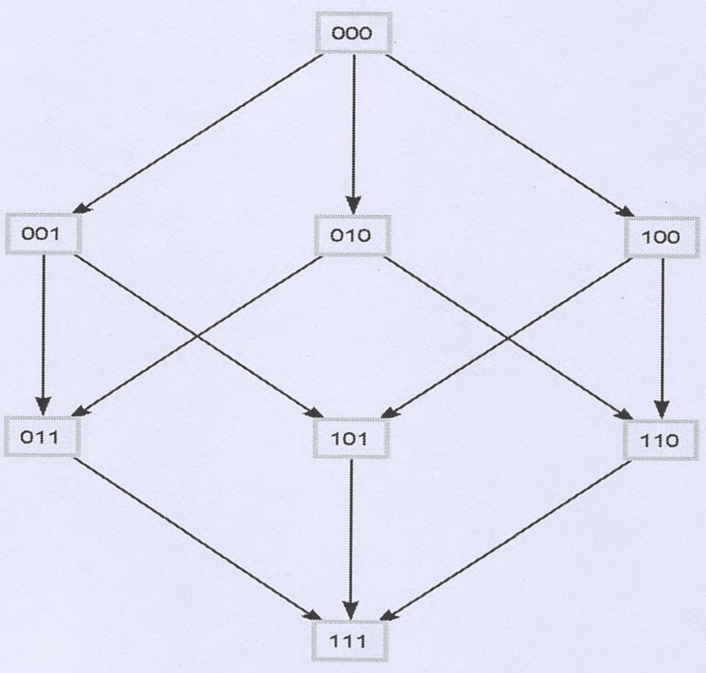

Figure 3– 3-tuple Cube

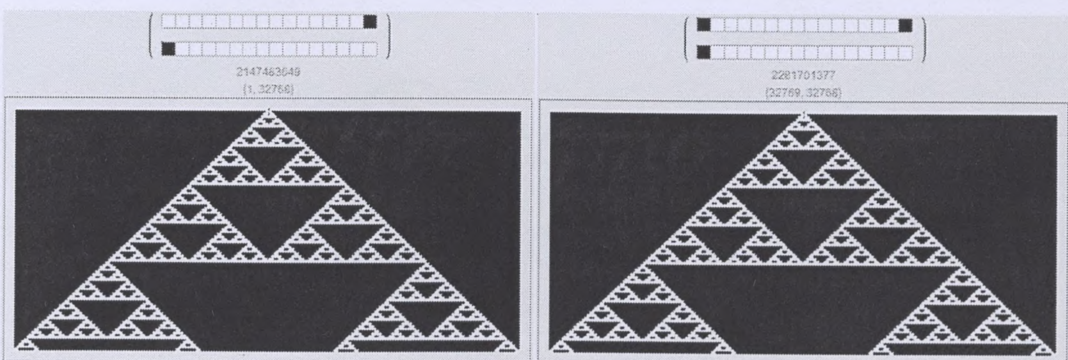

Figure 4– Similar Sierpiski Triangles

Rule	p_0	p_1
2,147,483,649	{16}	{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
2,281,701,377	{1, 16}	{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
2,147,483,649 - Algorithm	{16}	{2, 4, 9, 13, 16}
2,281,701,377 - Algorithm	{16}	{2, 4, 9, 13, 16}

Table 1– Primitive Comparison

After further testing, we realized that visually similar rules running through the algorithm all produce the same p_0 and p_1. This particular algorithm requires a large sample space against which to test. The creation of this sample space is both time consuming and computationally expensive. Thus, a different approach must be made that is fast and efficient. Let us start by defining a similarity measure.

4.2 Defining a Similarity Measure

The best similarity measure between rules can be a difficult topic for discussion. So many different methods exist for image processing like edge detection, symbol recognition, and digital morphology [6]. To properly categorize similarity, a similarity measure must be used that quickly and efficiently determines rule similarity. We took three different approaches to find the best method: Mathematica's built-in image processing components, comparison by complete output, and comparison by step.

Mathematica has rich built-in image processing capabilities. These include, but are not limited to: color manipulation, geometric operations, morphological operations, filtering and neighborhood processing, and computer vision. We used a computer vision function called Image Feature Track that automatically detects set points between different images. We took a

standard rule and compared that to each subsequent rule of our choosing. The result was images with an overlay of similar points in red as shown in Figure 5. The images of rules 2,147,483,649 and 2,281,701,377 are similar according to the Image Track Feature. The problem with using computer vision in this manner is that the Image Track Feature does not give the programmer much control of how these "similar points" are chosen. Instead, full reliance is on Mathematica to be the ultimate judge of similarity. We abandoned this technique in search of an approach with more flexibility.

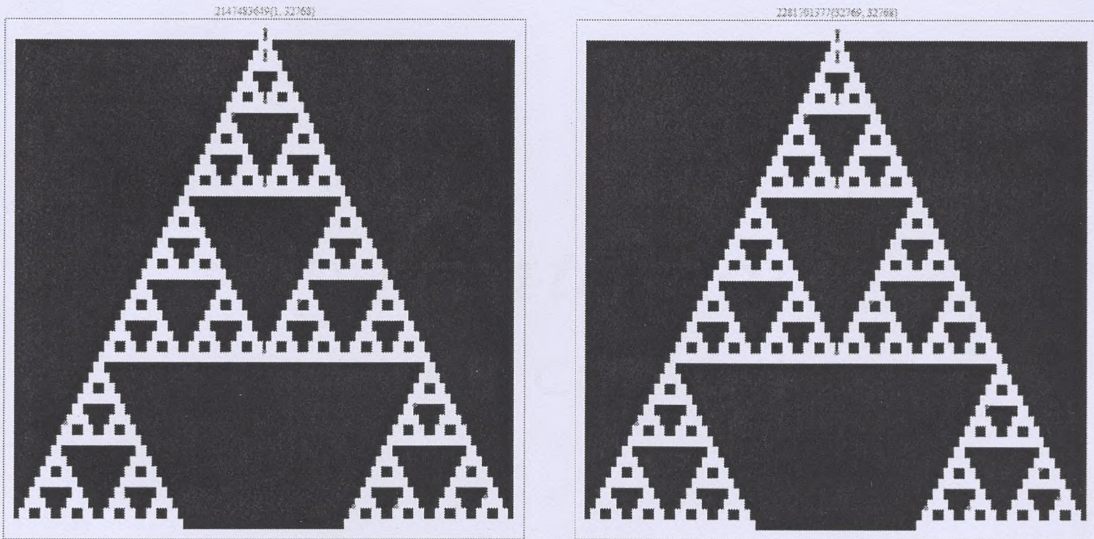

Figure 5- Computer Vision Comparison

For the second approach, we decided to work directly with the binary numbers composing the cellular automata. Our goal was to define an algorithm that directly compares the output of one rule with the output of another rule, which we call a comparison by output. Remember that these cellular automata images are merely a visual representation of 0s and 1s. Moreover, Mathematica represents cellular automata using a list of lists, where each inner list is the corresponding row in the image. The comparison by output uses this knowledge and calculates similarity based on each row of output. This process first generates each list of lists for both rules. Then, it programmatically loops through each inner list taking the sum of the

differences of each row, divides by the length of the row, subtracts this value by one, and gets the mean of all the rows. This equation to find similarity is denoted as s, where *st* is the standard rule, *tr* is the target rule, *l* is the length of the row, *j* is the index of the binary digit in the row, *i* is index of the row, and *n* is the number of rows. Note that Mathematica indexes its lists starting at 1.

$$s = \frac{\sum_{i=1}^{n}(1 - \frac{\sum_{j=1}^{l} |st_{i,j} - tr_{i,j}|}{l})}{n}$$

This approach gives us a direct similarity measure based on the locations of 0s and 1s in each row of output. For example, take the lists {{0, 0, 0, 0, 1}, {1, 0, 0, 0, 1}} and {{0, 0, 0, 0, 0}, {0, 0, 1, 0, 0}}. We will assume that these are our standard and target, respectively. Row_1 is then subtracted, {0, 0, 0, 0, 1} - {0, 0, 0, 0, 0} = {0, 0, 0, 0, 1}. The absolute value of the sum of this result is 1 divided by 5, the length of the row. Then, we take $1 - \frac{1}{5} = \frac{4}{5}$, which is the similarity of the standard's and target's row_1. The same procedure is done for row_2 of each list to get $\frac{2}{5}$ similarity. The last step is to get the mean of all these similarities, i.e. $\frac{\frac{4}{5}+\frac{2}{5}}{2} = \frac{3}{5}$.

The third approach used a technique similar to the second approach. The major difference being that the initial condition of the target rule was reset as the previous row in the standard rule's output. We called this similarity by step. So, the similarity changed from an exact comparison by appearance to a comparison of the rate of change from one row to the next. We followed the same formula from similarity by output. Such a measure can be useful for detecting how rules from one cluster globally change into another cluster. However, this approach was also

abandoned as it is beyond the scope of our current objective. We decided to use the comparison by output as the similarity measure for our algorithm.

4.3 Dynamic Bits

As mentioned before, a pattern began to emerge between certain bit locations in the primitives of the rules. With our new similarity measure in hand, we compared many different rules against one another for testing. Images like the Sierpinski triangles were found to not only be similar but also to be the exact same output. Our tests used a similarity measure of 100 percent (1.0) similarity. This means that different rules like 2,147,483,649 and 2,281,701,377 create the exact same output. This discovery raises a question. Can we create a search algorithm to find similar rules given our new similarity measure?

4.3.1 Recursive Algorithm

The creation of the search algorithm began using a recursive function that tracked the visited rules, similar rules, and next level of rules. The algorithm started with a single rule, the standard rule, and a threshold of similarity. It began to traverse the rule space outwardly, essentially becoming a 'rule crawler'. First, the crawler finds the neighbors of all incoming rules. These neighbors were complemented to the list of visited rules. The resulting list consisted of only neighbors that have yet to be visited. The crawler compared all the remaining rules to the standard. If the comparison between a target rule and the standard meet or surpass the threshold, the target is considered similar to the standard and is stored in a list. All targets are appended to a separate list for visited rules. Certain clusters produced tens of thousands of rules, which took too much time for rapid testing. So, a cutoff point of Hamming distance of 3 was implemented to decrease search time. When we refer to Hamming distance, we simply mean a certain number of steps away from the standard. A Hamming distance of 1 would be immediate neighbors of the

standard, while a Hamming distance of 2 would be 2 steps away for the standard. Even with a cutoff of Hamming distance 3, our algorithm took over 90 seconds to run to completion for some clusters. These long run times coupled with extreme resource consumption caused us to examine a different approach.

4.3.2 PrintSimilarBits[]

The recursive algorithm's results led to finding that certain bit positions were persistent across all rules found We defined a function to identify key bit positions in a rule. These positions come from breaking the rule into its primitives. Then, one must determine the positions of 1s occurring in primitive 0, and 0s in primitive 1. The resulting list contains key bit positions for both p_0 and p_1. In Figure 6, we show a function to take the running complement of the similar rule set from our recursive algorithm against the same standard used to find the similar set. This similar set is called 'many' in our code and contains 1,160 similar Sierpinski triangles. The output of PrintSimilarBits[] in Figure 6 shows that the 16th position of p_0 occurs in every p_0 in the set. Likewise, 2, 4, 9, 13, and 16 all occur in every p_1 in the set. Therefore, these positions are required to create this cluster of Sierpinski triangles. To validate that this process is correct, we run a separate rule that was chosen at random within the similar set, as seen in Figure 7. Notice that the p_0 is different, as it should be, but the result is still {{16}, {2, 4, 9, 16}} for the key bits in the cluster.

```
In[63]:= PrintSimilarBits[2147483649, many, 2, 1]
Out[63]= Standard Key Bits: {{16}, {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}}
         Targets Combined Key Bits: {{16}, {2, 4, 9, 13, 16}}
```

Figure 6– Key Bits Positions of Standard Sierpinski Triangle

```
In[66]:= PrintSimilarBits[2182090753, many, 2, 1]
Out[66]= Standard Key Bits: {{3, 16}, {2, 3, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 16}}
         Targets Combined Key Bits: {{16}, {2, 4, 9, 13, 16}}
```

Figure 7– Key Bits Positions of Neighboring Sierpinski Triangle

PrintSimilarBits[] can correctly determine the key bits of a cluster. However, it also requires a set of previously defined similar rules. Remember that this set was derived from the long-running recursive algorithm. A different, more concise method must be adopted to correctly identify the key bits in the cluster. After further examination in the navigational tool, it came to light that certain neighbor positions consistently moved out of the current cluster. That is to say, no matter where one moves in the cluster, a particular position will always move out of the cluster. In Figure 8, the standard rule from previous tests is the central rule in the navigator. Notice the positions where the right neighborhood moves out of the cluster (i.e. not a Sierpinski triangle). Figure 9 shows the rule 2,281,701,377 as the central rule. This rule was in the last position of the right neighborhood. Notice that, even when moving to a different rule, the four positions that move out of the cluster stay the same from rule to rule. This is generally the case while moving throughout the cluster.

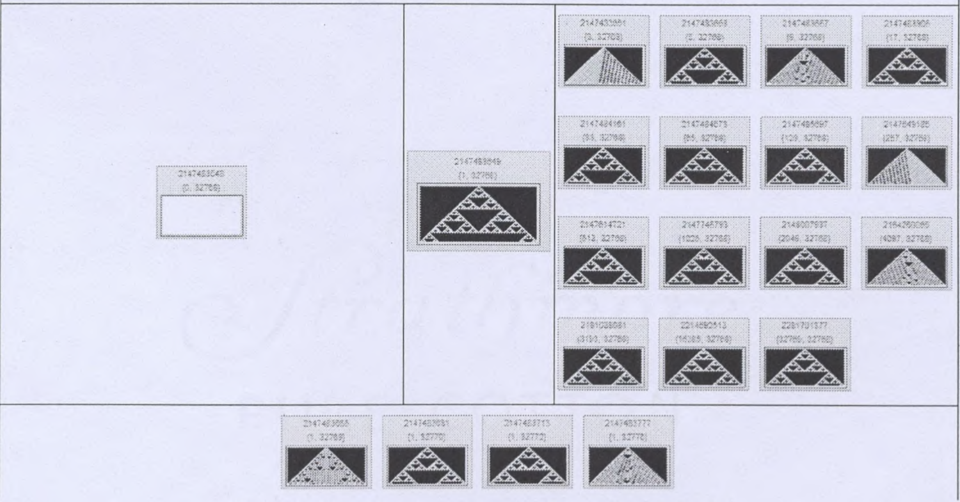

Figure 8– Navigator on Standard Rule

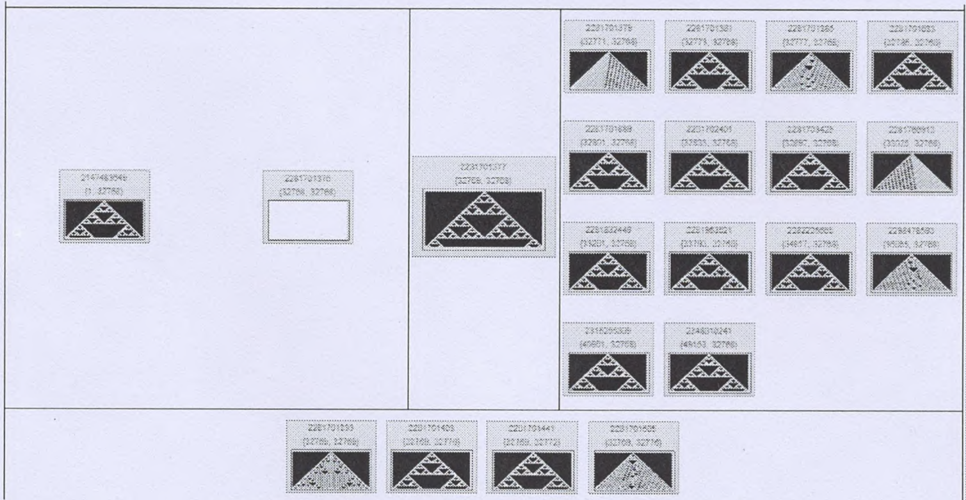

Figure 9– Navigator on Neighboring Rule

4.3.3 FindSimilarRules[]

Now that we have seen the general behavior of rule neighborhoods, we move on to the next algorithm called FindSimilarRules[]. Instead of processing the similarity of a large set, we only process the similarity of the immediate neighbors of the standard rule. Remember that each neighbor differs from the central (standard) rule by only a single bit. Each neighbor is tested against the standard for similarity by output. If the target is considered similar, the position of the different bit is considered dynamic and a running intersection of the key bits is taken. Dynamic, in this sense, means that the bit could be 0 or 1, and the output would not change. If the target is not similar to the standard, the differing bit position is marked as negatively affective because that bit position moves the output away from the current cluster. Once all neighbors have been tested, the key bit positions are marked as positively affective because these bits must be flipped to create the cluster. We called this function ClusterID[], and its output can be seen in Figure 10.

```
In[66]:= ClusterID[2147483649, 2]
Out[66]= {{D, D, D, N, D, D, D, N, D, D, D, N, N, N, N, P}, {N, P, D, P, D, D, D, D, P, D, D, D, P, D, D, P}}
```

Figure 10– Cluster Id

The cluster Id. allows us to find similar rules. We must convert the Id. into binary. In primitive p_0, a ‘P’ means the binary 1 goes there because the position is required to be flipped. An ‘N’ in p_0 means a binary 0 goes there because the position cannot be flipped. The exact opposite is true of p_1. Where there is a ‘P’, the binary 0 is used. Where there is an ‘N’, the binary 1 must be used. The interesting part of this process is when converting ‘D’ back into a binary number. Remember that the ‘D’ was given to all bit positions that were considered similar. So, this position can be 0 or 1. Once we lock the Ps and Ns appropriately, we apply a mask to the cluster Id. This mask generates all the binary numbers possible given that ‘D’ positions can flip. For example, take a simple cluster Id. {N, P, D, D, P}, where Ns will lock to 0s and Ps will lock to 1s. After we lock the bits, the result is {0, 1, D, D, 1}. Now we generate all the binary numbers around that to get {01001, 01011, 01101, 01111}. Figure 11 shows a run of FindSimilarRules[].

```
In[181]:= FindSimilarRules[2147483649, 2, 1]

Out[181]= {2147483649, 2147483681, 2147483713, 2147483745, 2147484161, 2147484193, 2147484225, 2147484257, 2147484673, 2147484705,
 2147484737, 2147484769, 2147485185, 2147485217, 2147485249, 2147485281, 2147485697, 2147485729, 2147485761, 2147485793,
 2147486209, 2147486241, 2147486273, 2147486305, [illegible], 2935911425, 2935911457, 2935911489, 2935911521,
 2935911937, 2935911969, 2935912001, 2935912033, 2935912449, 2935912481, 2935912513, 2935912545, 2935912961, 2935912993,
 2935913025, 2935913057, 2935913473, 2935913505, 2935913537, 2935913569, 2935913985, 2935914017, 2935914049, 2935914081}

large output | show less | show more | show all | set size limit...
```

Figure 11– Similar Sierpinski Triangle Rules

5 Results

The next section of the thesis outlines the verification and validation stages of our results. These are the questions we asked: “Are we building the product right?” and “Are we building the right product?” In order to answer these questions, we compare the results of our algorithm to the original specifications and to other clusters.

5.1 Verification

Can we design an algorithm that efficiently identifies and collects similar rules in a cellular automata rules space? The answer is a resounding yes. We have proven that, under the partially ordered system proposed by Dr. Rodrigo Obando, a cluster of similar rules can be identified. At the same time, similar rules can be extracted from the rules space. One function created by Dr. Obando, called EquivalentClass[], allows one to find the exact reflection, complement, and reflection-complement of any given rule. In Figure 12, rule 2,147,483,649 at radius 2 has been passed as arguments. The result of this function call returns only 2 rules, itself and 2,147,483,646. Since the reflection (across the y axis) of a Sierpinski triangle is exactly the same as the original, only the complement and the original are returned. Both rules have been plotted to show their differences. Next, the ContainsAll[] function takes as arguments the rule set produced by FindSimilarRules[] and 2,147,483,646. The result of ContainsAll[] is false showing that the complement rule, 2,147,483,646, does not exist in the similar set. Only rules fitting the exact output pattern of the standard rule appear in the output of the FindSimilarRules[] function. Also, we can further verify the accuracy of FindSimilarRules[] by using the rules set generated by the recursive algorithm from earlier trials. By using the ContainsAll[] function again in Figure 13, we prove that all of the 1,160 similar rules found by the recursive algorithm appear in the output of FindSimilarRules[].

```
EquivalentClass[2 147 483 649, 2]
{2 147 483 646, 2 147 483 649}

ArrayPlot[CellularAutomaton[{#, 2, 2}, {{1}, 0}, {50, All}], PlotLabel → #] & /@ EquivalentClass[2 147 483 649, 2]
```

2147483646 2147483649

```
ContainsAll[FindSimilarRules[2 147 483 649, 2, 1], {2 147 483 646}]
False

FindSimilarRules[#, 2, 1] & /@ EquivalentClass[2 147 483 649, 2]

{{2 038 530 462, 2 038 530 494, 2 038 530 526, 2 038 530 558, 2 038 530 974, 2 038 531 006, 2 038 531 038, 2 038 531 070, 2 038 531 486, 2 038 531 518, 2 038 531 550, 2 038 531 582,
  2 038 531 998, 2 038 532 030, 2 038 532 062, 2 038 532 094, 2 038 532 510, 2 038 532 542, 2 038 532 574, 2 038 532 606, 2 038 533 022, 2 038 533 054, 2 038 533 086, 2 038 533 118,
  2 038 533 534, 2 038 533 566, 2 038 533 598, 2 038 533 630, 2 038 534 046, 2 038 534 078, 2 038 534 110, 2 038 534 142, 2 038 534 558, 2 038 534 590, 2 038 534 622, 2 038 534 654,
  2 038 535 070, 2 038 535 102, 2 038 535 134, 2 038 535 166, 2 038 535 582, 2 038 535 614, 2 038 535 646, 2 038 535 678, 2 038 536 094, 2 038 536 126, 2 038 536 158,
  2 038 536 190, 2 038 536 606, 2 038 536 638, 2 038 536 670, [illegible], 2 147 477 438, 2 147 477 470, 2 147 477 502, 2 147 477 918, 2 147 477 950, 2 147 477 982, 2 147 478 014,
  2 147 478 430, 2 147 478 462, 2 147 478 494, 2 147 478 526, 2 147 478 942, 2 147 478 974, 2 147 479 006, 2 147 479 038, 2 147 479 454, 2 147 479 486, 2 147 479 518,
  2 147 479 550, 2 147 479 966, 2 147 479 998, 2 147 480 030, 2 147 480 062, 2 147 480 478, 2 147 480 510, 2 147 480 542, 2 147 480 574, 2 147 480 990, 2 147 481 022,
  2 147 481 054, 2 147 481 086, 2 147 481 502, 2 147 481 534, 2 147 481 566, 2 147 481 598, 2 147 482 014, 2 147 482 046, 2 147 482 078, 2 147 482 110, 2 147 482 526,
  2 147 482 558, 2 147 482 590, 2 147 482 622, 2 147 483 038, 2 147 483 070, 2 147 483 102, 2 147 483 134, 2 147 483 550, 2 147 483 582, 2 147 483 614, 2 147 483 646}, {[illegible]}}

large output   show less   show more   show all   set size limit...
```

Figure 12– Equivalent Class Tests

```
In[183]:= ContainsAll[FindSimilarRules[2 147 483 649, 2, 1], many]

Out[183]= True
```

Figure 13– FindSimilarRules[] Contains All Recursively Identified Rules

5.2 Validation

Validation asks if FindSimilarRules[] correctly produces the expected output from the user. Does our algorithm actually deliver similar rules from the rule space? Figure 12 from above, displays a large output of 524, 240 rules. A tremendous set of Sierpinski triangles like that would be far too large to display in this thesis. Since we proved that all the rules generated by our recursive algorithm are also contained in the output of FindSimilarRules[], we decided to show a sample of rules from that set instead. Figures 14 and 15 are the first 104 and last 84 rules, respectively. All of the rules from the recursive algorithm yield the exact image compared to the standard, but to prove the FindSimilarRules[] yields exact matches, we create the test found in Figure 16. This test first collects the results of FindSimilarRules[]. Next, it maps all of those rules into a comparison functions built into Mathematica called SameQ[]. SameQ tests two arguments for an exact match. What we have done is evaluate the standard rule 2,147,483,649

and every rule from FindSimilarRules[] (stored as the variable 'sim') as arguments into SameQ[]. The result of this mapping is a list of truth values. Finally, we use the Apply[] function to apply the And operator across the entire truth values list, which returns back True. This value of True proves that the entire set given by FindSimilarRules[] is an exact match to the standard rule when a measure of 100 percent is used in FindSimilarRules[].

```
ArrayPlot[CellularAutomaton[{#, 2, 2}, {{1}, 0}, {50, All}], PlotLabel → #] & /@ many
```

2 147 483 649, 2 147 483 681, 2 147 483 713, 2 147 484 161, 2 147 484 673,
2 147 485 697, 2 147 487 745, 2 147 491 841, 2 147 500 033, 2 147 614 721,
2 147 745 793, 2 148 007 937, 2 148 532 225, 2 149 580 801, 2 151 677 953,
2 155 872 257, 2 181 038 081, 2 214 592 513, 2 281 701 377, 2 684 354 561,
2 147 483 745, 2 147 484 193, 2 147 484 225, 2 147 484 705, 2 147 484 737, 2 147 485 185,
2 147 485 729, 2 147 485 761, 2 147 486 209, 2 147 486 721, 2 147 487 777, 2 147 487 809,
2 147 488 257, 2 147 488 769, 2 147 489 793, 2 147 491 873, 2 147 491 905, 2 147 492 353,
2 147 492 865, 2 147 493 889, 2 147 495 937, 2 147 500 065, 2 147 500 097, 2 147 500 545,
2 147 501 057, 2 147 502 081, 2 147 504 129, 2 147 508 225, 2 147 614 753, 2 147 614 785,
2 147 615 233, 2 147 615 745, 2 147 616 769, 2 147 618 817, 2 147 622 913, 2 147 631 105,
2 147 745 825, 2 147 745 857, 2 147 746 305, 2 147 746 817, 2 147 747 841, 2 147 749 889,
2 147 753 985, 2 147 762 177, 2 147 876 865, 2 148 007 969, 2 148 008 001, 2 148 008 449,
2 148 008 961, 2 148 009 985, 2 148 012 033, 2 148 016 129, 2 148 024 321, 2 148 139 009,
2 148 270 081, 2 148 532 257, 2 148 532 289, 2 148 532 737, 2 148 533 249, 2 148 534 273,
2 148 536 321, 2 148 540 417, 2 148 548 609, 2 148 663 297, 2 148 794 369, 2 149 056 513,
2 149 580 833, 2 149 580 865, 2 149 581 313, 2 149 581 825, 2 149 582 849, 2 149 584 897,
2 149 588 993, 2 149 597 185, 2 149 711 873, 2 149 842 945, 2 150 105 089, 2 150 629 377,
2 151 677 985, 2 151 678 017, 2 151 678 465, 2 151 678 977, 2 151 680 001, 2 151 682 049

Figure 14– Small Set of Similar Rules from Beginning

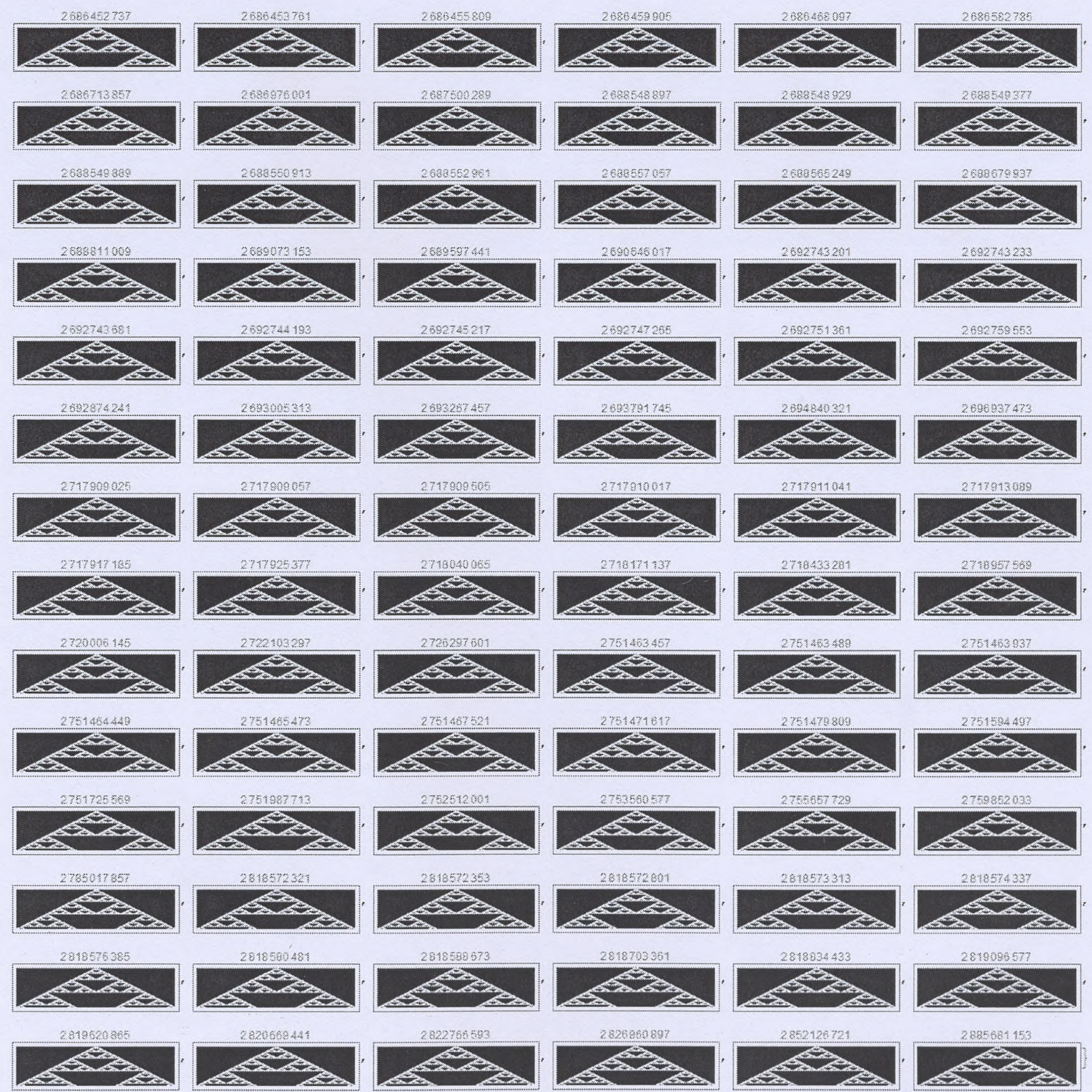

Figure 15- Small Set of Similar Rules from Ending

```
sim = FindSimilarRules[2147483649, 2, 1]
```

```
{2147483649, 2147483681, 2147483713, 2147483745, 2147484161, 2147484193, 2147484225, 2147484257, 2147484673, 2147484705,
 2147484737, 2147484769, 2147485185, 2147485217, 2147485249, 2147485281, 2147485697, 2147485729, 2147485761, 2147485793,
 2147486209, 2147486241, 2147486273, 2147486305, 2147486721, [illegible], 2935911009, 2935911425, 2935911457, 2935911489,
 2935911521, 2935911937, 2935911969, 2935912001, 2935912033, 2935912449, 2935912481, 2935912513, 2935912545, 2935912961,
 2935912993, 2935913025, 2935913057, 2935913473, 2935913505, 2935913537, 2935913569, 2935913985, 2935914017, 2935914049, 2935914081}
```

large output | show less | show more | show all | set size limit...

```
Apply[And, Map[SameQ[CellularAutomaton[{#, 2, 2}, {{1}, 0}, {50, All}], CellularAutomaton[{2147483649, 2, 2}, {{1}, 0}, {50, All}]] &, sim]]
True
```

Figure 16– And Operator Across Recursively Found Rules

6 Future Work

Continued work on general behavior of cellular automata rule spaces could yield interesting results. For instance, one could now use FindSimilarRules[] to generate a particular list of rules. Then, we could apply a different rule across each bit of an initial bit string. The result would be a cellular automaton image where each column uses a different rule. The neighboring rules would 'fight' over the evaluation of each step. Based on our investigations, we hypothesize that this multi-ruled cellular automaton would behave in the same manner as the comprising rule set.

Another possibility of future research could be defining a more precise similarity measure to observe similarity in a different way. Using the comparison by step similarity measure that we created would produce a different rule set because it measures rate of change. This similarity measure could give one insight on how rules change from state to state, or cluster to cluster, inside the rule space. One may begin to ask new question. Can specific bit positions determine cluster dominance? Do bit positions affect the rate of change from one output step to another? One could even relate this research back to Markov Chains. Can a different similarity measure be used to find similar Markov Chains based on dynamic bit positions?

Rule engineering may arise out of the discovery of cluster Ids. A desired output could be generated by manually assigning the locked and dynamic bits. Rule engineering has the potential to change digital art. Instead of hand drawing a texture for wood, a cluster Id. could generate an output that mimics the behavior of wood. The cluster of that rules would also lead to other similar yet unique wood patterns. The possibilities are endless when considering that a 1-dimesional cellular automaton at radius two generates well over four billion images. By simply extending the radius size to three, the output jumps to well over three trillion images.

7 Limitations

An assumption was made in the results displayed in this thesis. For instance, we assumed a similarity threshold of one hundred percent. This means each target rule had to match the standard rule exactly. Using a lower threshold yields more rules. In most cases, moving below eighty-five percent increased execution time exponentially. As the threshold becomes less strict, the number of dynamic (D) positions in the cluster id increases. Remember that a ‘D’ in the cluster Id. means that position could be a 0 or 1, so each ‘D’ changes the range of similarity by 2^n, where n is the number of ‘D’s present.

The results of FindSimilarRules[] heavily depends on the similarity measure. A comparison by output measures exact similarity. However, a different similarity measure may be used to capture image translation as similar as well. Currently, FindSimilarRules[] will not account for an image translation, rotation, or dilation from the standard. This more sophisticated similarity measure could prove useful when comparing between rule spaces of differing sizes. A radius 1 Sierpinski triangle is of a smaller size compared to a radius 2 Sierpinski triangle. Finding similarity across radii may offer more insight to global behavior of these clusters.

8 Conclusion

Many readers may ask, ‘What are practical applications for finding similar cellular automata rules?’ Let us answer that question with an illustration. Let us assume that there exists a box filled with 100 colored cubes of uniform size with colors of red, blue, green, and white. Now, the contents of the box are dumped onto the floor. The objective is to sort the pile of cubes by color. The process may take a minute or two, but easily enough, sorting a small number of 100 cubes would not be a considerable challenge. Now, let us assume that there exists a box of over four billion cubes. The existing colors now blend into millions of combinations of red, blue, green, and white. The objective remains the same: sort the pile of cubes by color. The task is much more daunting. Where cellular automata rule spaces were once defined as randomly distributed, Obando proved that a partially ordered system gives rise to clustering in the space [5].

Using this system, we have defined an algorithm to be a framework for identifying and collecting similar rules in a cellular automata rule spaces using a similarity measure that compares direct output. Using this framework, one can simply replace the similarity measure to observe completely different results similarly to the comparison by step method. Over the course of this thesis, we have asked and answered three questions. Clusters are defined by a cluster identification string known as a cluster Id. These cluster Ids. are unique for every cluster. Table 1 showed us that two separate rules can yield the same cluster Id. This means the two rules in question must belong to the same cluster. Because the similarity measure selected in this thesis compared exact outputs, rules belonging to the same cluster must also be globally similar. In other words, rules belonging to a specific cluster all produce the same general behavior when applying our similarity measure. Now, local rule behavior may reveal global behavior in the cluster.

Code Appendix

```
(*Function that sets up the pre-computed standard rule and calls the recursive function SimilarHelper.*)

Similar[rule_, threshold_, init_: {{1}, 0}, k_: 2, radius_: 1,   steps_: 30] :=

 Module[{VR = {rule}, CR = {rule}, timer = AbsoluteTime[],    fullRule = {}},

  ParallelSow[rule, v];

  ParallelSow[rule, s];

  fullRule = CellularAutomaton[{rule, k, radius}, init, {steps, All}];

  SimilarHelper[fullRule, CR, VR, threshold, init, k, radius, steps, 1];

  Print["Total Time: " <> ToString[AbsoluteTime[] - timer] <> " seconds"]   ]


(*Function that recursively calls itself until Hamming distance 3. When detecting a target rule that is similar the standard rule, the
value is stored away. Returns a list of lists → {Null, {{List of visited rules}, {List of similar rules}}}*)

SimilarHelper[rule_, currentRules_, vRules_, threshold_, init_, k_,   radius_, steps_, hamDist_] :=

 Module[
{visitedRules = vRules, rulesToVisit = {}, currentLevel = {}, timer = {}, stepSize = steps, preComputedRule =
DynamicBits[rule, radius]},

  Print["Hamming Distance: " <> ToString[hamDist]];
  Print["\tCollecting Neighbors..."];

  timer = Timing[Complement[ DeleteDuplicates[ Flatten[Map[Flatten[Rest[NeighborRules[#, radius]]] &, currentRules]]],
visitedRules]];

  currentLevel = timer[[2]];

  Print["\tThere are " <> ToString[Length[currentLevel]] <> " rules in this level. (" <> ToString[timer[[1]]] <> ") seconds."];

  Print["\tTesting Neighbors..."];

  SetSharedVariable[visitedRules];

  SetSharedVariable[rulesToVisit];

  timer = Timing[Parallelize[

    Map[ (ParallelSow[#, v]; AppendTo[visitedRules, #];

      If[ AvgCompareRulesByOutput[rule, #, init, stepSize, k, radius] >=  threshold,

       ParallelSow[#, s];

       AppendTo[rulesToVisit, #]  ];) &,

    currentLevel], Method -> "FinestGrained"]];

Print["\tThere are " <> ToString[Length[rulesToVisit]] <> " rules to visit. (" <> ToString[timer[[1]]] <> ") seconds."];

If[Length[rulesToVisit] > 0,

 If[hamDist < 3,
```

```
visitedRules =
  SimilarHelper[rule, rulesToVisit, visitedRules, threshold, init,
   k, radius, stepSize, hamDist + 1]],
 Map[Print["Leaf Rules:" <> ToString[#]] &, currentRules]] ]


(*Compares the target to the standard by the output of each rule. Can accept an integer standard or a pre-computed version of the standard.*)

NewCompareRulesByOutput[standard_, target_, init_, t_, k_, r_] :=
 Module[  {standardList = {},   targetList = CellularAutomaton[{target, k, r}, init, {t, All}],   returnList = {}},
  standardList = Switch[standard,
   _Integer, CellularAutomaton[{standard, k, r}, init, {t, All}],
   _List, standard,
   _, {}];
  Return[
  Mean[Table[
N[1 - Apply[Plus,  Abs[standardList[[currentStep]] -   targetList[[currentStep]]]]/ Length[Last[standardList]],  4],
{currentStep, 1, t}]]]]


(*Returns the average (similarity) of the of the target to the standard.*)

AvgCompareRulesByOutput[standard_, target_, init_, t_, k_, r_] :=
 Return[NewCompareRulesByOutput[standard, target, init, t, k, r]]


(*Function that determines if the target is similar to the standard based on dynamic bits.*)

DynamicBitsQ[standard_, target_, radius_] :=
 Module[{standardBits = "", targetBits = ""},
 standardBits = Switch[standard,
   _Integer, DynamicBits[standard, radius],
   _List && Length[standard] == 2, standard,
   _, {}];
 targetBits = Switch[target,
   _Integer, DynamicBits[target, radius],
   _List && Length[target] == 2, target,
```

```
  _, {}];
 Return[ContainsAll[targetBits[[1]], standardBits[[1]]] && ContainsAll[targetBits[[2]], standardBits[[2]]]]]


(*Function that returns a list of rules based of the similarity of their dynamic bit location.*)
FindSimilarRules[standard_, radius_: 1, threshold_: 0.99] :=
 Map[FromDigits[#, 2] &, CreateRules[IntegerDigits[standard, 2, 2^(2*radius + 1)], ClusterToRule[standard, radius, threshold]]]

SetAttributes[FindSimilarRules, Listable];

(*Function that takes in a cluster id or rule, then returns a binary form of that cluster id.*)
FromClusterID[standard_, radius_, threshold_: 0.98] :=
 Module[{temp = {}, id = {}},
  id = Switch[standard,
    _Integer, ClusterID[standard, radius, threshold],
    _List, standard,
    _, {}];
  id[[1]] = id[[1]] /. {"P" -> 1} /. {"N" -> 0} /. {"D" -> 1};
  id[[2]] = id[[2]] /. {"P" -> 0} /. {"N" -> 1} /. {"D" -> 0};
  Table[
   AppendTo[temp, Take[id[[2]], {pos, pos + (2^radius - 1)}]];
   AppendTo[temp, Take[id[[1]], {pos, pos + (2^radius - 1)}]],
   {pos, 1, 2^(2 radius + 1)/2, 2^radius}];
  Return[Flatten[temp]]]

(*Function that converts a cluster id or rule into its expanded cluster id form.*)
ClusterToRule[standard_, radius_, threshold_: 0.98] :=
 Module[{temp = {}, id = {}},
  id = Switch[standard,
    _Integer, ClusterID[standard, radius, threshold],
    _List && Length[standard] == 2, standard,
    _, {}];
```

```
Table[
 AppendTo[temp, Take[id[[2]], {pos, pos + (2^radius - 1)}]];
 AppendTo[temp, Take[id[[1]], {pos, pos + (2^radius - 1)}]],
 {pos, 1, 2^(2 radius + 1)/2, 2^radius}];
Return[Flatten[temp]]]
```

```
(*Function that takes in a binary number and a cluster id as a mask to produce all the possible numbers according to the mask.*)
CreateRules[in_, mask_] :=
 Module[{p = Position[mask, "D"]},
  ReplacePart[in, Rule @@@ Transpose[{p, #}]] & /@ Tuples[{0, 1}, Length@p]]
```

```
(*Function that returns the dynamic bit locations as a list of lists -> {p0, p1}.*)
DynamicBits[standard_, radius_] :=
 Module[{standDigits = Map[IntegerDigits[#, 2, 2^(2 * radius + 1)/2] &, BreakRule[standard, radius]], p0 = {}, p1 = {}},
  Table[If[standDigits[[1]][[pos]] == 1, AppendTo[p0, pos]],
{pos , 1, Length[standDigits[[1]]]}];
  Table[If[standDigits[[2]][[pos]] == 0, AppendTo[p1, pos]],
{pos , 1, Length[standDigits[[2]]]}];
  Return[{p0, p1}]]
```

```
(*Helper function to PrintSimilarBits. Counts all p0 bits that are 1s and all p1 bits that are 0s.*)
PrintSimilarBitsHelper[standard_, targets_, radius_: 1] :=
 Module[
  {p0 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
   p1 = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
   standDigits = Map[IntegerDigits[#, 2, 2^(2 * radius + 1)/2] &, BreakRule[standard, radius]],
   targetBits = Map[IntegerDigits[#, 2, 2^(2 * radius + 1)/2] &, Map[BreakRule[#, radius] &, targets]]
   },
 Map[
  If[DynamicBitsQ[standDigits, #, radius],
```

```
(Do[If[#[[1]][[pos]] == 1, p0[[pos]] = p0[[pos]] + 1], {pos , 1,
   Length[#[[1]]]}];
  Do[If[#[[2]][[pos]] == 0, p1[[pos]] = p1[[pos]] + 1], {pos , 1,
   Length[#[[2]]]}];)] &,
 targetBits];
Return[{p0, p1}]]
```

```
(*Function that prints out the dynamic bit locations for the standard and neighboring rules collectively.*)
PrintSimilarBits[standard_, targets_, radius_: 1, threshold_: 0.75] :=
 Module[{bits = PrintSimilarBitsHelper[standard, targets, radius],   p0 = {}, p1 = {}, max = 0},
  max = Max[bits];
   Do[If[bits[[1]][[pos]]/max >= threshold,
    AppendTo[p0, pos]], {pos , 1, Length[bits[[1]]]}];
  Do[If[bits[[2]][[pos]]/max >= threshold, AppendTo[p1, pos]], {pos ,
    1, Length[bits[[2]]]}];
  Return["Standard Key Bits: " <> ToString[DynamicBits[standard, radius]] <> "\nTargets Combined Key Bits: " <>
ToString[{p0, p1}]]]
```

```
(*Function that gets the position of the single different bit between standard and target.
It is assumed that the standard and target have only one bit of difference.
Returns a list →{Position of bit, Integer representing corresponding primitive list (1 for p0 and 2 for p1)}*)
GetDifferentBit[standard_, target_, radius_] :=
 Module[{   standPrims = DynamicBits[standard, radius], targetPrims = DynamicBits[target, radius], complPrims = {}},
  AppendTo[complPrims, Complement[standPrims[[1]], targetPrims[[1]]]];
  AppendTo[complPrims, Complement[targetPrims[[1]], standPrims[[1]]]];
  If[Length[Flatten[Select[complPrims, UnsameQ[#, {}] &]]] > 0,
   Return[Append[Flatten[Select[complPrims, UnsameQ[#, {}] &]], 1]]];
  AppendTo[complPrims, Complement[standPrims[[2]], targetPrims[[2]]]];
  AppendTo[complPrims, Complement[targetPrims[[2]], standPrims[[2]]]];
  Return[Append[Flatten[Select[complPrims, UnsameQ[#, {}] &]], 2]]  ]
```

```
(*Function that displays the significant bits for the given rule (standard).
```

P → Positively effective in the formation of the cluster. This means that this bit is required to create the rule's image.

D →Dynamic in the formation of the cluster. This means that this bit could be on or off. Neither will affect the rule's image.

N → Negatively effective in the formation of the cluster. This means that flipping this bit makes the image deviate from its original cluster.*)

```
ClusterID[standard_, radius_: 2, threshold_: 0.99 , steps_: 100] :=
 Module[  {p0 = PadRight[{}, 2^(2 * radius + 1)/2],    p1 = PadRight[{}, 2^(2 * radius + 1)/2],
   standDigits =
    Map[IntegerDigits[#, 2, 2^(2 * radius + 1)/2] &,  BreakRule[standard, radius]],
   temp = {},
 intersection = DynamicBits[standard, radius]},
  Map[
If[AvgCompareRulesByOutput[standard, #, {{1}, 0}, steps, 2, radius] >= threshold,

    temp = GetDifferentBit[standard, #, radius];
    intersection[[1]] =  Intersection[intersection[[1]], DynamicBits[#, radius][[1]]];
    intersection[[2]] =  Intersection[intersection[[2]], DynamicBits[#, radius][[2]]];
    If[temp[[2]] == 1, p0[[temp[[1]]]] = "D", p1[[temp[[1]]]] = "D"],

    temp = GetDifferentBit[standard, #, radius];
    If[temp[[2]] == 1, p0[[temp[[1]]]] = "N",
     p1[[temp[[1]]]] = "N"]] &,
  Rest[NeighborRules[standard, radius]], {2}];

 temp = DeleteDuplicates[ Flatten[Map[Position[intersection, #] &, intersection, {2}], 2]];
 Map[
  If[#[[1]] == 1,
    p0[[intersection[[1]][[#[[2]]]]]] = "P",
    p1[[intersection[[2]][[#[[2]]]]]] = "P"] &, temp];
 Return[(*"p0 = "<>ToString[p0]<>"\np1 = "<>ToString[p1]*){p0, p1}]  ]
```

References

[1] Bar-Yam, Y. (2002). "General Features of Complex Systems." Encyclopedia of Life Support Systems. EOLSS UNESCO Publishers.

[2] Huffman, J. (2014). "An Exploration of Complex Systems in 16 Dimensions." Columbus State University.

[3] Imre, A., Csaba, G., Ji, L., Orlov, A., Bernstein, G. H., and Porod, W. (2006). "Majority Logic Gate for Magnetic Quantum-Dot Cellular Automata." Science Magazine. Vol. 311 Pg. 205-208.

[4] Langston, C. G. (1990), "Computation at the edge of chaos: Phase transitions and emergent computation." Emergent Computation. Pg. 12. North-Holland, Amsterdam.

[5] Obando, R. (2015). "Partitioning of Cellular Automata Rule Spaces." Complex Systems, 24. Complex Systems Publications, Inc.

[6] Parker, J. R. (2010), "Algorithms for Image Processing and Computer Vision." Wiley Publishing, Inc. Second Edition.

[7] Sipper, M. (1994). "Non-Uniform Cellular Automata: Evolution in Rule Space and Formation of Complex Structures." Artificial Life IV. The MIT Press. Pg. 394-399.

[8] Wolfram, S. (1985). "Twenty Problems in the Theory of Cellular Automata." Physica Scripta. Nobel Symposium. vol. T9. Pg. 170 – 183.

[9] Wolfram, S. (1994). "Cellular Automata and Complexity: Collected Paper." Addison-Wesely.

958S15 9748
08/16/16 31180
EHGroup