

Optimum Cellular Automata Configurations for Encryption

A Master's Thesis by

Daniel Nichols

Fulfilling a partial requirement for the Degree of Master of Science

Engineering Technology

Middle Tennessee State University

August 2015

Committee Members:

Dr. Karim Salman, Chair

Dr. Walter Boles

Dr. Saleh Sbenaty

ABSTRACT

Elementary cellular automata (ECA) is using Boolean logic functions to generate populations of ones and zeros. Population takes place temporally; each row of bits are populated based on the data from the previous row. The rules followed for generation of the next row are the functions mentioned above. Some rules produce very linear and predictable sets of binary data whereas some of them are chaotic in nature, like rule 30, and can be used to produce pseudo-random populations. This state by state calculation is ideal for cryptography in that it relies on an initial state (password or key) and can be streamed or generated as more data is transferred; the patterns are also largely unknown and are not predictable or linear in some cases. Currently, research shows many promising configurations of population generation that can be used for securing digital data or communications. Unfortunately the chaotic rules tend to repeat themselves after a given amount of temporal generation. This cyclic behavior is non-linear and near impossible to detect without generating the ECA. This research is aimed at finding the relationships between row length and cycle length in an effort to make a given key more effective by lengthening or eliminating this repetitious behavior.

TABLE OF CONTENTS

1	Research Questions	vi
2	List of Tables	vii
3	List of Figures	viii
4	Background and Introduction	1
4.1	History	1
4.2	Introduction to Encryption	1
4.3	Chaotic CA Basics	3
5	Topic Importance	8
6	Tools and Methodology	9
6.1	Programming Methods and Failures	9
6.1.1	Introduction	9
6.1.2	Languages	9
6.1.3	User Interface	10
6.1.4	Form and Program Architecture	10
6.1.5	Cellular Automata Calculation	11
6.1.6	Data Storage and Manipulation	14
6.1.7	Dynamics of Cellular Automata and Cycle Detection	15
6.1.8	Parallelism and Cellular Automata	18

6.1.9	Parallel - Micro	22
6.1.10	Recommended Programming Approach	22
6.2	Dynamics of Cellular Automata	26
6.2.1	Reversibility of Cellular Automata	27
6.2.2	Garden of Eden States and Importance	29
6.3	Adjacencies and Multi-dimensional Configurations	37
6.3.1	Periodicity	38
6.3.2	Neighborhoods	39
6.3.3	Extra Dimensional CA	41
6.4	Multi-Rule Configurations	43
6.4.1	Spatial Configuration	43
6.4.2	Temporal Configuration	44
6.4.3	Spatial-temporal Configuration	44
7	Results	49
7.1	One Dimensional	49
7.1.1	Single Rule	49
7.1.2	Two Rules	50
7.2	Two Dimensional	52

7.2.1	Single Rule	52
7.2.2	Two Rules	53
7.3	Three Dimensional	55
7.3.1	Single Rule	55
7.4	Discussion	56
	Publication Reference	57
8	Appendix	58
8.1	Two Rule 1D Charts	59
8.1.1	Classical Temporal	60
8.1.2	Classical Spatial	65
8.1.3	Alternate Temporal	70
8.1.4	Alternate Spatial	75

1 RESEARCH QUESTIONS

What programming practices lead to rapid cellular automata calculation?

What are the dynamics of chaotic elementary cellular automata?

Which rules and bit lengths are most suitable for pseudo-random number generation?

Which adjacency number and configuration is ideal for pseudo-random number generation?

2 LIST OF TABLES

Table 1 Rule group 1 _____	6
Table 2 Rule group 2 _____	6
Table 3 Rule group 3 _____	7
Table 4 Rule group 4 and 5 _____	7
Table 5: Number of GOEs in a given length for $k=1$ to 32. _____	34
Table 6: The top GOE patterns found in seed lengths 1-27. _____	36

3 LIST OF FIGURES

Figure 1: The encryption and decryption process of a PRNG stream cypher. _____	2
Figure 2: Rule 30 visualized. _____	3
Figure 3: The distribution and collection of seeds and cycles. _____	15
Figure 4: A larger overview of distributed CA computing. _____	16
Figure 5: The tortoise and hare algorithm visualized. _____	19
Figure 6: The CUDA application algorithm visualized. _____	21
Figure 7: Seed length $k = 3$; transients and cycle. _____	26
Figure 8: The reverse algorithm visualized with colored steps. _____	27
Figure 9: The single state failures of rule 30 and even seed lengths _____	29
Figure 10: A full state map of seed length six. _____	30
Figure 11: A full state map of seed length five. _____	31
Figure 12: A full state map of seed length seven. _____	32
Figure 13: A full state map of seed length eight. _____	33
Figure 14: Three graphs visualizing Table 5. _____	34
Figure 15: Dimensional adjacencies visualized. 1D to 4D. _____	37
Figure 16: The periodic nature of a seed length of 6; indices 0-5. _____	38
Figure 17: The differences in concatenation of each row created by CA. _____	38
Figure 18: The toroidal nature of a two dimensional CA. _____	39
Figure 19: The Von Neumann adjacencies (left) versus the Moore adjacencies. _____	39
Figure 20: A graph of randomness versus the adjacency configuration used. _____	40

Figure 21: Dimensional differences of randomness. _____	42
Figure 22: The special rule changes visualized. _____	43
Figure 23: Temporal rule changes visualized. _____	44
Figure 24: Arrow configuration: two-step process visualized. _____	46
Figure 25: Moore configuration: two-step process visualized. _____	48
Figure 26: Single rule, one dimensional randomness. _____	49
Figure 27: A small sample of Spatial-Alternate two rule tests _____	50
Figure 28: A small sample of Classical-Temporal two rule tests _____	51
Figure 29: Randomness graphed for a two dimensional CA. _____	52
Figure 30: Randomness graphed for 2D CA using the successful 1D configurations. __	53
Figure 31: Three dimensional CA randomness graph. _____	55

4 BACKGROUND AND INTRODUCTION

4.1 HISTORY

Research into elementary cellular automata has been going on publically since the 1980's [2]. Only a few realized its application in computer encryption and steady research has been completed over the years since [1]. Recently there has been a surge in encryption methods due to many common encryption methods that have been broken. One person considered to be the father of cellular automata is Stephen Wolfram. Wolfram studied and published many papers and patents on the subject; his findings and conjectures are being used as a baseline for this research. An attempt to reproduce his data will be made whilst using advances in modern technology to expand upon the limit he reached.

4.2 INTRODUCTION TO ENCRYPTION

Encryption is used to obfuscate a message of any kind into a form in which an eavesdropper cannot interpret what the original message contained. In the case of digital communications on a basic level all messages are just a collection of binary ones and zeros whether they are pictures, text, video or files.

Encryption relies on an algorithm to transform the data using a private key, password, or seed. This key is propagated in such a way that it complexes the data and causes it to be incomprehensible. This is usually done with a linear, reversible Boolean operator

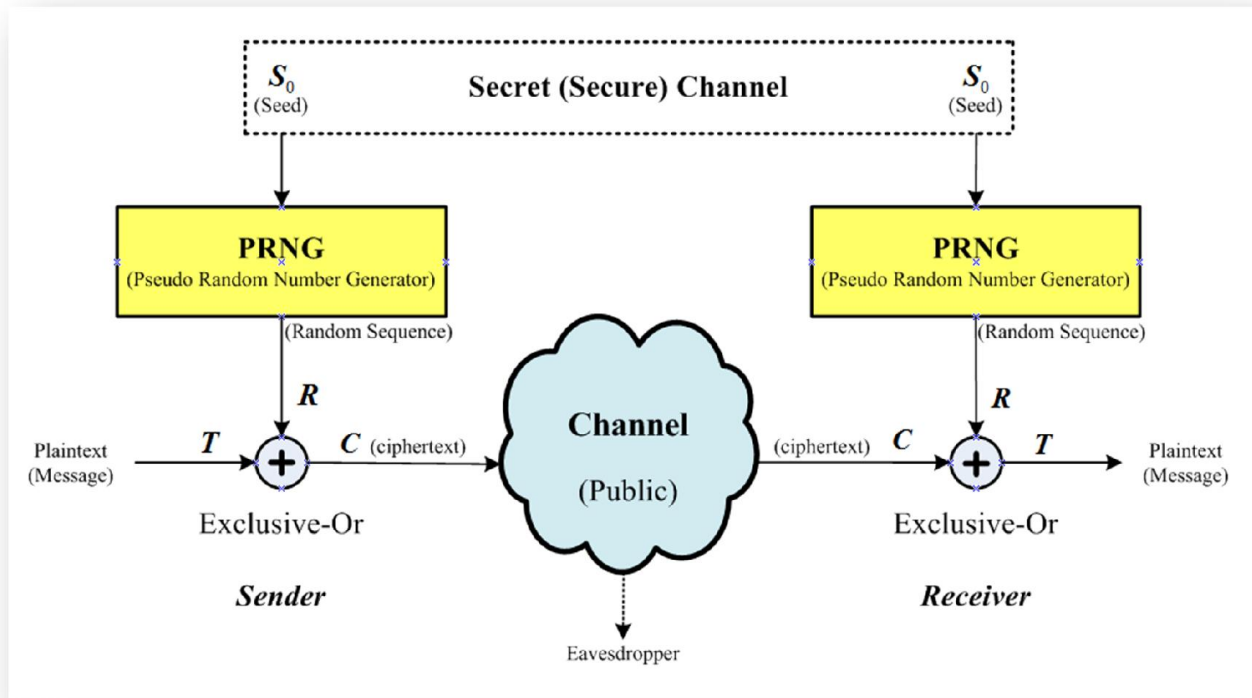


Figure 1: The encryption and decryption process of a PRNG stream cipher.

such as the exclusive or. The data is then transferred via an unsecure method and then simply decrypted by the party with the proper private key.

There are two basic methods for the data to be encrypted, the block cipher, where the set of data is finite, and the stream cipher, in which the data is constantly flowing.

Cellular automata is perfect for the stream cipher as it is generated from the previous state and can continue indefinitely.

The methods of encryption normally used work on the data itself, transforming the data by using an algorithm that changes the data in a way that can be reversed using a key.

This method has flaws in that the formulas used in this process can be reverse engineered to produce the encrypted data. There are many ways to defeat such a system but it is still prevalent today.

4.3 CHAOTIC CA BASICS

Cellular automata is different in that only its very next state can be predicted by mathematics. The rule-space used in this research is limited to three bits; currently the next state is influenced by the previous state of that cell and its neighbors to the left and right. These three bits are used in a Boolean function to determine the next state. To the

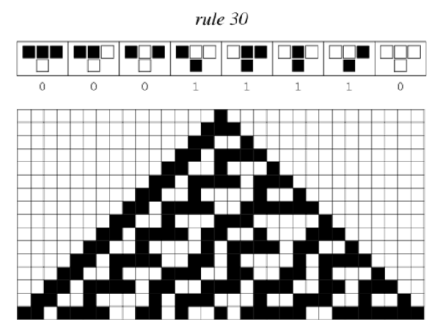


Figure 2: Rule 30 visualized.

right is an example (ones are black and zeros are white) of rule 30. At the top of the chart the integer thirty is represented in binary "00011110". This is the lower square in the "T" arrangement above each one or zero. The three squares above this represent its position, counting from zero on the right "000" to seven on the left "111". Rule 30 is a nice example because a simple state, like all zeros with one in the middle escalates to chaos. Other rules are not so useful and in fact, research has shown the two major rules for chaotic cellular automata are rule 30 and 45 [3].

This initial state of the generation is often called the seed but can represent the private key mentioned above. If two parties have the same key they can generate over ten thousand more random bits stemming from the original seed. This gives the user the ability to encrypt a massive amount of data with a relatively small seed.

Rule 30 is an extremely simplified implementation of an encryption routine. Each time step can render a limitless amount of possibilities given a specific rule set using adjacency rules, arrangement rules, and even multiple rules.

Previous research has been done to verify specific types of rules in the 3 bit rule space. Wolfram classified all rules into four types and outlined their properties [1].

Using this data more research has been done to clarify which rules would be useful whilst calculating chaotic CA for pseudo-random number generation [3]. It was decided that there are sixteen chaotic rules that are viable for pseudorandom number generation. These rules are:

30,45,60,75,86,89,90,101,102,105,135,149,150,153,165,195 as stated on page 41 of [3].

These rules can be broken into equivalence groups using concepts found in [4] that outline the relationships between inversion and reflection of rule bits.

1. Non-linear group 30 consisting of the four rules: 30, 86, 135 and 149
2. Non-linear group 45 consisting of the four rules: 45, 75, 89 and 101
3. Linear group 60 consisting of the four rules: 60, 102, 153, and 195
4. Linear group 90 consisting of the two rules: 90, 165
5. Linear group 105 consisting of the two rules: 105, 150

Table 1 Rule group 1





Rule 30	Rule 86	Rule 135	Rule 149
			

Table 2 Rule group 2





Rule 45	Rule 75	Rule 89	Rule 101
			

Table 3 Rule group 3









Rule 60	Rule 102	Rule 153	Rule 195
			

Table 4 Rule group 4 and 5

Rule 90	Rule 165	Rule 105	Rule 150
			

5 TOPIC IMPORTANCE

Most modern encryption methods rely on static mathematical principles to obfuscate the data in a way that is irreversible. Unfortunately these methods are widely known and are constantly being attacked. Since the methods are purely mathematical they can be reverse engineered and spoofed.

Cellular automata is not formulaic in that it relies on a previous state to make a calculation. It works on itself and is therefore unpredictable mathematically thus far.

This property makes it ideal for encryption in that it has no basis mathematically to work backwards from. There are of course linear rules that are very predictable mathematically but the chaotic rules are truly perplexing.

The purpose of this research is to systematically try to develop ways to understand the properties of these systems so that it may be used to its full potential and also identifying any possible characteristics that may cause the system to fail.

6 TOOLS AND METHODOLOGY

Many methods were considered and most were tested to different conclusions as for their usefulness and efficiency. The first section of this article will discuss the various permutations of algorithms used to test hypotheses.

6.1 PROGRAMMING METHODS AND FAILURES

6.1.1 Introduction

Programming languages used in this research included C# and C++. MySQL was also used to log the data collected. A statistics software called diehard was used to determine the possible random p-values. This was used to scale the randomness of a given CA and seed.

6.1.2 Languages

C#'s high level was a great test tool as a proof of concept whilst testing CA.

Unfortunately it seems to execute CA rather slowly in comparison to C++; yet it provides an excellent framework for Windows forms.

C++ allows for the closest high level path to x86 (assembly), and it proved to be much faster than C# for direct CA calculations and memory management. C# dynamically allocates memory, which is likely the reason for the detrimental effects in speed.

It is possible that speed is not a major concern in one's application, but clarity. In that case, scripting languages such as Python or Lua would be ideal; they have the advantage of no compilation, increased portability, and ease of use in development.

6.1.3 User Interface

Due to the widespread use of Windows computers, the simplest method for interacting with the underlying C++ programs was through Windows forms and the .NET framework.

Keeping in mind the goal was testing, the Windows form method has drawbacks when wanting to migrate to other systems such as Linux or Mac OS X, where the UI can be different depending on usage.

6.1.4 Form and Program Architecture

The first method of transporting data from C# to C++ was running the C++ compiled exe via a C# form with a text file argument specifying settings for the particular CA. This method is necessary due to the fact the CA seeds can be longer than the command line parameters of windows processes. This method can be quite frustrating to debug as both code bases must be updated simultaneously; also to add a function or process takes valuable programming time to generate and receive output for settings passage.

The second, possibly slower, but much more adaptable method is compiling C++ into a Dynamic Link Library, or DLL, then importing and executing the external methods via C#. This requires heavy data manipulation prior to execution of the C++ method due to the data storage differences between the languages.

The last method arrived upon was concocted using the above two methods. Using a single project the DLL is created then used by a mother C++ program that takes in a text based argument. In this way the DLL still may be used externally and text configuration files are still logged and documented.

OOP, or Object Oriented Programming tends to exchange computational efficiency for clarity, yet it provides an excellent framework for testing CA in many configurations with little repetitious code. The small performance decrease was acceptable for the benefit of customization of a CA calculation. This also does not complement DLLs by any means but can be adapted using a wrapper function. Since CA can be executed once to receive a particular output the overhead involved in declaring classes is minimal.

6.1.5 Cellular Automata Calculation

When imposing rules to propagate the CA there are preferred methods for doing calculation that increase speed and minimize code length. Previously code had been written where there was a fixed C++ function for every rule in binary as well as a char ('1' and '0'). While this code executes fast it requires an extreme excess of code as well

as a very large margin for error as each rule must be converted into conventional logic gates.

Fortunately rules are represented in binary and their logic can easily be found in storage. For example 30, represented as an unsigned single byte, is stored as 00011110. The position of each bit can be represented by a three bit binary number and conversely can be used to look up whether the CA is to produce a 1 or a 0 in the next time step.

It is possible to store a three dimensional array of booleans with two array elements for each dimension. This way three bits can be passed to receive the next state for a particular rule. This evolved into a four dimensional array that stored every rule so any Elementary Cellular Automata (ECA) rule can be used (3 bits, 256 rules).

Below is an example of such an implementation, where `getBit(i,j)` returns the “j”th bit of the binary number `i`.

```
bool getBit(int from, int pos) { return (from >> pos) & 1; }

bool minterms[256][2][2][2];
for (int i = 0; i < 256; i++)
{
    minterms[i][0][0][0] = getBit(i,0);
    minterms[i][0][0][1] = getBit(i,1);
    minterms[i][0][1][0] = getBit(i,2);
    minterms[i][0][1][1] = getBit(i,3);
    minterms[i][1][0][0] = getBit(i,4);
    minterms[i][1][0][1] = getBit(i,5);
    minterms[i][1][1][0] = getBit(i,6);
    minterms[i][1][1][1] = getBit(i,7);
}
```

The lookup time for this type of calculation is constant as it progresses to the exact element needed to do calculation. This saves valuable calculation time and has monumental effects on the overall time as this calculation is done millions, perhaps billions of iterations before execution completion.

6.1.6 Data Storage and Manipulation

Using the methods above it is still possible to achieve moderately slow results by streaming data to a file. This can be said of most programs that do file system manipulation but is especially important in CA; large buffer reads and writes save computation time. There are many reasons for this, all of which contribute greatly to its efficiency. These include sequential reads/writes to drive, eliminating unnecessary os/file system communication and overhead, and benefiting from memory speeds in CA calculation.

This method of file IO is *essential* to CA calculation. If large file storage is not an option, a large buffer should be used to communicate the data at specific times in execution, such as dumping it into a SQL database in memory.

6.1.7 Dynamics of Cellular Automata and Cycle Detection

6.1.7.1 Introduction and first method

Cycle detection is important in the testing of the cryptographical usefulness of CA. There are many algorithms and data structures that can be used to find a cycle in a given CA. Logically the first course of action is to record every state the CA produces and for each new time step look back through the previously stored data and determine if the state is a duplicate.

An algorithm was contrived to identify GOE states for a given seed length by maintaining a database of every seed possibility and running them, one by one, until they cycle. After every state generation, or time step, the next step was deleted from the database. Therefore, after running every seed the database only contains GOE states. This was exhaustive and computer resource intense; we were unable to produce reliable results past bit length eighteen. This method is also not scalable because as every seed length increases the possibilities exponentially grow.

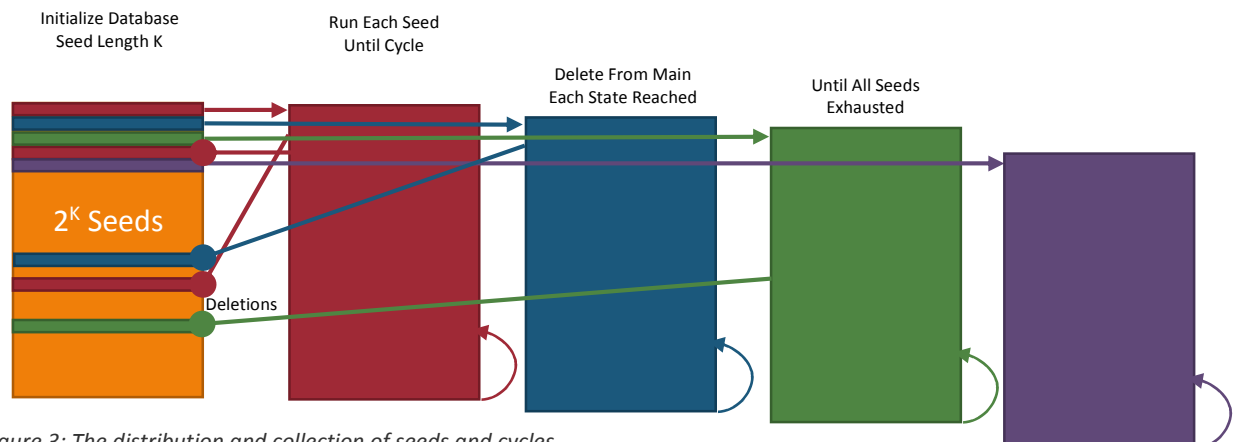


Figure 3: The distribution and collection of seeds and cycles.

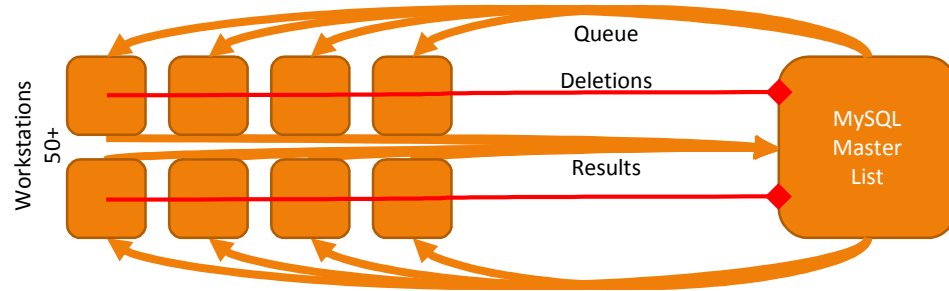


Figure 4: A larger overview of distributed CA computing.

This was implemented using a MySQL database server and idle workstations. The SQL database was populated using a program with all possibilities of a given seed length. Then, a service was distributed across all workstations to pull from that database and run the ECA until it cycled. The workstations would also delete states that are generated from every row out of the main database because these states are accounted for. When the ECA has cycled the service reports back to the server and stores the following information: seed, transient length, cycle length, point of entry and a cycle id.

Although this is the most thorough and straightforward approach it can be quite resource intensive. For example, a cycle cannot be longer than memory can hold and seeing as how CA is a power growth in cycle length, given seed length, this method is completely un-expandable.

6.1.7.2 Memory Intensive Methods

This method was first tested with a linked list as storage which proved only a frustration and timely lesson in efficiency. Then a tree implementation was used and also proved inefficient due to its unbalanced nature. Finally a treap was programmed which provided functionality for a balanced binary tree and quite efficient lookups. This method stores around 14GB of data for a cycle length of 50 million. This is the only algorithm in our research that proved the ability to detect transient length (path to cycle) and an exact entry point.

6.1.7.3 Computationally Limited (Constant Memory) Methods

There is another algorithm used to detect cycles in CA and quite a few other pseudo-random number generators or PRNGs. This is dubbed the Tortoise and the Hare algorithm due to the nature of its process. It finds a cycle using a fixed memory amount and just requires increased execution time to detect longer cycles. There is a small improvement using Brent's Algorithm and this was the eventual method used in a CUDA (Compute Unified Device Architecture, NVIDIA's Parallel GPU API) implementation to detect cycles for given seeds in parallel. The only drawback to this method is it provides only the cycle length and an arbitrary state in the cycle; transient lengths or entry points cannot be determined without further calculation.

6.1.8 Parallelism and Cellular Automata

When looking at cellular automata, it can be simple to argue that it would benefit from parallelism (such as threading, cluster computing, NVIDIA CUDA, AMD Mantle, etc.) but like any other problem, it will raise the complexity and possibly blur results. This conversation breaks down into serial vs. parallel rather quickly. Basic CA is serial. Parallel CA can be looked at in two ways - macro and micro.

6.1.8.1 Serial

The basic process of CA is serial - results of the last step are fed into the next one. As an example, a one dimensional seed of length 8 that wraps at its ends has a three-bit rule (0 - 255) applied to each of the seed's eight members, using the left and right bits as its adjacencies to use for the three bits in the rule's lookup table. Each time step of this process generates a global change to the seed. The third time step depends on the second, which is the essence of the serial process in CA.

6.1.8.2 Parallel - Macro

On a macro scale, multiple CAs can be tested at once, i.e. create several threads with a separate process for calculating a CA. If the computing hardware has 8 real (idle) threads, there is effectively 8 times the processing power of a single machine by setting a CA process to run on each thread. Bear in mind that work by the operating system must take place somewhere and that some threads which may appear idle are probably being used every so often for various processes. For this reason, testing and calculating CA via clustering can be very effective even with a non-parallel CA application.

Using this technique code was generated for a CUDA application to parallelize the cycle testing of CA. The algorithm used is introduced in 6.1.7.3, the tortoise and hare algorithm.

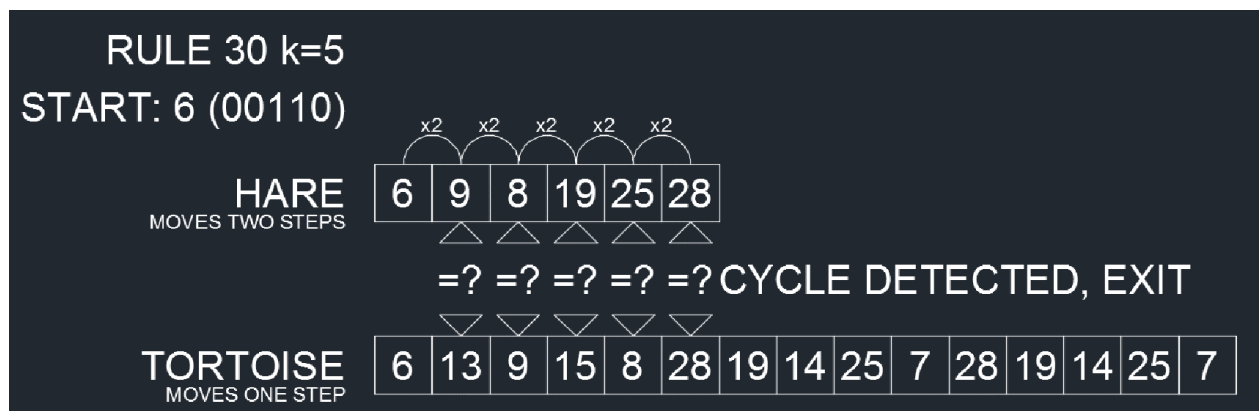


Figure 5: The tortoise and hare algorithm visualized.

The above illustration shows the implementation of the tortoise and hare algorithm implemented given a particular rule, k and starting state. The CUDA application used this algorithm along with memory manipulation to run a large quantity of these cycle detection runs concurrently.

Using TITAN GPU equipment I have been able to parallel process cycle testing for any given seed (bit) length. In our GPU we have 15 multiprocessors with 64 warps each; and each warp can support 32 threads. This means the GPU can handle around 7680 threads at any one moment. Given that CA is binary and expands as such 8192 threads are launched at once for the GPU to process.

In a traditional computer a maximum of 8 threads can be run at any time. The GPU provides 1000x faster results than using a CPU calculation method.

In the figure below only four states are shown but are actually 8192 in length. Each set of 8192 are sent to the device to be calculated then appended to the output csv file.

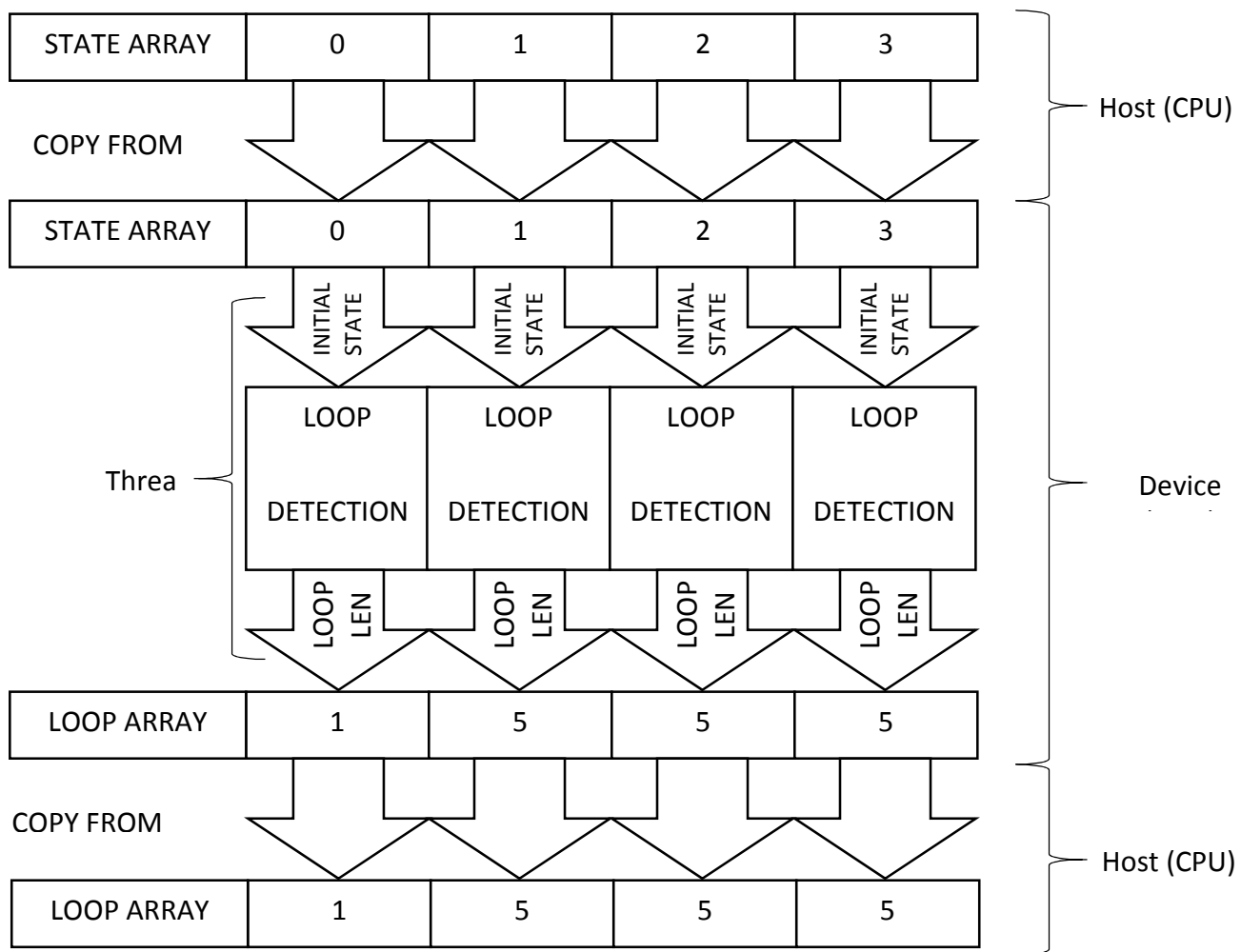


Figure 6: The CUDA application algorithm visualized.

6.1.9 Parallel - Micro

On the micro scale, it is necessary to look at the internals of the CA process. A single seed has k number of members. As a result, the number of calculations each time step is a multiple of k . In situations where those calculations are large, i.e. it is using a significant amount of time per calculation, the possibility of internally threading the CA process could be useful. It is important to realize that threading can present challenges not thought of in the basics of serial CA. Operations must be synchronized after a predetermined point, and if the thread synchronization takes longer than traditional serial CA, then it might be a better (cleaner, easier) path to simply not parallelize that part of the process.

6.1.10 Recommended Programming Approach

6.1.10.1 User Interface

An IDE was used that supports creating a form and that is easily manipulated; where events and properties are automatically generated. The underlying code is ideally non OS dependent leaving room for many options for the interface.

6.1.10.1.1 Windows

Visual C# with Visual Studio form was used to import and use the low level DLL for its calculation.

The C++ DLL driver program uses the command line interface in windows to prompt the user for the text configuration file.

6.1.10.1.2 UNIX Based Systems

Because the C++ DLL driver is multi-platform compile-able all that is needed for calculation is an interface to generate the needed configuration files.

6.1.10.2 Back-end Calculation

The goal is creating a seamless, error-free CA calculator that is solid in execution but flexible in input.

6.1.10.2.1 Seed

Seeds are ideally stored in a bitset, an array of booleans, a string, or an array of characters. If using bool or char to store a binary bit in C++ the stored size is an entire byte rather than a bit; this multiplies storage necessary by eight. In this research an OOP approach was taken to store the binary value as well as that nodes adjacencies used to calculate its next step in the same structure.

6.1.10.2.2 Rules

Rules are best represented by a structure that can retrieve a value in a single lookup.

The system is mentioned above and provides an excellent basis for speedy calculation of a rule.

6.1.10.2.3 Adjacencies

In CA, adjacencies are the elements that determine the next state of the bit. For one dimensional CA with a three-bit rule, adjacencies the current bit is combined with the bit to the left and right. Ideally adjacencies are calculated once and a pointer is used to refer to the memory address in which it is stored so optimal efficiency may be employed.

6.1.10.2.4 CA Calculation

The process of temporally iterating the seed is core functionality. The structure used to store the seed nodes in the OOP structure created was used to calculate the next time step.

6.1.10.2.5 End Results

For random number testing, the end result could be constructing a binary file or streaming random data to a suite designed to test that data for randomness.

6.1.10.3 Random Number Testing

Most test suites provide source code or in some cases the ability to inject a custom DLL that performs the random generation of fixed length words. If using source, keeping the generated pseudo-random data in memory is most efficient when using the test suite; as this reduces the file IO time to read and write files to disk. In the case of DLL generation the suite will need to perform a particular function that will return a single

pseudo-random number. Background calculation must be done to generate more numbers and source the seed.

6.1.10.3.1 Test Suites

In the Diehard test suite, there are 229 p-values (statistical probability values - for this circumstance it is the probability that this set of data is random) generated from the tests and an overall p-value, for a total of 230. The limits on passing were p-values between 0.0005 and 0.9995, a threshold of 0.05%.

6.1.10.3.2 Cycle and Transient Correlations

One inevitable global dynamic in Cellular Automata is that a cycle will develop. Using the algorithms above, a cycle detection can be written into the calculation to determine cycle and transient length to study these dynamics so they may be avoided in pseudo-random number generation.

6.2 DYNAMICS OF CELLULAR AUTOMATA

As mentioned before there are certain undeniable characteristics of Cellular Automata. Every possible seed or origin state must exist in a cycle or proceed and terminate into one. There are certain states however that cannot be generated by a specific rule, they must be a starting state or they will never be achieved. These state came to be known as “Garden of Eden” states or GOE’s. They are valuable in that they are usually the furthest point from each cycle; this gives a long sequence of random states before entering a cycle; we call this length the “Transient”.

Below is an example of a seed of length three:

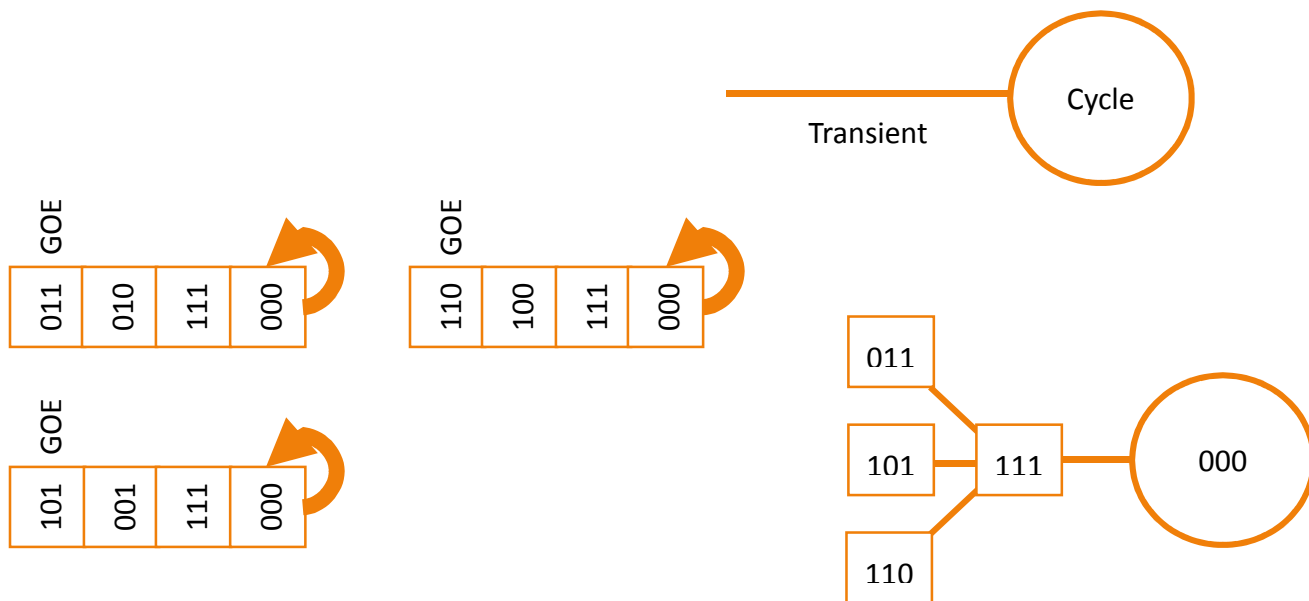


Figure 7: Seed length $k = 3$; transients and cycle.

6.2.1 Reversibility of Cellular Automata

Another algorithm was developed to reverse the ECA process which was dubbed accordingly the “Reverse Algorithm”. This algorithm recursively looks at previous possible states and eliminates them based on their neighbors. In this way we are able to have a simple test to know if any state is a GOE. This is useful in determining maximum transients and cycles which in turn can be used in matching the length of data to encrypt to a specific seed to ensure no bits are repeated. Below is the algorithm for rule 30 and seed ‘011’. The tables above the digits are the possible previous states.

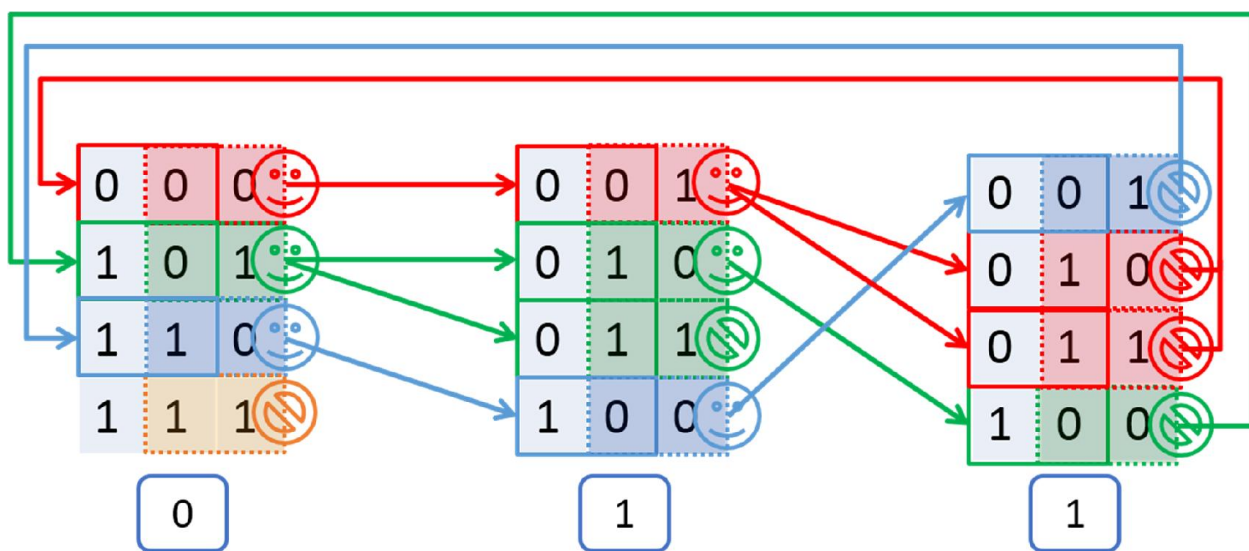


Figure 8: The reverse algorithm visualized with colored steps.

In chaotic CA a particular seed can only produce a single output, but in some cases two separate seeds can produce the same single output. This accounts for branches in the state diagrams above [6.2] and below [6.2.2.1]. This property makes finding which state

produced a particular next state difficult as it can sometimes yield multiple possibilities, or none in the case of a GOE. Each pre-image of three adjacencies must share with other pre-images surrounding it. This relationship is defined above between solid and dotted boxes, and arrows between them; they are the same adjacencies and they must match for a pre-image to be viable. The colors represent passes through the pre-images to rule out particular ones. All neighboring adjacencies must match (smiley) to find a previous state that can produce the tested seed; in the example above there are no total matches and therefore the tested seed is considered a GOE.

6.2.2 Garden of Eden States and Importance

6.2.2.1 Garden of Eden Introduction

GOE states are important in studying CA because they traverse every possible transient. This greatly reduces the amount of tests that have to be run to determine cycles for a particular seed length. There are holes in this assumption as some cycles in CA have no transients. Generally they are small in length for chaotic rules and can largely be ignored and avoided when choosing seeds.

For example, one of the most famous chaotic rules, rule 30; any even number seed with alternating bits will be permanently stuck in a cycle of a single state:

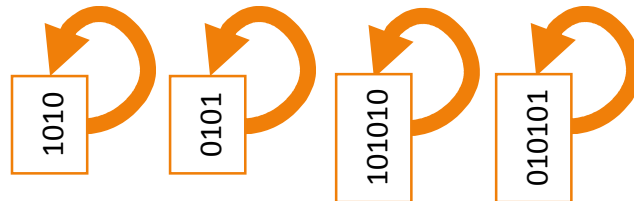


Figure 9: The single state failures of rule 30 and even seed lengths

Below is a state map for rule 30 and a k length of six. All states lead to 0 except 21 and 42 which are examples of the above bit-flipping. The binary was converted to decimal for visual ease.

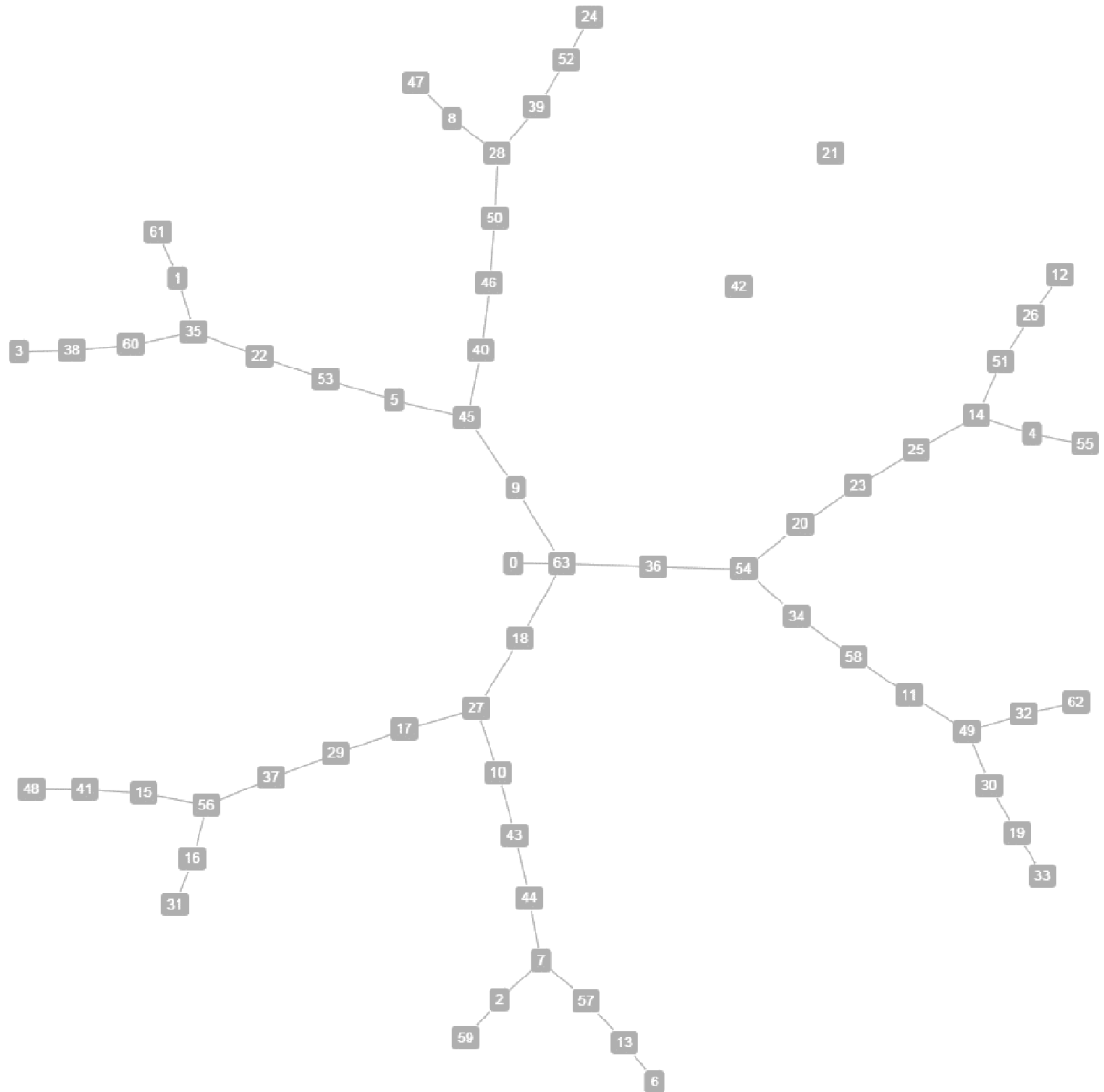


Figure 10: A full state map of seed length six.

After $k = 7$ the physics engine in the browser begins to slow and the states start merging together. The memory in the browser is also strained and viewing larger sized seeds becomes impossible. Many have put together much greater accumulations of states by using GPU technology and have succeeded with rule 30 up to length 14. Unfortunately the number of states being viewed at once is $2^{14} = 16384$ which is hard to differentiate. These state diagrams do however give insight into what chaotic CA resembles and how complex the patterns eventually get.

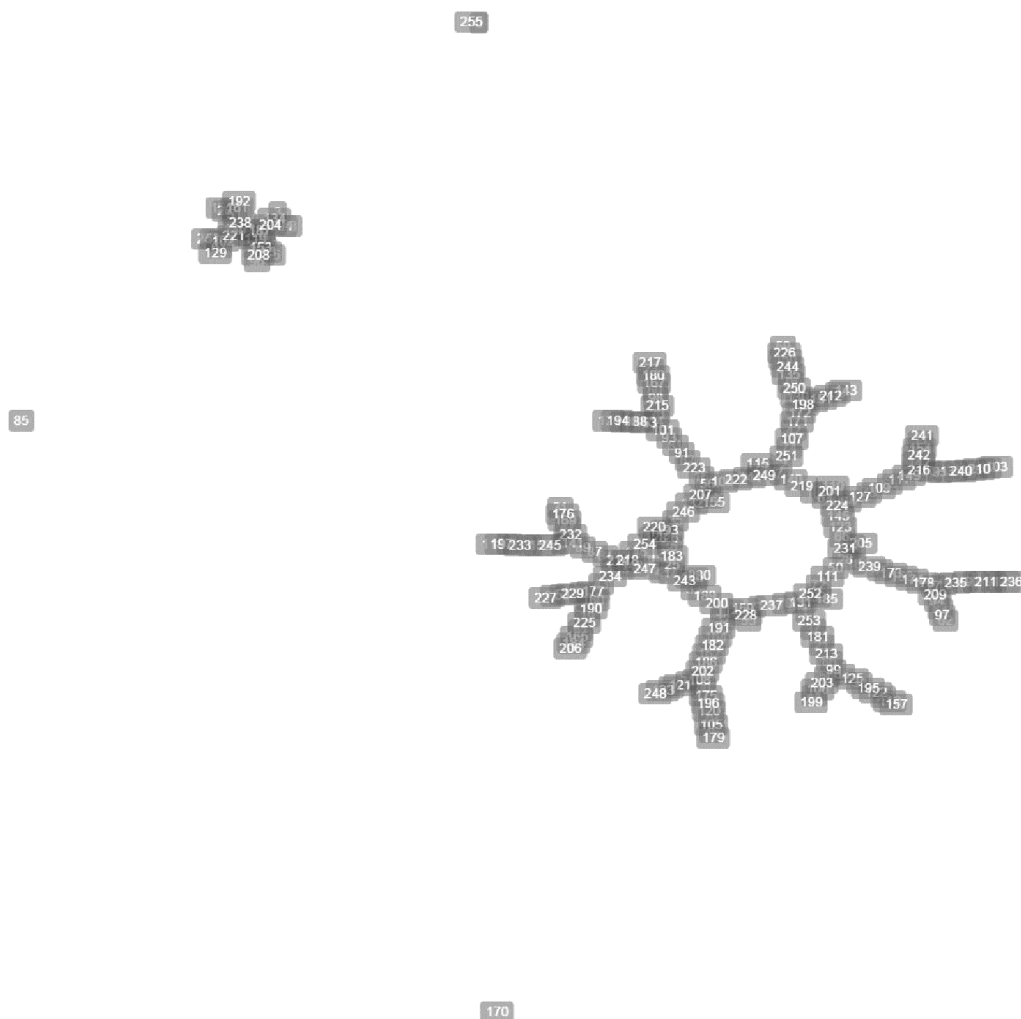


Figure 13: A full state map of seed length eight.

6.2.2.2 Numeric Garden of Eden Data

GOE states make up a small population of a large amount of possibilities for any seed length k . Represented below is a table with the numbers mentioned and graphed:

Table 5: Number of GOEs in a given length for $k=1$ to 32.

Size k	Total States	GOEs	% GOE	% Reduced from last k
1	2	1	50.00%	
2	4	1	25.00%	1
3	8	3	37.50%	0.333333
4	16	5	31.25%	0.6
5	32	6	18.75%	0.833333
6	64	12	18.75%	0.5
7	128	22	17.19%	0.545455
8	256	33	12.89%	0.666667
9	512	57	11.13%	0.578947
10	1024	101	9.86%	0.564356
11	2048	166	8.11%	0.608434
12	4096	280	6.84%	0.592857
13	8192	482	5.88%	0.580913
14	16384	813	4.96%	0.592866
15	32768	1373	4.19%	0.592134
16	65536	2337	3.57%	0.587505
17	131072	3962	3.02%	0.589854
18	262144	6708	2.56%	0.590638
19	524288	11382	2.17%	0.589352
20	1048576	19305	1.84%	0.589588
21	2097152	32721	1.56%	0.589988
22	4194304	55485	1.32%	0.589727
23	8388608	94094	1.12%	0.589676
24	16777216	159536	0.95%	0.589798
25	33554432	270506	0.81%	0.589769
26	67108864	458693	0.68%	0.589732
27	134217728	777765	0.58%	0.589758
28	268435456	1318777	0.49%	0.589762
29	536870912	2236162	0.42%	0.58975
30	1073741824	3791692	0.35%	0.589753
31	2147483648	6429246	0.30%	0.589757
32	4294967296	10901569	0.25%	0.589754

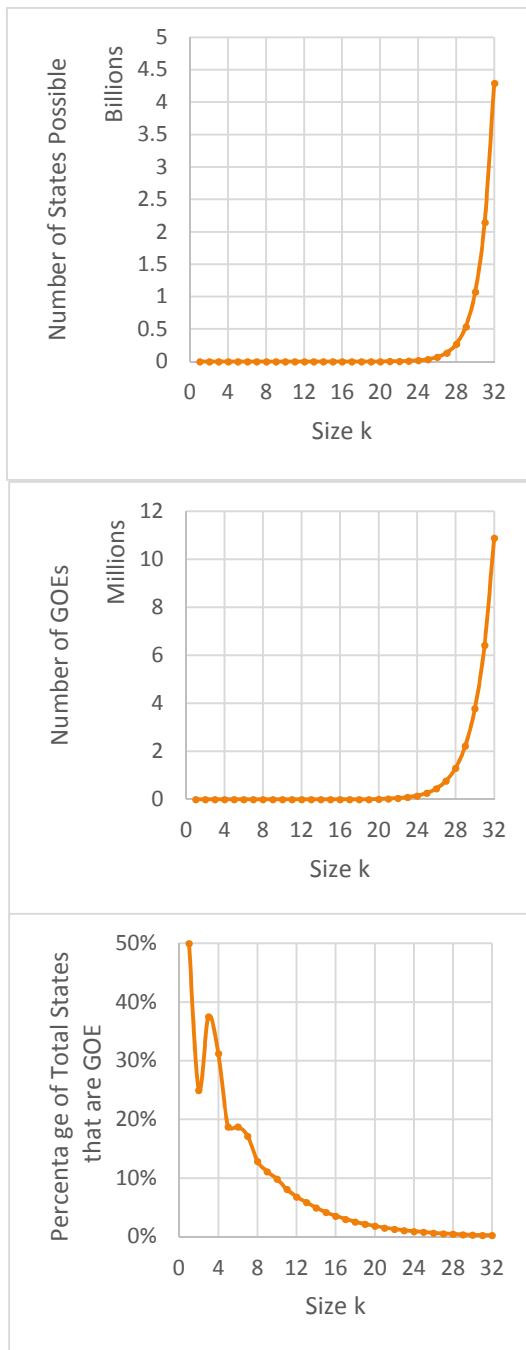


Figure 14: Three graphs visualizing Table 5.

Shown first is easily identifiable as a power graph, it is simply: $f(x) = 2^x$. It shows the number of possible states given a particular seed length; since each position in the seed can only have one of two states the number of possibilities is 2^k .

The second plot is the number of GOE states in every lot of seed possibilities. The graph is remarkably similar to the first and the instinctual explanation is more states leads to more GOEs. While this is true the reasoning behind this phenomena is more complex. This rotational symmetry will be discussed in the future.

The percentage of total states to GOEs seems to approach a constant ~ 0.58975 . This is directly related to new symmetries created when more possibilities are added.

6.3 ADJACENCIES AND MULTI-DIMENSIONAL CONFIGURATIONS

In traditional one dimensional ECA the two adjacent bits are used in conjunction with the current state of the bit to generate the next state, as mentioned in more detail in the introduction. Other research has been put forward to suggest using two dimensional arrays of bits to generate pseudo-random data [7]. Whilst programming these adjacencies for one and two dimensional CA it was discovered that the math associated with converting and allocating 2D into 1D was expandable to any dimension. This was unique to my research and no documentation has been found pertaining to this subject.

Below is a representation of how this was achieved:

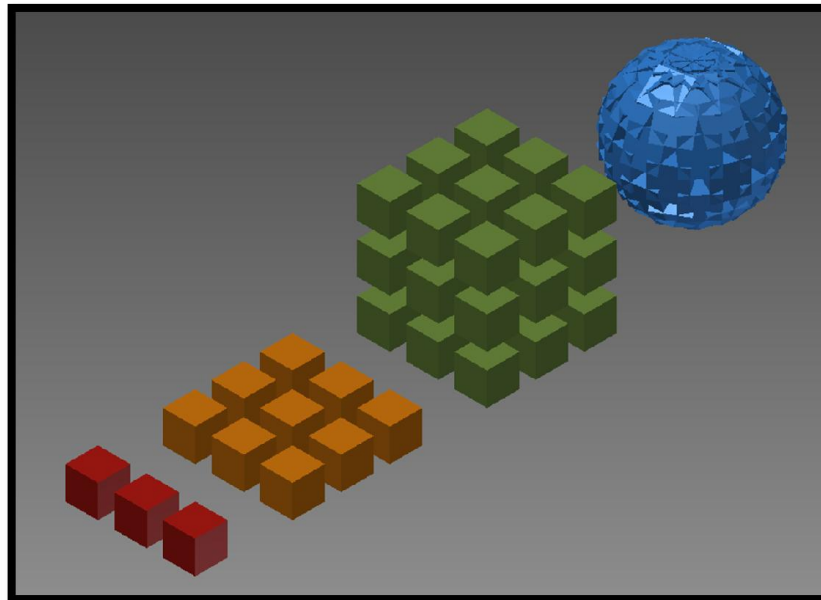


Figure 15: Dimensional adjacencies visualized. 1D to 4D.

6.3.1 Periodicity

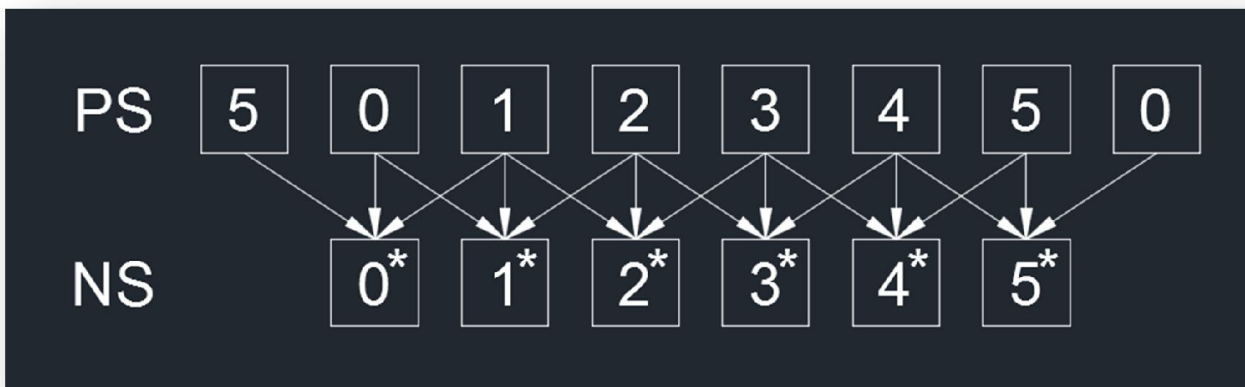


Figure 16: The periodic nature of a seed length of 6; indices 0-5.

Normally in 1D CA the left and rightmost adjacencies are looped back around to create a periodic CA. This periodicity creates a continuous chaotic pattern. Many researchers in this subject refer to the current seed as a “window” or “sliding window” [6] because it is just a snapshot of the ring of values. As the previous state, “PS” and the next state, “NS” progress each ring is adjoined to the next creating a cylinder of values that are used as a cypher. There were two terms to describe the two concatenation methods alternate



Figure 17: The differences in concatenation of each row created by CA.

and classical. This concatenation method was important when determining the effectiveness of the cypher.

In two dimensional CA the seed becomes a torus due to its periodicity. Two dimensional CA and its properties and variations are discussed in the next section.

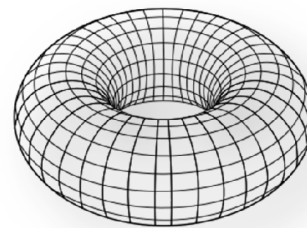


Figure 18: The toroidal nature of a two dimensional CA.

6.3.2 Neighborhoods

When discussing Cellular Automata in two dimensions two neighborhoods are generally discussed. In this research adjacencies were used to discuss the cells in the neighborhood. The two dimensional neighborhoods are Von Neumann and Moore neighborhoods.

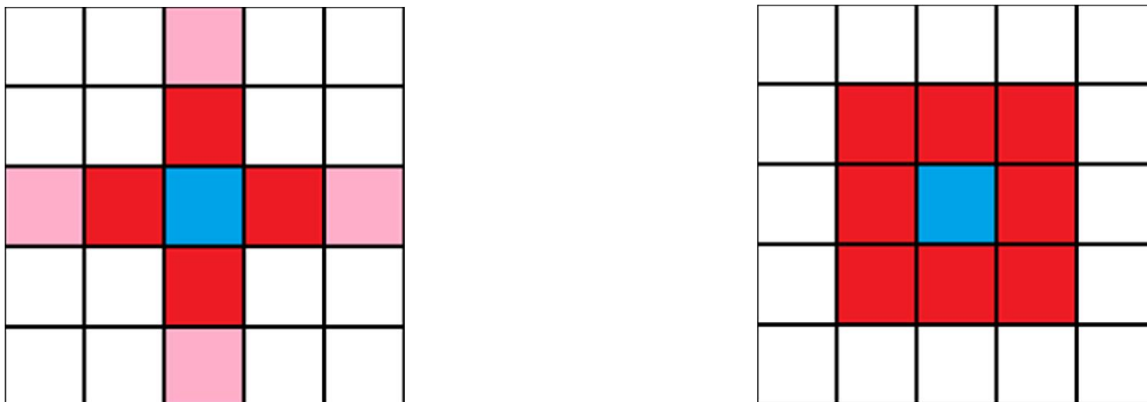


Figure 19: The Von Neumann adjacencies (left) versus the Moore adjacencies.

Through research with two dimensional CA with chaotic rules diehard results showed that Moore was a better candidate to produce more random results. This is explained simply by introducing more adjacencies that in turn produces more obscurity.

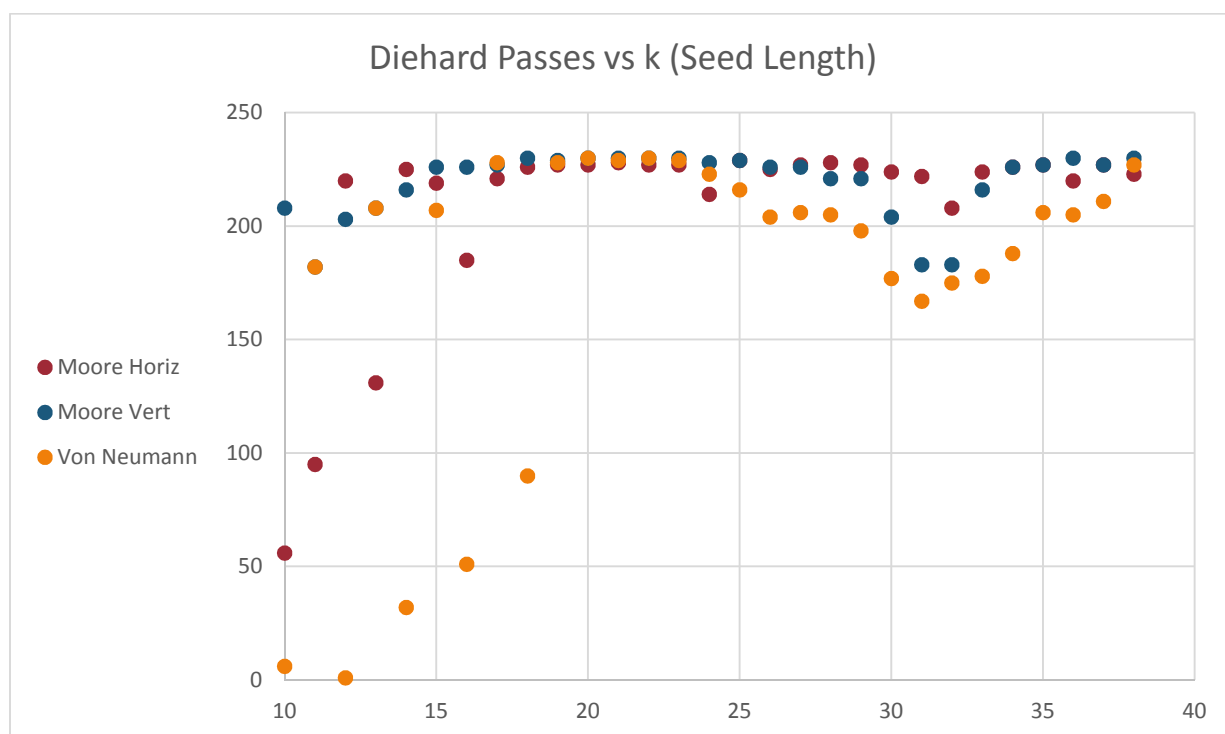


Figure 20: A graph of randomness versus the adjacency configuration used.

Diehard test suite uses 32 bit integers to test randomness. In configurations where $k \approx 32$ the tests tend to fail due to correlation of integers. In the graph above all three tests droop around 32. Von Neumann is the worst, then Moore vertical, then Moore horizontal. Using 9 adjacencies versus 5 works better for defeating correlation around 32. Vertical and horizontal are indications of how the adjacencies are reduced.

6.3.3 Extra Dimensional CA

Whilst programming the CA engine I discovered when creating a one dimensional memory array from a two dimensional CA that the formulas and methods could be used for any dimension.

Row Size = r

Number of cells = k

Dimension = δ

Number of Adjacencies = a

Moore :

$$a = 3^\delta$$

$$k = r^\delta$$

Von Neumann :

$$a = 2\delta$$

$$k = r^\delta$$

In the beginning of this section there is a visual of the progression of Moore Adjacencies; 3, 9, 27, 81. This is using the Moore equation for adjacencies, which gave rise to the first 4D calculation of CA. The code written to test multidimensional CA was written to be configurable and is therefore recursive; it is computationally intense. If a single configuration was used however a fixed code function could be written to have a constant execution time and speed calculation.

As dimension increases so does number of passing tests, also the 32 dependency decreases. On low numbers however the greater dimensions fail.

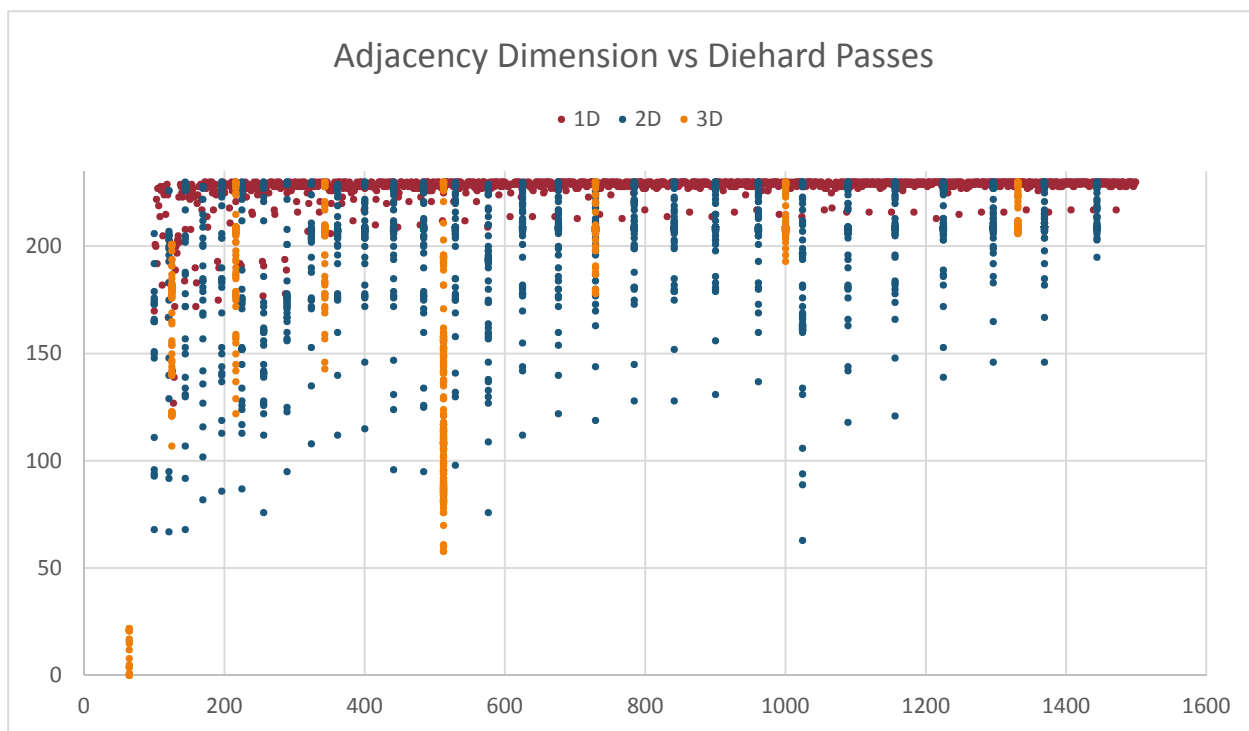


Figure 21: Dimensional differences of randomness.

6.4 MULTI-RULE CONFIGURATIONS

During research it was surmised that using multiple rules instead of a single rule would be beneficial to the randomness of the numbers and reduce the likelihood of a particular seed reaching a small loop.

First, a fixed method of using a particular rule was created so that we may see comparable results.

Whilst brainstorming three methods were constructed to best simulate multi-rule functionality.

6.4.1 Spatial Configuration

In this method a different rule is used for each k position. For example, in a two rule configuration, at index 0 rule 30 is used whereas the next index, 1 rule 45 is used, 30 for index 2 etc. This method is so used because it depends on a particular location in a given time step.

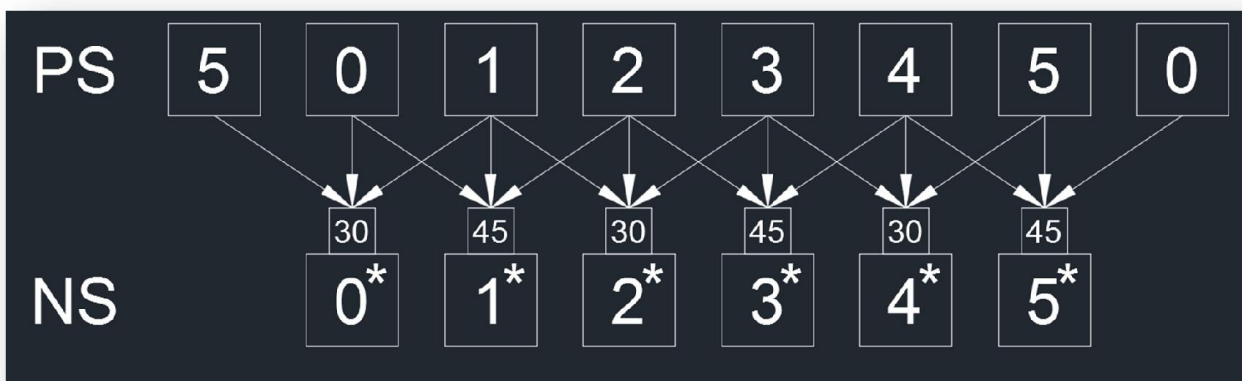


Figure 22: The special rule changes visualized.

6.4.2 Temporal Configuration

In a Temporal multi-rule configuration used rules change on the time step of the CA. For example, in a two rule configuration, at time step 0 all calculations use rule 30 whereas on the next time step (1) rule 45 is used to calculate the next step.

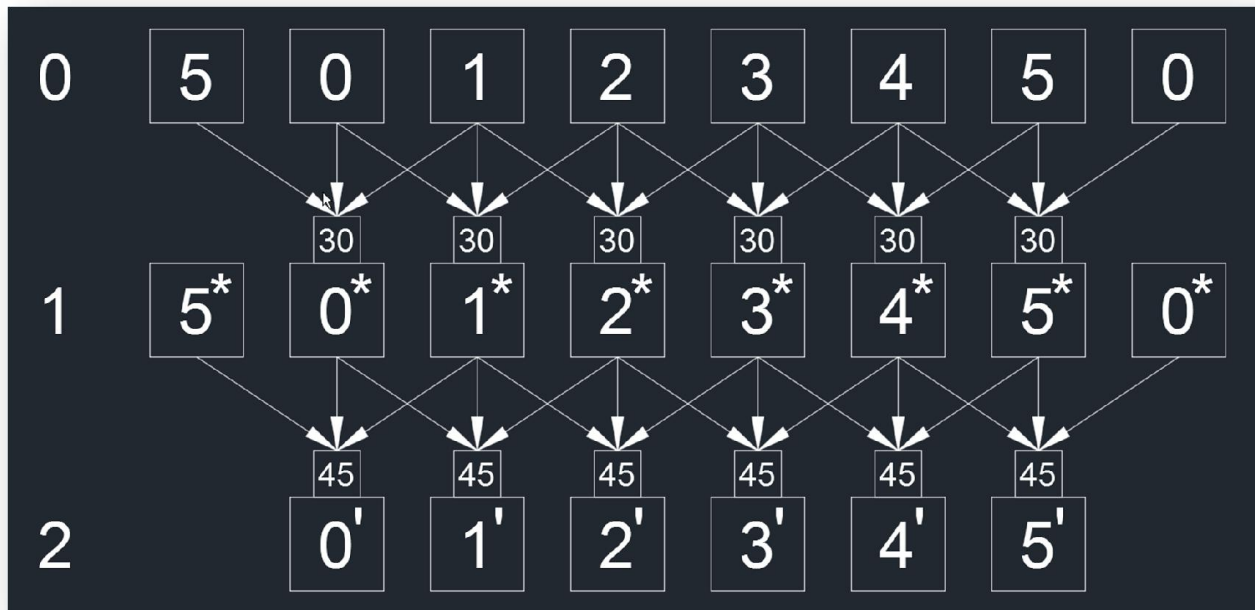


Figure 23: Temporal rule changes visualized.

6.4.3 Spatial-temporal Configuration

This configuration uses different rules inter-spatially. This is impossible in one dimensional configurations due to only one calculation being made. In a two dimensional context however many calculations can be made to reduce adjacencies to a single bit. The rules used in this research deal with three bits but in multi-dimensional configurations there are many more than three adjacencies.

6.4.3.1 Arrow: Von Neumann Adjacencies

Arrow was an ideal solution to the five adjacencies that two dimensional Von Neumann configuration presents. For example an up arrow consists of taking the left, right and top Von Neumann adjacencies and running a three bit rule on them. The resulting bit takes the place of the top bit and another calculation is made in the same time step to combine the top, middle and bottom bit for the final next state for the middle bit. This is easier demonstrated with a diagram. In two dimensional CA there are four different configurations for arrow. A binary string encoder/decoder was developed for the definition of arrow configuration. In the diagram the calculation shown is for bit "4".

In three dimensions there are 24 different configurations. The math is as follows.

$$\text{Dimension} = \delta$$

$$\text{Number of Configurations} = c$$

$$c = \prod_{i=2}^{\delta} 2i(i-1)$$

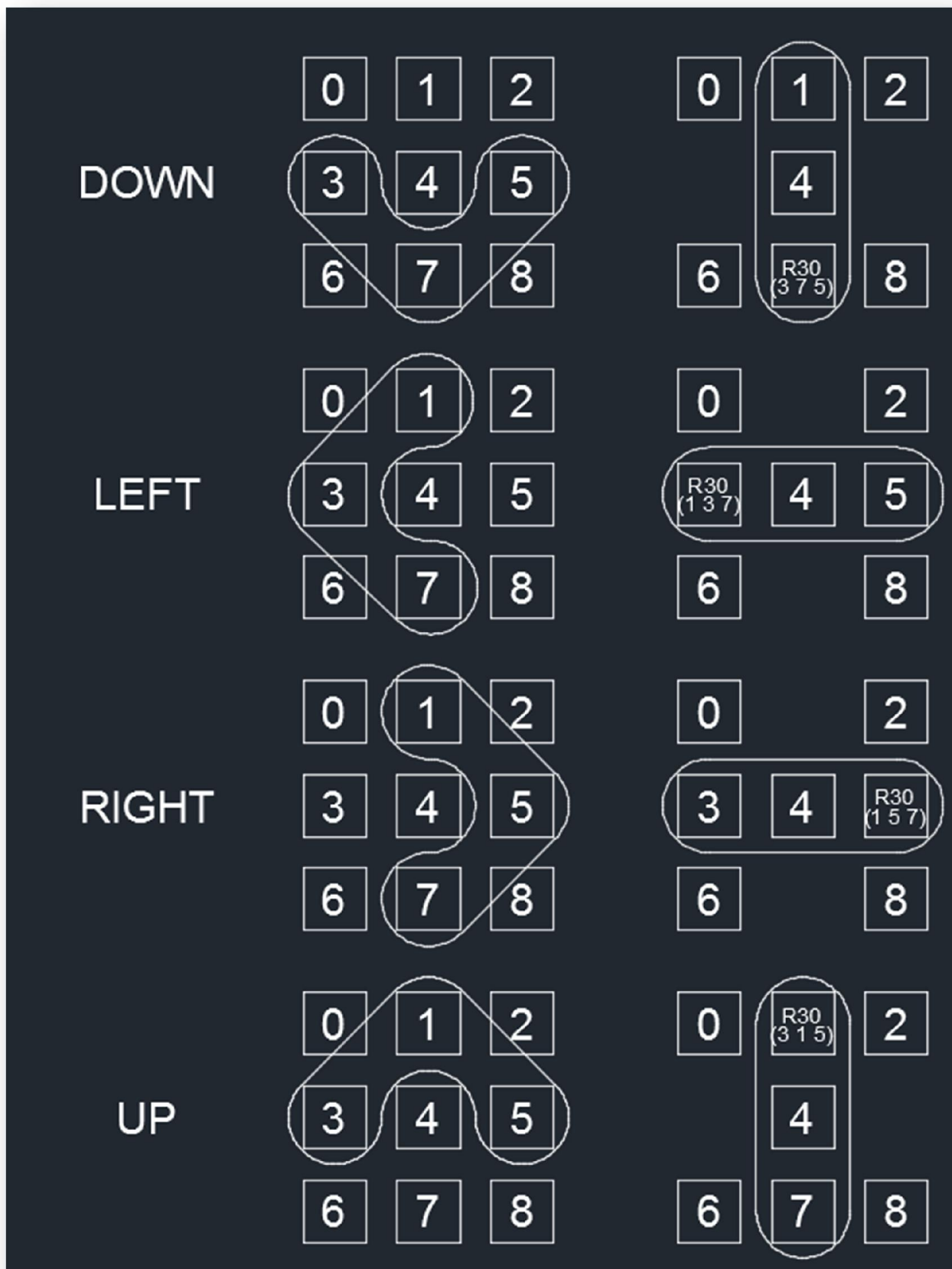


Figure 24: Arrow configuration: two-step process visualized.

6.4.3.2 Moore Adjacencies

Moore adjacencies are ideal for dealing with three bit rules as they are always divided evenly by three. The number of adjacencies is 3^δ , where δ is dimension. Naturally this can be reduced to a single bit fairly quickly. The number of steps to complete a single time step calculation is directly proportional to the dimension the CA is being calculated in. This reduction was classified into two categories: horizontal and vertical. Obviously this terminology breaks down in greater than two dimensions but the math remains the same.

This was programmed recursively and is therefore quite slow; if a fixed method was chosen the hardware or software performing the time steps could be hard-coded and would therefore be much faster than the recursive method used in this research. The recursive implementation yields higher configurability and was therefore our choice in testing.

In the diagram the calculation shown is for bit "4".

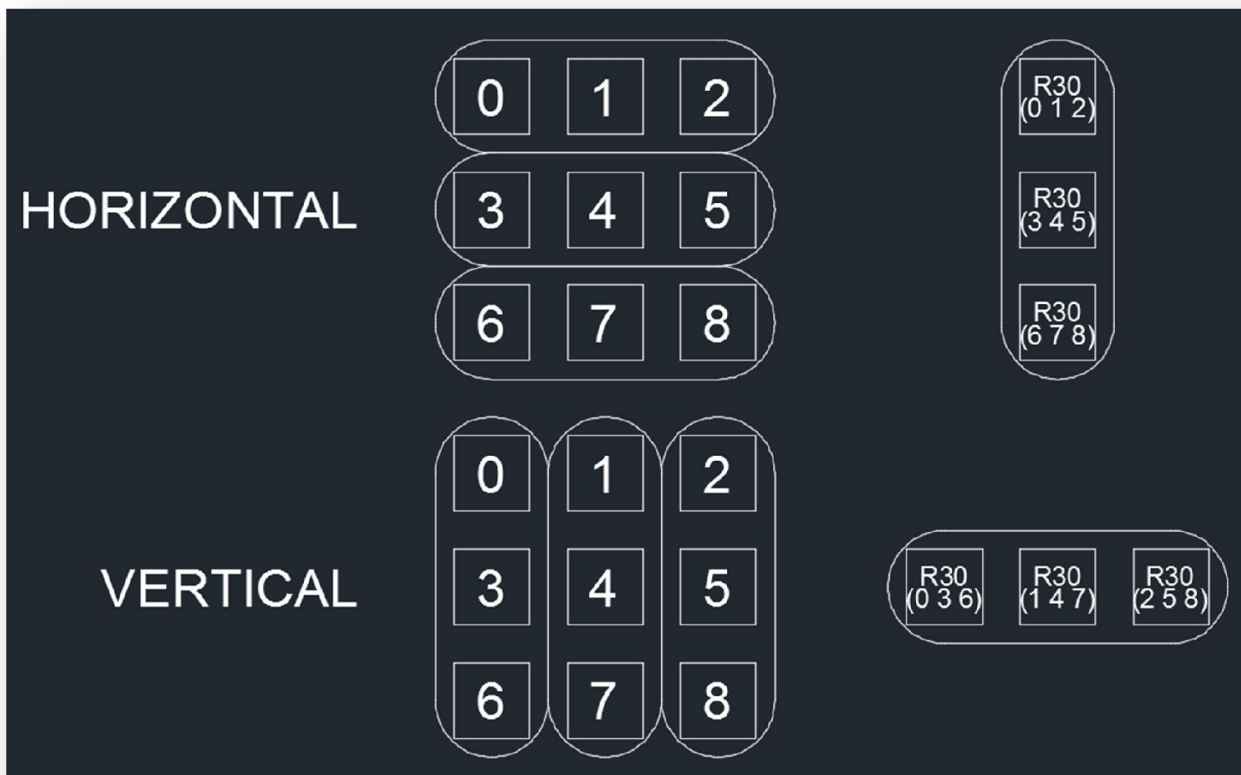


Figure 25: Moore configuration: two-step process visualized.

7 RESULTS

Whilst testing multitudes of configurations and rules and rule combinations it became increasingly clear that the Diehard test suite was inadequate for the upper end of the tests; the best configurations were passing all of the tests and not leaving much room to determine a better configuration.

7.1 ONE DIMENSIONAL

In one dimensional CA there are many promising configurations that may be possible to generate a stream of pseudo-random bits.

7.1.1 Single Rule

Rule 30 proved useful above $k \geq 113$. The only exception are k 's that are divisible by 32 minus one. Avoid $k \bmod 32 = 1$. This fallacy is suspected to arise from diehard as it uses 32 bit samples from the binary data and therefore shows correlation in testing.

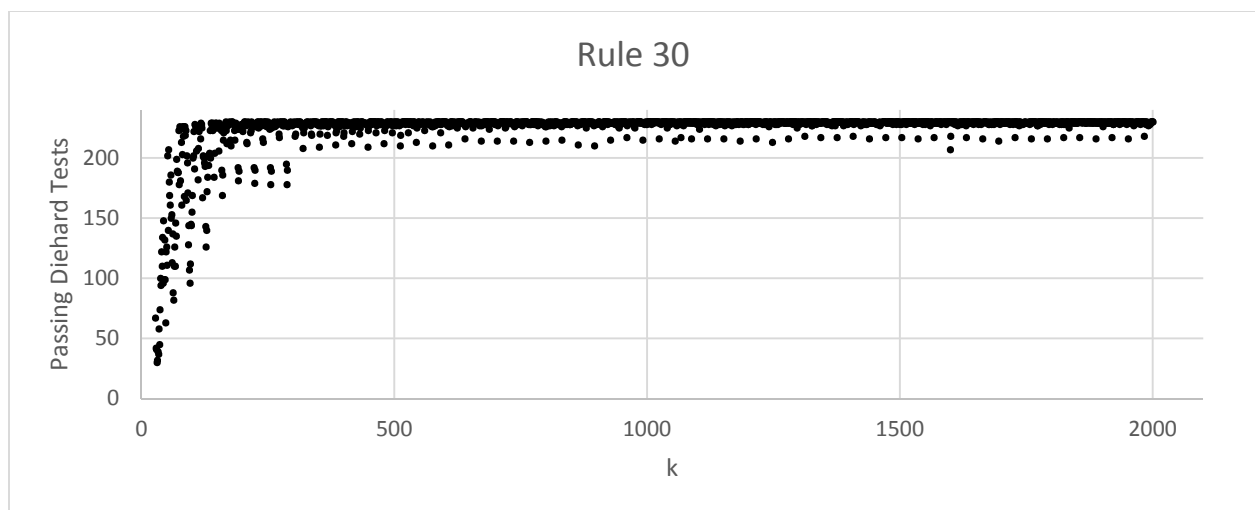


Figure 26: Single rule, one dimensional randomness.

7.1.2 Two Rules

For the two rule calculation all possibilities were tested in a 1D configuration. The accumulated data was quite large so it was injected into a MySQL database. 637,787 rows were accumulated and sorted and shown using Google Charts, PHP, and a bit of JavaScript magic. The reason this combination was chosen was to be able to interact with the charts and hover over a particular plot to find its value. Also, large compilation screens were generated so that many charts could be analyzed at once. The two rules are displayed separated by a "-". On the right the first letter is the concatenation configuration (A-Alternate, C-Classical) and the second letter is the multi-rule configuration (S-Spatial, T-Temporal).

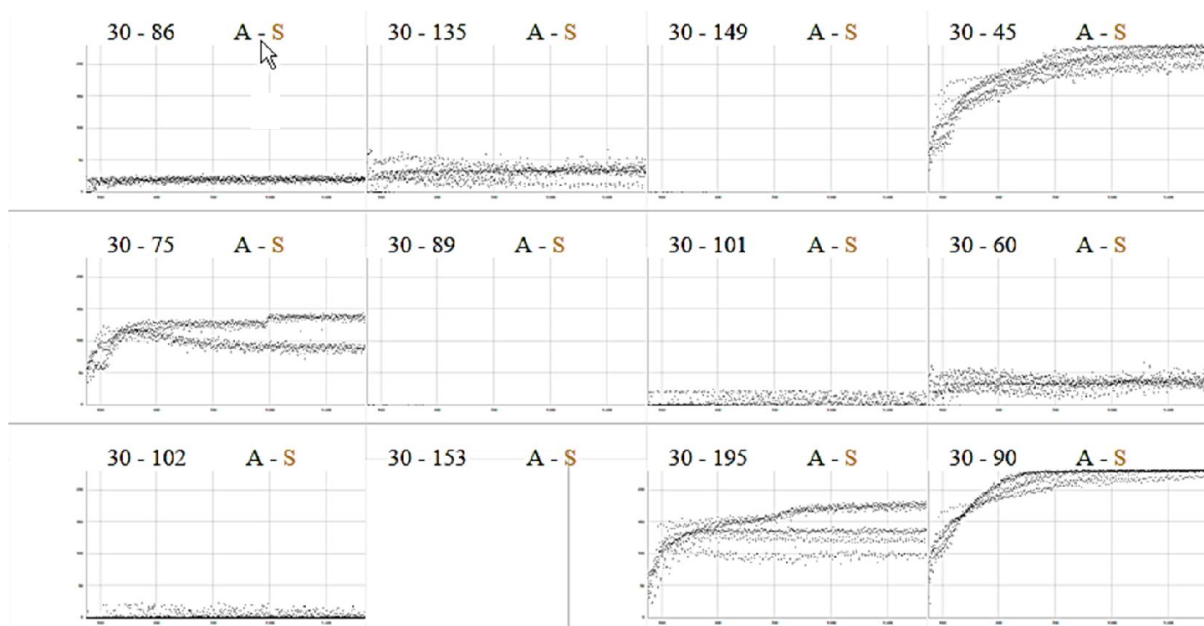


Figure 27: A small sample of Spatial-Alternate two rule tests

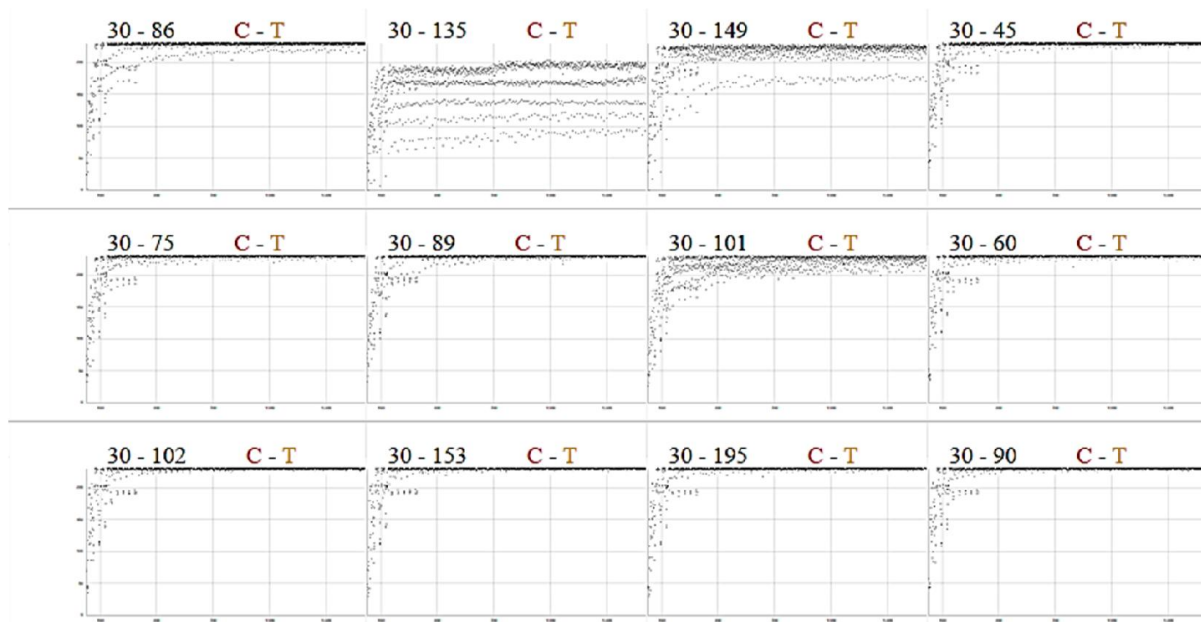


Figure 28: A small sample of Classical-Temporal two rule tests

It was determined that the best configurations almost always contained rule 90 and the classical temporal set of data performed universally better. A full listing of this data is available in the reference section.

Further research will have to be conducted for greater than two rules as the calculations performed with the two rule configuration took close to a month to complete.

7.2 TWO DIMENSIONAL

7.2.1 Single Rule

The single rule results in two dimensional CA showed a preference towards the Moore Horizontal configuration as shown earlier in section 6.3.2.

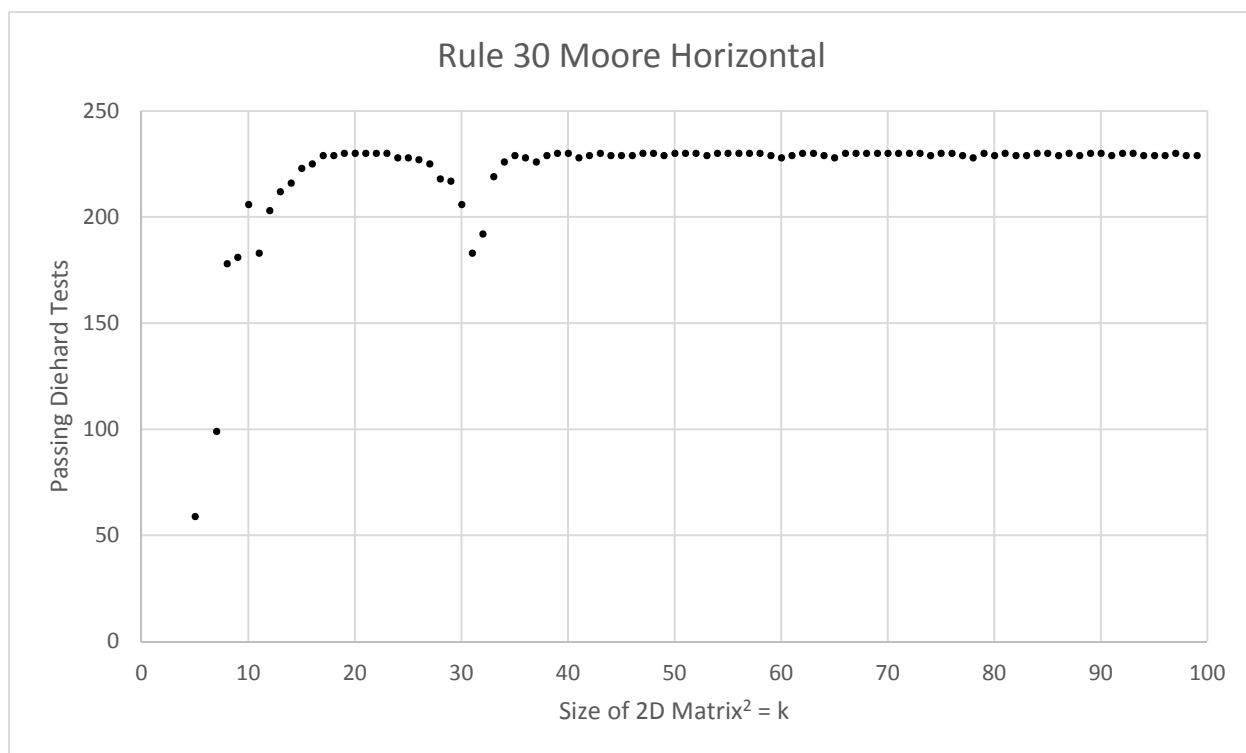


Figure 29: Randomness graphed for a two dimensional CA.

There is a drastic dip at size close to 32 as this is the size of data that Diehard Test Suite uses. In these tests there was a huge improvement over the one dimensional CA as it does not fall below 228 tests after size 40. Whereas one dimensional failed every $k \bmod 32 = 1$.

7.2.2 Two Rules

Using what was learned from one dimensional space with two rules a few tests were run to determine the best two rule combination for Moore Horizontal.

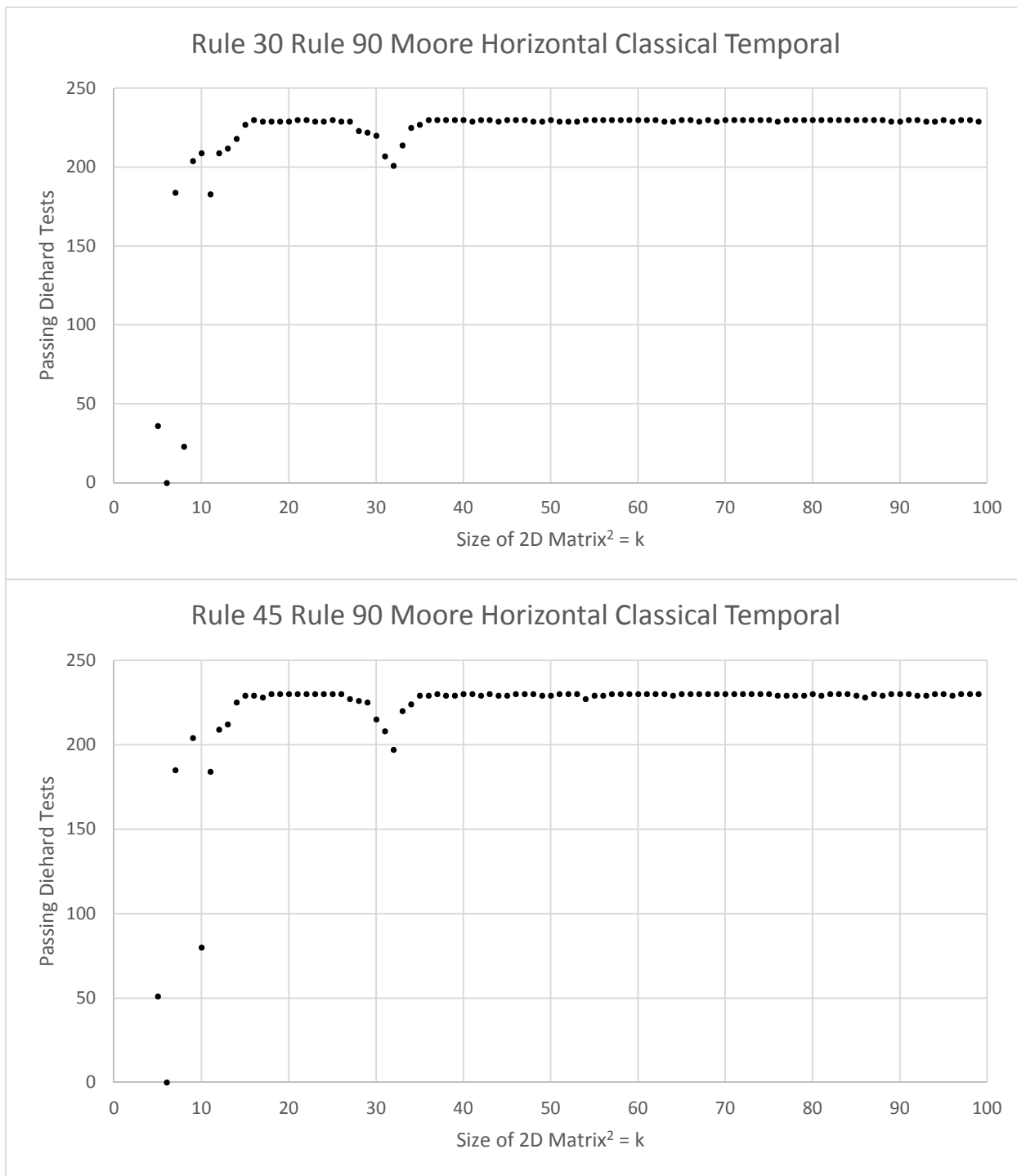


Figure 30: Randomness graphed for 2D CA using the successful 1D configurations.

First rule 30 and rule 90 were tried using a Classical Temporal configuration in two dimensional space. The two rule configuration is only mildly better than the single rule configuration which is what lead to the exploration of upper dimensions in CA calculation. Introducing another rule into the calculation increases complexity but seems to have little measurable effect at sizes greater than fifty.

7.3 THREE DIMENSIONAL

7.3.1 Single Rule

In a three dimensional space the Moore Adjacencies add up to a total of 27. Every cell uses 27 other cells to choose its next state. This exponential growth in complexity yields excellent results even using a single rule. At this point another testing suite was sought because Diehard is not thorough enough to determine small differences in results.

Considering every size increment also constitutes a power increase in k it is understandable how these tests are far superior as the sheer volume of bits involved. In other words, there are more bits dependent on other bits as well as an increase in total bits.

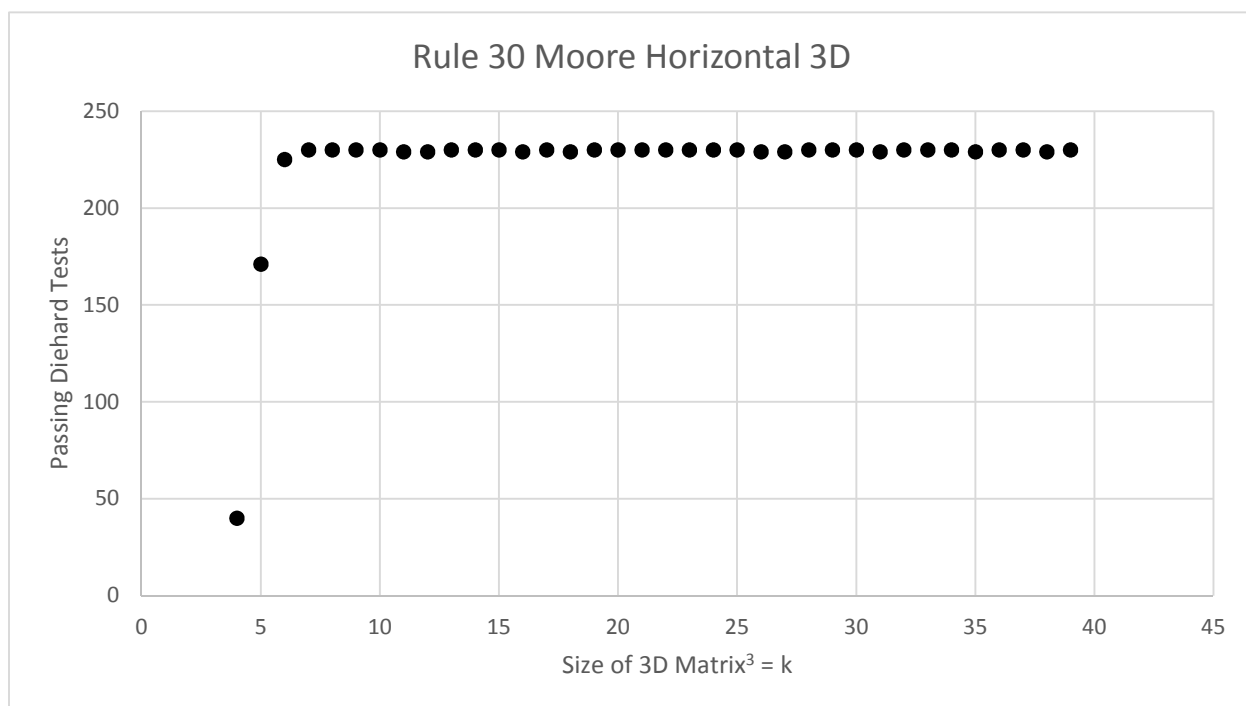


Figure 31: Three dimensional CA randomness graph.

7.4 DISCUSSION

This research was aimed at creating a better way to encrypt modern digital data. Current systems such as hashing and DES/AES are flawed in that given rise in computing capability the easier breaking the system will become. The reason DES was broken was this cause exactly. Industry's solution: double or quadruple the bits, so that it is harder to break. The same vulnerabilities exist the only difference is time to break. I propose a new system, such as CA, that after every time step becomes exponentially harder to break due to possible pre-images. The origin of course being a password or seed of your choosing. The way this data is generated is chaotic in nature and math does not exist to break CA's transformations.

An encryption system based on CA is far superior and the possibilities are so endless many generations of configurations could potentially provide security to everyone forever.

Of course this is not without some criticism; the only reason one would need such a system is to beat government cryptanalysis as everyone does not possess the resources to crack AES. Not only that but this system is strictly serial in that it cannot be done in parallel; every state depends on the previous.

No doubt this is the option of the future but much more research into implementation, standards and configurations must be done to make it viable.

PUBLICATION REFERENCE

- [1] WOLFRAM, S.: 'A New Kind of Science'. Champaign, IL: Wolfram Media, 2002.
- [2] WOLFRAM, S.: 'Random Sequence Generation by Cellular Automata', Advances in Applied Mathematics. Volume 7, Issue 2, June 1986, Pages 123-169.
- [3] SALMAN, K.: 'Analysis of Elementary Cellular Automata Boundary Conditions', International Journal of Computer Science & Information Technology. Volume 5, Number 4, August 2013
- [4] ANDREW WUENSCH, AND MIKE LESSER, 'The Global Dynamics of Cellular Automata', Reference Volume I, Addison Wesley Publishing Company, 1992, ISBN: 0-201-55740-1.
- [5] MARSAGLIA, G.: 'The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness', Florida State University, <http://i.cs.hku.hk/~diehard/>
- [6] SIPPER, M.: 'Generating Parallel Random Number Generators by Cellular Programming.' International Journal of Modern Physics Volume 7, Issue 2, 181-190, 1996. ISSN: 01291831.
- [7] FUKS, H; SKELTON, A.: 'Response Curves For Cellular Automata in One and Two Dimensions - An Example of Rigorous Calculations.' Aug. 9, 2011

8 APPENDIX

SUPPLEMENTARY REFERENCE CHARTS

8.1 TWO RULE 1D CHARTS

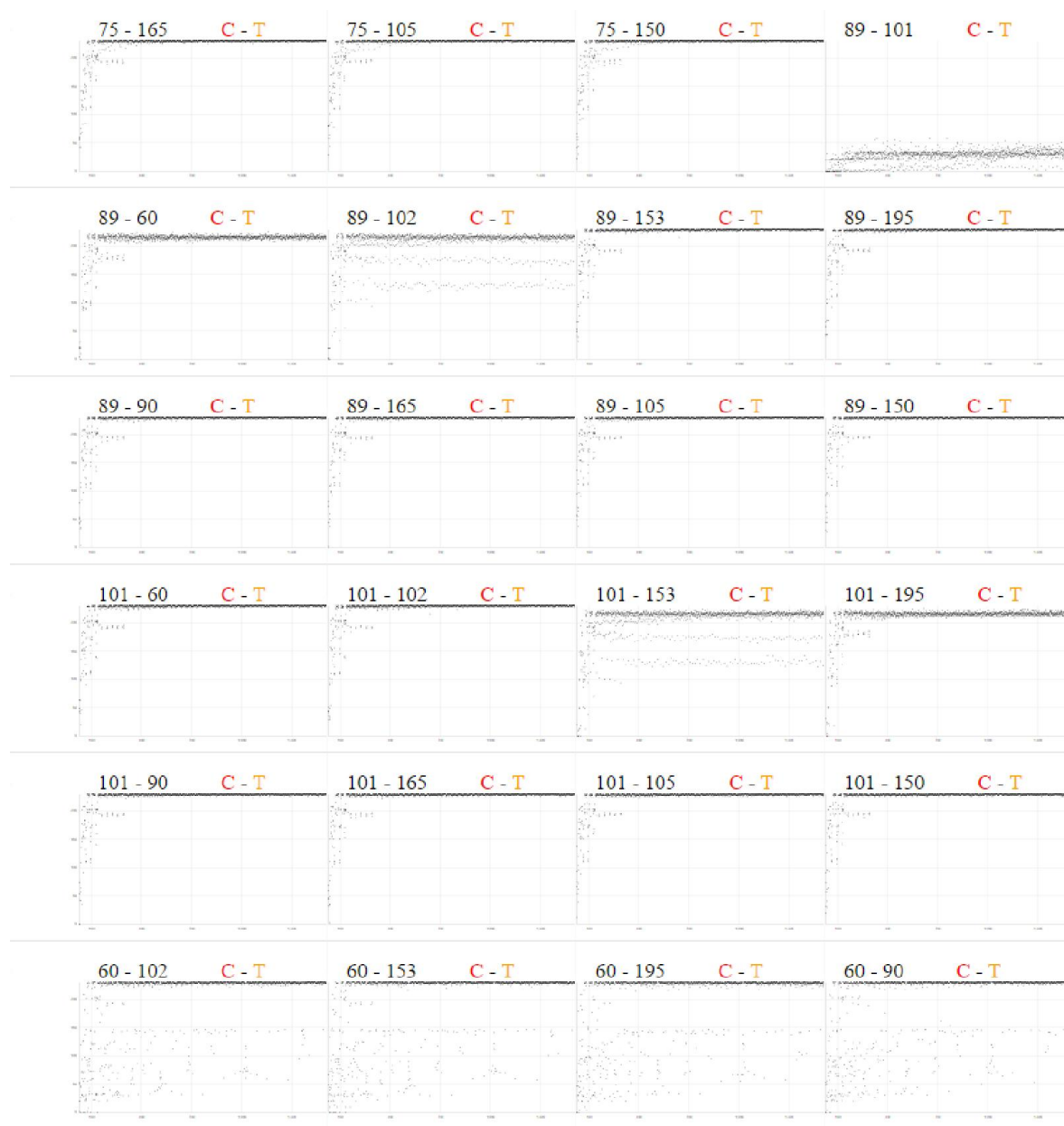
The following are supplementary reference charts gathered and stored into a MySQL database then queried and displayed using the Google Chart API. Some data is missing due to the fact that tests could not be completed because there were not enough random numbers. These test areas are identified with a red bar in the top left indicating that the data is nonexistent.

8.1.1 Classical Temporal











8.1.2 Classical Spatial



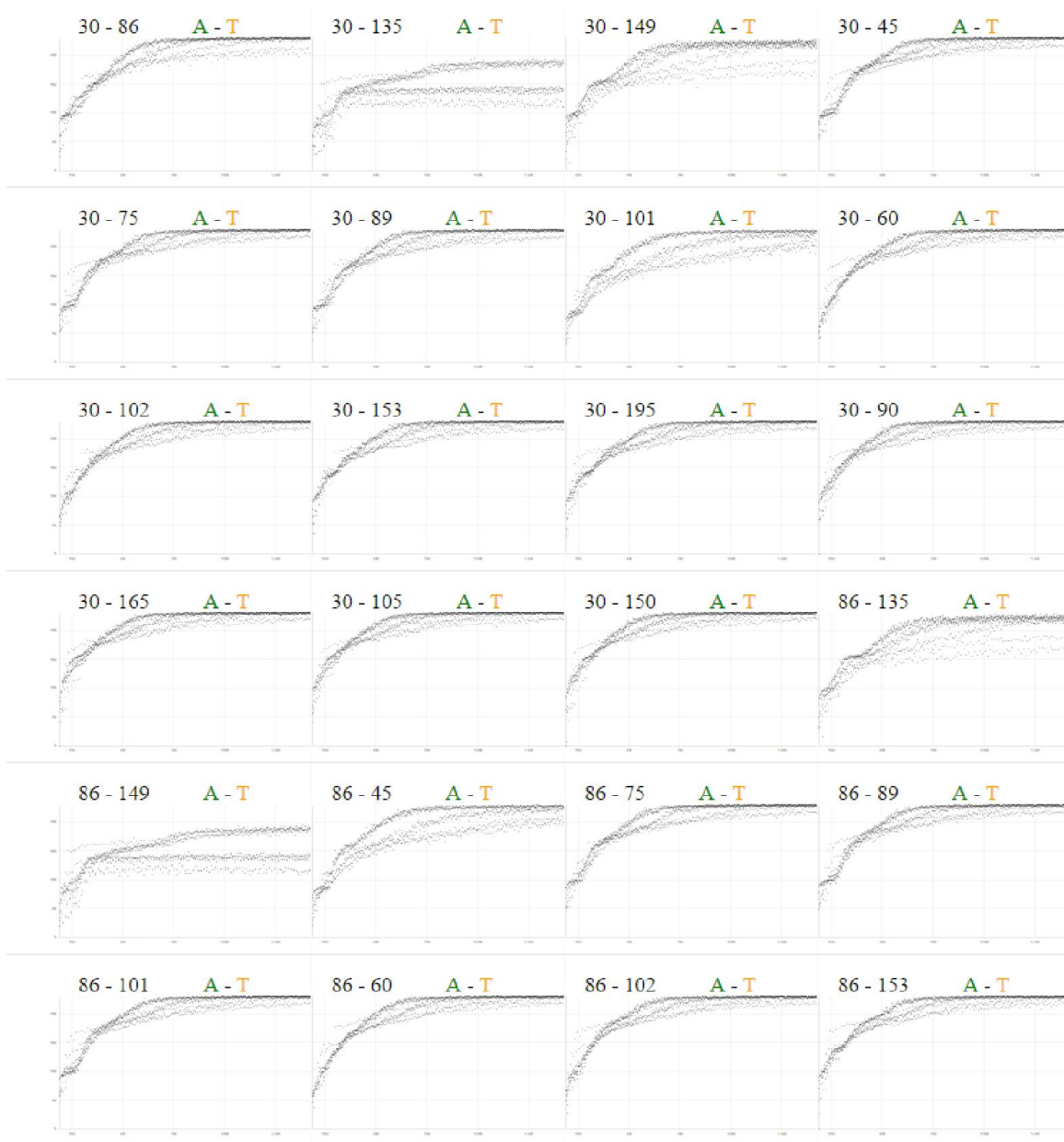




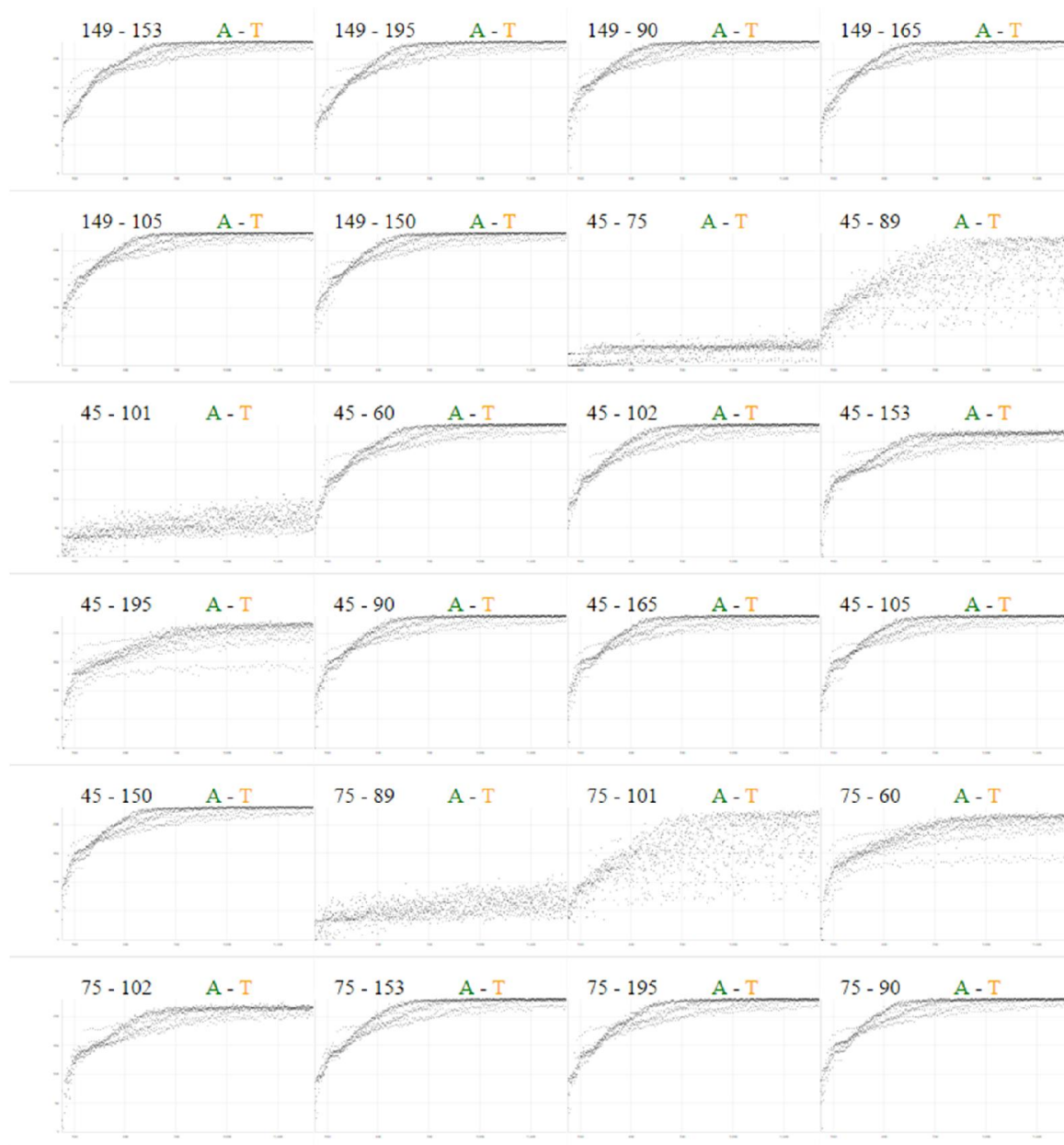


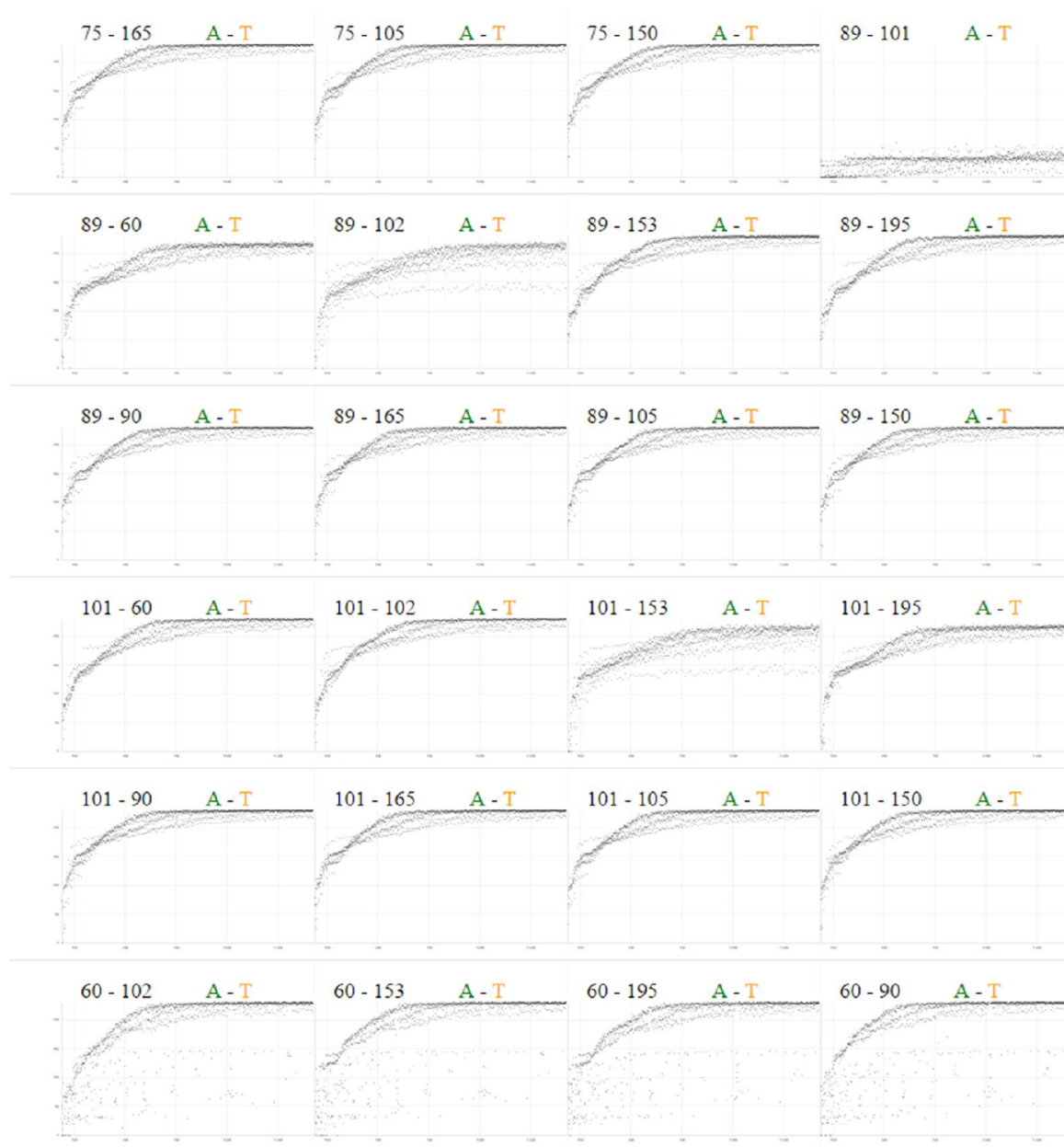


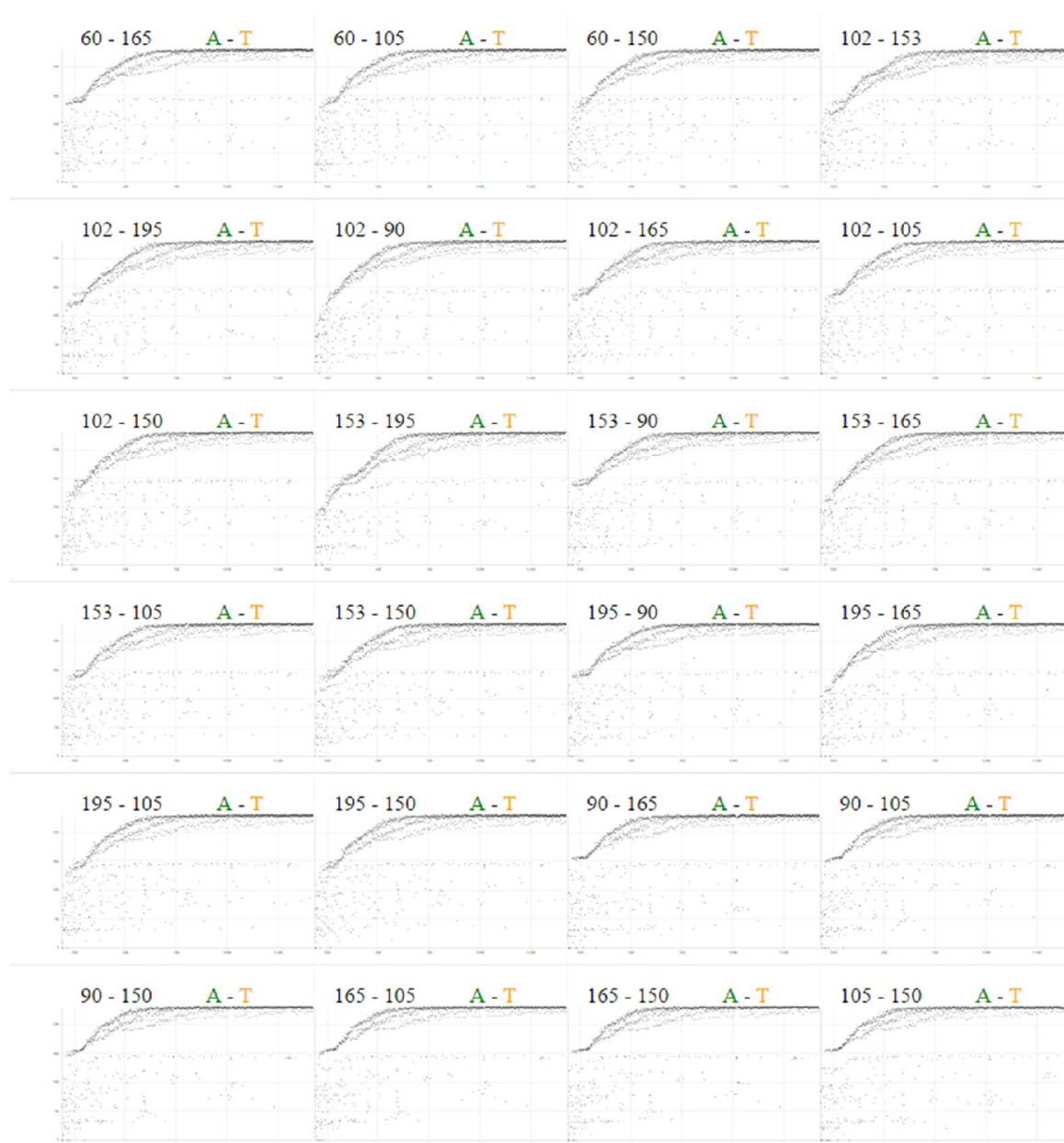
8.1.3 Alternate Temporal











8.1.4 Alternate Spatial

