Automatic Detection of Interesting Cellular Automata

Qitian Liao

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-150
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-150.html

May 21, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

First I would like to thank my faculty advisor, professor Dan Garcia, the best mentor I could ask for, who graciously accepted me to his research team and constantly motivated me to be the best scholar I could. I am also grateful to my technical advisor and mentor in the field of machine learning, professor Gerald Friedland, for the opportunities he has given me. I also want to thank my friend, Randy Fan, who gave me the inspiration to write about the topic. This report would not have been possible without his contributions. I am further grateful to my girlfriend, Yanran Chen, who cared for me deeply. Lastly, I am forever grateful to my parents, Faqiang Liao and Lei Qu: their love, support, and encouragement are the foundation upon which all my past and future achievements are built.

Automatic Detection of Interesting Cellular Automata

by Qitian Liao

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Daniel D. Garcia
Research Advisor

5/10/2021
(Date)

* * * * * * *

Professor Gerald Friedland
Second Reader

5/10/2021
(Date)

Abstract

A cellular automaton consists of a grid of cells, and the grid can be in any finite number of dimensions. Each cell is in one of a finite number of states, and evolves with respect to time steps according to a set of evolution rules based on the previous states of its neighbors and itself [21]. The evolution rules are applied iteratively for as many steps as desired to produce new generations. There are many possible configurations; this report specifically explores two-dimensional outer totalistic cellular automata using the Moore neighborhood with decay, meaning that sick cells are not able to recover and have to move one step closer to death at each generation.

The challenge when searching for interesting patterns in two-dimensional cellular automata is a huge parameter search space. The number of possible combinations of rule parameters can easily exceed 2^{18}. Our research focused on adjusting the rules to find new, interesting spaceships (oscillating translators that move across the grid). Existing research has not discovered a clear pattern among rules that generate spaceships.

Manually searching for the interesting rules would be unrealistic, but fortunately, the introduction of neural networks has revolutionized a variety of tedious classification tasks. This report explores the use of neural networks to detect interesting cellular automata rules, specifically Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), feature extraction, entropy analysis, and other techniques. We then put the trained machine learners into practice and detected several new rules with only three states. We discovered an entire *family* of spaceships of different periods, as well as many other interesting results.

Acknowledgement

Throughout my years at UC Berkeley, I met many amazing friends, peers, and mentors, all of whom have contributed to my accomplishments and progress. It is very important for me to recognize all the people who have helped me during my graduate studies and the completion of this report. First I would like to thank my faculty advisor, professor Dan Garcia, the best mentor I could ask for, who graciously accepted me to his research team in my undergraduate years and constantly motivated me to be the best scholar I could. He has always been enthusiastic about my work and gave me tremendous support and encouragement. As a busy person, he always made time for me whenever I needed discussion or feedback, and his comments enlightened me in many ways when writing this report. I am also grateful to my technical advisor and mentor in the field of machine learning, professor Gerald Friedland, for the opportunities he has given me and the invaluable guidance during the weekly meetings. I would also like to thank my fellow graduate student and friend, Randy Fan, who gave me the inspiration to write about the topic. Parts of this report were adapted from an unpublished class project report Randy and I worked on during our graduate years. This report would not have been possible without his contributions. To my girlfriend, Yanran Chen, who supported me in every way possible during the pandemic. My life would have been mundane without her immeasurable love and constant company. Lastly, I am forever grateful to my parents, Faqiang Liao and Lei Qu: their love, support, and encouragement are the foundation upon which all my past and future achievements are built.

List of Figures

1.1 Neighborhood configuration 2
1.2 Transitional state diagram of the Game of Life 3
1.3 Spaceships in the Game of Life 5
1.4 Examples of famous spaceships 6
1.5 “Instant birth, gradual death, no recovery” Model 7
2.1 Wolfram’s rule 30 11
3.1 Example of a generated frame 16
3.2 Stitched square grid 18
3.3 Examples of stitched images 19
3.4 Entropy pattern distribution 22
4.1 One-period spaceships 25
4.2 A non-spaceship that looks like one 26
4.3 Two-period spaceships 27
4.4 Tagalongs of the two-period spaceships 27
4.5 Similar two-period spaceship 28
4.6 Extended two-period spaceships 28
4.7 Tagalongs of four-period spaceships 31
4.8 Three four-period spaceships 32
4.9 Another three four-period spaceships 33
4.10 Eight-period spaceships 34
4.11 Tagalongs of eight-period spaceships 34
4.12 Frankenstein spaceships 35
4.13 A four-period rake 35
4.14 An eight-period rake 36
4.15 Another eight-period rake 37
4.16 The new Life 38
4.17 Destroyed spaceships in collision 39
4.18 Example of a “combined spaceship” 39
4.19 Example of a “murdered spaceship” 40
4.20 Another example of a “murdered spaceship” 40

4.21 A rocket spaceship 41
4.22 Example of a "combined spaceship" 42
4.23 Example of a "murdered spaceship" 43
5.1 Possible cellular automata state transitional diagrams 47

List of Tables

3.1 Structure of the RNN 17
3.2 Structure of the CNN 21

Contents

Abstract i

Acknowledgement ii

List of Figures iii

List of Tables v

1 Introduction 1

2 Related Work 10

3 Methodology 14

3.1 Dataset Generation 14

3.2 Sequence Training with RNN 16

3.3 Data Preprocessing, Feature Extraction, Training with CNN 18

3.4 Entropy Analysis 21

4 Spaceship Discoveries 23

4.1 Spaceships in three-state cellular automata 23

4.2 Spaceship collision behaviors in three-state cellular automata 38

4.3 Other interesting discoveries 40

5 Future Work 44

6 Conclusion 50

7 References 52

A Appendix 55

A.1 Cellular Automata Generation Algorithm 55

A.2 The 35 Selected Interesting Rules 56

A.3 Frame Extraction 57

A.4 Recurrent Neural Network Implementation 57

A.5 Image Stitching Function 57

A.6 Image Feature Extraction with NASNet-Large 58

A.7 Image Feature Extraction with Image Pixels 59

A.8 Convolutional Neural Network Implementation 59

A.9 Image Cross-Entropy Computation 60

A.10 Maximum Memory Capacity Prediction 60

A.11 Spaceship Image and GIF generation 61

A.12 Initial configuration of the gliders 63
A.12.1 Code for figure 1.3, the Game of Life 63
A.12.2 Code for figure 1.3, the light-weight spaceship 63
A.12.3 Code for figure 1.3, the mid-weight spaceship 64
A.12.4 Code for figure 1.3, the heavy-weight spaceship 64
A.12.5 Code for figure 4.1 64
A.12.6 Code for figure 4.2 65
A.12.7 Code for figure 4.3 65
A.12.8 Code for figure 4.3 66
A.12.9 Code for figure 4.3 66
A.12.10 Code for figure 4.3 66
A.12.11 Code for figure 4.3 66
A.12.12 Code for figure 4.5 67
A.12.13 Code for figure 4.6 (1/5) 67
A.12.14 Code for figure 4.6 (2/5) 67
A.12.15 Code for figure 4.6 (3/5) 67
A.12.16 Code for figure 4.6 (4/5) 68
A.12.17 Code for figure 4.6 (5/5) 68
A.12.18 Code for figure 4.8 (1/6) 68
A.12.19 Code for figure 4.8 (2/6) 68
A.12.20 Code for figure 4.8 (3/6) 69
A.12.21 Code for figure 4.9 (4/6) 69
A.12.22 Code for figure 4.9 (5/6) 70
A.12.23 Code for figure 4.9 (6/6) 70
A.12.24 Code for figure 4.10 (1/2) 70
A.12.25 Code for figure 4.10 (2/2) 71
A.12.26 Code for figure 4.12 (1/2) 71
A.12.27 Code for figure 4.12 (2/2) 72
A.12.28 Code for figure 4.13 72
A.12.29 Code for figure 4.14 73
A.12.30 Code for figure 4.15 74
A.12.31 Code for figure 4.16 74

A.12.32 Code for figure 4.17 74
A.12.33 Code for figure 4.18 74
A.12.34 Code for figure 4.19 75
A.12.35 Code for figure 4.20 75
A.12.36 Code for figure 4.21 76
A.12.37 Code for figure 4.22 76
A.12.38 Code for figure 4.23 77

1 Introduction

Merriam-Webster defines a cellular automaton as follows [16]:

> cellular automaton, **sel**-yuh-ler aw-**tom**-uh-ton
> [noun]
> an element in a computer simulation composed of semi-autonomous interacting elements, specifically: any of such elements that are visualized on a computer screen as square or hexagonal cells comprising an array, grid, or lattice, that are controlled by similar but separate software routines or hardware devices, that can exist in a number of states, that are influenced by the states of their neighbors, and that are used to simulate diverse complex systems.

The most common configurations are the von Neumann and the Moore neighborhoods shown in Figure 1.1, which include the surrounding four and eight cells respectively. There are three types of two-dimensional cellular automata rules [1]: totalistic rules depend only on the states of the cells in the neighborhood, outer totalistic rules also depend on the state of the center cell, and growth totalistic rules make any cell that becomes live remain live forever. Cellular automata have attracted much attention among scientists and have been regarded by Stephen Wolfram as the "new kind of science" [1].

We normally restrict ourselves to systems whose behavior we can readily understand and predict because otherwise, we cannot be sure that the system will do what we want. However, unlike carefully engineered machinery, everything in nature is fundamentally made of particles flowing in space with arbitrary rules. Cellular automaton, like nature, operates under no such constraints of predictability or controllability. Applying a simple cellular automaton rule to a simple initial configuration can lead to a result that shows an immense level of complexity. The most fascinating aspect of it is that it seems to involve generating something from nothing, a practice that humans are simply not used to. Therefore, because of the resemblance between cellular automata and nature, it is natural to think of their dynamics as a micro-world where the cells constitute their own ecosystems.

Arguably the most famous cellular automaton is John Conway's the Game of Life that was initially revealed to the public in a 1970 Scientific American article [14]. The Game of Life is a two-dimensional cellular automaton with two possible states, alive and dead, using the Moore neighborhood and the following set of rules [20]:

1. Any live cell with two or three live neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. All other dead cells stay dead.

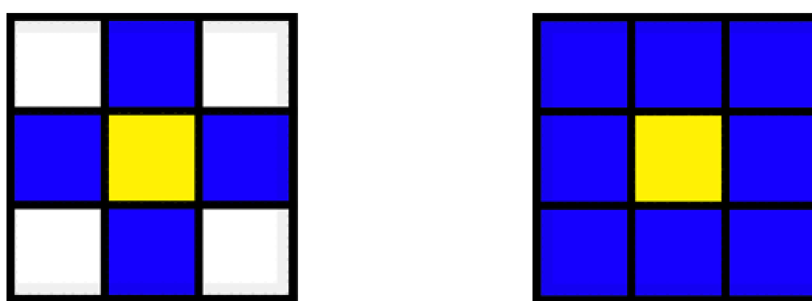


Figure 1.1: The von Neumann neighborhood (on the left) includes the surrounding four cells of the center cell. The Moore neighborhood (on the right) includes the surrounding eight cells of the center cell.

Since there are two possible states in the Game of Life, the live cells satisfying the survival rule and the dead cells satisfying the born rule will be alive in the next generation, and the remaining cells will all be dead. The probability of surviving and being born is $\frac{2}{9}$ and $\frac{1}{9}$ respectively. The evolution rule is often visualized using cellular automaton state transition diagrams, where each vertex on the graph represents one of the states, and each directed edge represents a viable evolution from one state to another. The probability of that transition, if available, will be highlighted on the edge. Figure 1.2 shows the transitional state diagram of the Game of Life.

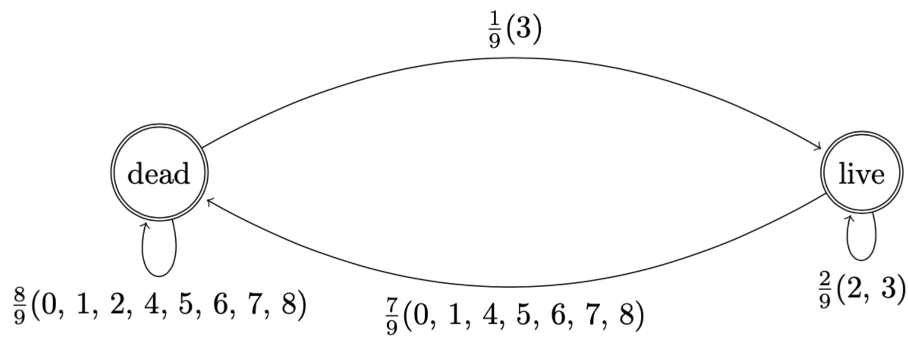


Figure 1.2: Transitional state diagram of the Game of Life. The probability of the transition followed by the specific satisfying state requirements in parentheses is highlighted on the corresponding edge. For example, the edge connecting the dead state to the live state represents the action of a cell being born, which has a $\frac{1}{9}$ chance of occurring (only when exactly three live cells surrounded it in the previous step).

Cellular automata become much more complicated and interesting when there are more than two possible states. In this case, apart from the live (i.e, healthy) and dead cells, there are also transitional dying cells (i.e., sick cells) in between and we are free to implement new rules to define how all of these states transition to other states. In the rest of the report, healthy and live cells refer to the same thing and are thus used interchangeably. These newly added rules, combined with the survival and born rules, constitute a new set of evolution rules.

Some of the most frequent patterns generated by the rules are “still life”, which are static patterns that do not change between generations, and “oscillators”, which are periodic patterns that return to their initial state after a finite number of generations [21]. While these patterns are fun, the feature that makes the Game of Life so well-known is undoubtedly the discovery of “spaceships” [23]. *Spaceships*, also known as translating oscillators or “fish”, are automata that travel by looping through a short series of iterations and end up in a new location after each cycle returns to the original configuration [18]. They are usually considered the most interesting pattern and are widely used for modeling complicated nonlinear systems in computational science, physics, chemistry, and biology [21]. If we think of cellular automata as an ecosystem, then the spaceships are a unique kind of independent life form within. Finding space-

ships can potentially help us answer questions like how accurately we are simulating biological life, or whether our artificial life forms can adapt to a changing environment.

Period and *speed* are the two frequently-utilized metrics to describe a spaceship. Period refers to the number of ticks a pattern must iterate through before returning to its initial configuration [20]. The speed of the spaceship is expressed in terms of the metaphorical "speed of light", c [17, 19]. The speed of light is a propagation rate across the grid of exactly one step, either horizontally, vertically, or diagonally, per generation. Since a cell can only influence its nearest neighbors, the speed of light is the upper bound to the speed at which any pattern can move. Generally, if the spaceship in a two-dimensional automaton is translated by (x, y) after n generations, then the speed v is defined as:

$$v = \frac{\max(|x|, |y|)}{n} c$$

The most famous spaceship found by Conway is the first one in Figure 1.3 with a period of 4 and a speed of $\frac{c}{4}$, as it takes four generations for a given state to be translated by one cell diagonally. There are many other cellular automata rules besides the Game of Life that can produce spaceships. Figure 1.4 shows some of the famous spaceships and *glider generators*, a pattern with a main part that repeats periodically, like an oscillator, and that also periodically emits spaceships [24], that have been generated with other sets of rules featured in Cellular Automata Rules Lexicon [6].

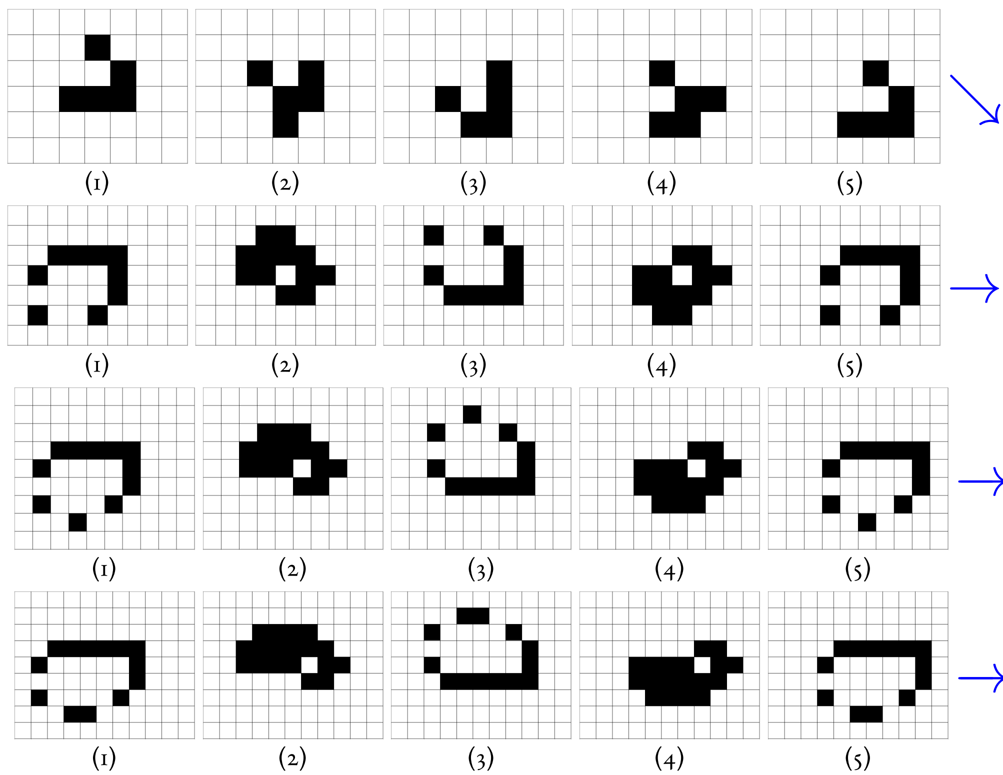


Figure 1.3: Spaceships generated by the Game of Life "2,3/3/2" rule. The top is the original spaceship first found by Conway. The rest are the light-weight, mid-weight and heavy-weight spaceships respectively. All four spaceships have a period of four. The light-weight, mid-weight and heavy-weight spaceship have a speed of $\frac{c}{2}$, as it takes four generations for a given state to be translated by two cells. Code is provided in A.12.1, A.12.2, A.12.3, A.12.4.

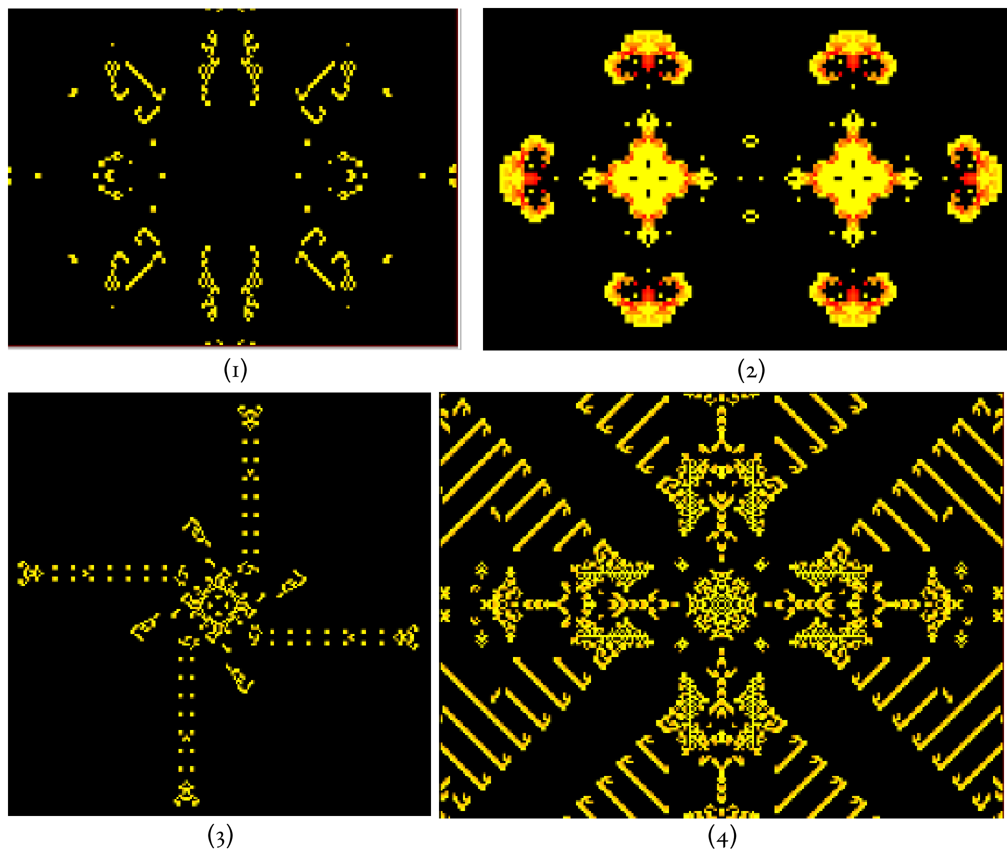


Figure 1.4: Four examples of discovered glider generators with different rules. They are "Brian's Brain" with rule "/2/3" (top left), "Burst" with rule "0,2,3,5,6,7,8/3,4,6,8/9" (top right), "Brain6" with rule "6/2,4,6/3" (bottom left), and "Star Wars" with rule "3,4,5/2/4" (bottom right) [6].

The spaceships in Figure 1.4 belong to the category of outer totalistic generations of two-dimensional cellular automata and are generated using the Moore neighborhood with decay. In other words, they use the "Instant birth, gradual death, no recovery" model depicted in Figure 1.5.

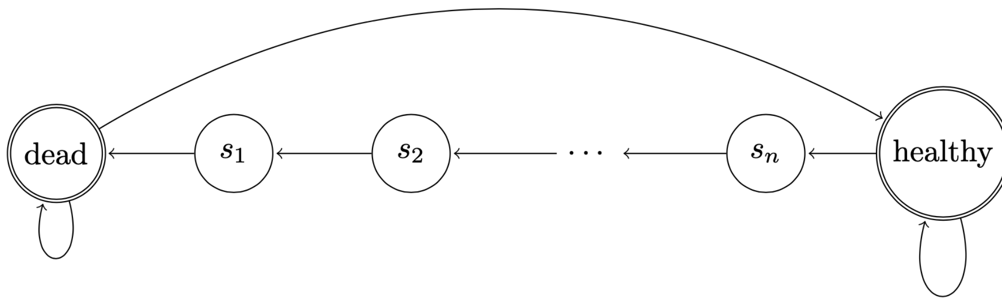


Figure 1.5: ”Instant birth, gradual death, no recovery” Model. Healthy cells can get sick. Sick cells are not able to recover, and they will be one step closer to death at each step. Dead cells cannot be born sick.

The behavior of sick cells under this model is deterministic as they are not able to recover and can only approach one step closer to death at each step. The healthy cells that do not satisfy the survival rule will become sick and inevitably enter the path of gradual death. Hence, we do not need extra parameters to categorize the behavior of sick cells. Specifically, like the Game of Life, three parameters constitute the rules of the model:

1. The survival rule that determines which of the live cells survive in the next step.
2. The born rule that determines which of the dead cells are born in the next step.
3. The number of total possible states.

The canonical way to represent the evolution rules of the generations is “S/B/C”, where S, B, and C represents the survival rule, the born rule, and the count of states cells can have respectively [6]. Hence, the Game of Life can be summarized as the “2,3/3/2” rule.

Not all rules are able to produce beautiful results like the ones in Figure 1.4. In fact, the results from most of the rules turn out to be unappealing. Configurations under some rules always die out, while others might lead to explosive growth. It is worth noting that spaceships exist for many unstable rules, especially those that lead to explosive growth. However, in this case, there is no real value in exploring them because they often disappear quickly and move around recklessly without clear patterns. There-

fore, we shall only consider spaceships for stable rules that exhibit bounded growth and eventually yield a finite number of gilders. One potential problem is that there may be some carefully constructed initial configuration within an interesting rule that could lead to explosive growth or stasis [2]. However, statistically it has an extremely low probability if the initial configuration is randomly generated. In this report, we decided to define the rules that satisfy these requirements using the "Instant birth, gradual death, no recovery" model with the Moore neighborhood as interesting. Specifically, the definition of *interesting rules* in this report includes:

1. It uses the Moore neighborhood.
2. It uses the "Instant birth, gradual death, no recovery" model, which means the evolution rule consists of the survival rule, the born rule, and the number of possible states.
3. Random initial configurations will always eventually stabilize (i.e., non-static and non-explosive).
4. It produces a finite number of spaceships.

All the remaining rules in the same model that lead to stasis, noise with no discernible patterns moving across the screens, or some patterns other than spaceships, are classified as boring. Finding out what the interesting rules are and what the spaceships look like is a daunting task. Under most circumstances, it is impossible to tell whether the rule is interesting or boring just by looking at the parameters of the rules. Furthermore, in one single case where we have a total of 10 possible states (i.e., 8 sick states), there are 2^9 survival rules and 2^9 born rules, which already leads to a total of 2^{18} combinations of rules. Given the fact that we may also want to explore rules with other numbers of total states, this eventually becomes an impossible task if it needs to be done by hand. That led us to explore automated detection of interesting rules so that users do not have to manually go through the process.

This report explores the possibility of using deep neural networks to detect these in-

teresting cellular automata rules. Deep neural networks, a branch of machine learning, are computational algorithms that can extract information from complicated data to detect patterns or trends which are too convoluted for human brains and other computer software. The most unique property of neural networks is that once trained, they can learn and adapt to new situations on their own. In this way, their learning process resembles the cognitive development of the human brain, which is made of neurons, the fundamental building unit for information transmission. These characteristics make neural networks much better candidates than humans to distinguish the interesting rules in cellular automata. We will train the neural network on a dataset consisting of samples of interesting and boring rules so that the machine learner can gradually recognize the decisive properties that distinguish interesting from boring rules. After the training, validating, and testing processes, our machine learner would be ready to dive into the remaining search space of rules that have not yet been classified and collect the interesting ones. The best part about it is that humans do not need to be involved in the exploration process at all, which is the most tedious and time-consuming step. All we have to do is manually inspect the rules that have been classified as interesting by our machine learner and record the spaceships within the patterns if they have been classified correctly.

Specifically, we first built the dataset from scratch by programming the generation algorithm and applying data augmentation to selected known rules. We used Recurrent Neural Networks (RNN), Convolutional Neural Networks (CNN), feature extraction, entropy analysis, and other techniques to help find interesting rules that generate spaceships. We then put the trained machine learners into practice and detected several new rules with only three states. We discovered an entire *family* of spaceships of different periods, Frankenstein spaceships, the new Life, and many other interesting results.

2 Related Work

John Conway, regarded as the father of cellular automata and the "founder of life", first described this elegant mathematical model of computation in 1970. From his famous rules of the Game of Life emerged a five-celled organism that moves diagonally across the grid. This discovery has attracted a group of fanatics who dedicated themselves to constructing rules in hopes of spotting new life forms. However, there has not been a systematic method of identifying interesting rules and the progress of searching has been rather slow.

The hype for cellular automata reached its peak when Stephen Wolfram published "A New Kind of Science" in 2002, which is also regarded as the encyclopedia of cellular automata [1]. In the book, he introduced a large variety of cellular automata with many arbitrary rules that generate interesting results. However, the most important lesson from his book is that complexity arises from simplicity and there is incredible richness in the computational universe. Even the simplest rule can produce the most complicated and unpredictable behavior. The vast space of the computational universe and the scarcity of the discovered rules gives us the potential to mine the interesting rules and harness them for our purposes. Wolfram describes the process of looking for something interesting in the space of cellular automata as very different from our accustomed approach of building models step by step while ensuring that we have control over their behaviors. He makes the rather counterintuitive claim that we should not try building anything at all. Instead, we should just define what we want and then search for it in the computational universe. It is sometimes very easy and fast to find what we want. For example, Wolfram quickly came across with rule 30 [22] in Figure 2.1, which is a one-dimensional cellular automata rule with two states and later became one of the best-known generators of apparent randomness, just by enumerating the rules. However, in other cases, it might take much longer, like it took Wolfram millions of attempts to find the simplest universal Turing machine.

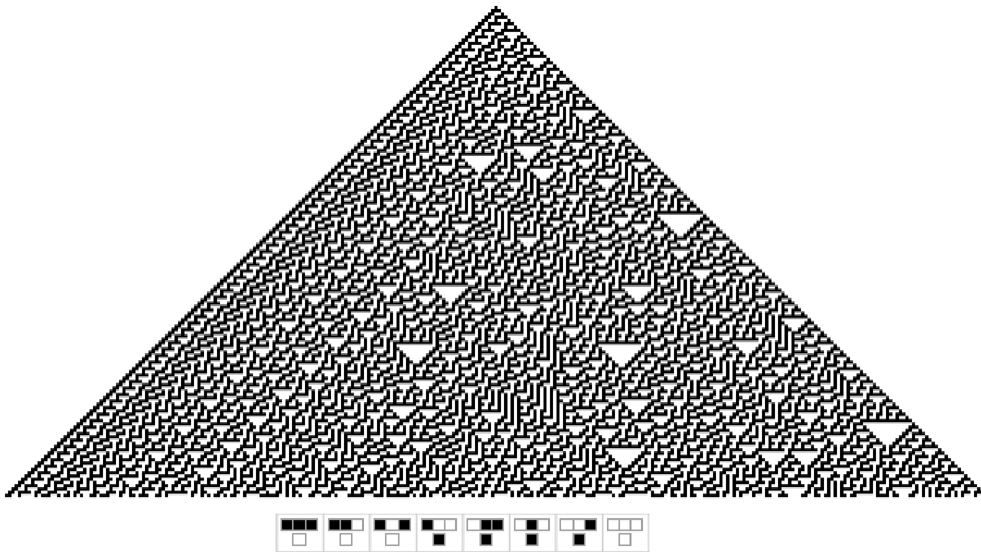


Figure 2.1: Wolfram's Rule 30 [22].

The justification he makes for his claim is that when one looks at what the cellular automata is doing, one does not comprehend how it really works. Just like nature, one might be able to analyze some of its parts and be impressed with how smart they are, but it would be extremely hard to understand the full picture. Wolfram claims by the laws of computational irreducibility that we had to do an irreducible number of computations to figure out precisely what the generated patterns look like. In other words, there are no shortcuts and the only viable method is to do a full-sized simulation. Therefore, it is radically different from exact science where we normally predict the behaviors of the models by solving mathematical equations. Hence, it is in vain to even attempt to manipulate or systematically build a cellular automata rule which generates the desired pattern.

Nevertheless, despite the unpredictable nature of cellular automata, there have been some approaches to find interesting two-dimensional cellular automata rules containing spaceships. The most naïve is to repeatedly create random neighborhood rules and inspect if it generates an interesting result. This method only works for lower dimensional cellular automata (specifically one-dimension), whose parameter search space is relatively small. In this case, we can brute force all the possible sets of rules

and manually inspect which ones are interesting after the generations are saved. However, when it comes to higher-dimensional cellular automata, even a two-dimensional one, the parameter search space becomes gigantic, and most rules would not produce spaceships. Hence, this method becomes inefficient and random as the experiment turns into a pure matter of luck. Another approach is to make slight modifications to known interesting rules, such as John Conway's the Game of Life, to generate similar or refined patterns. This specifically involves modifying one or two of the parameters while leaving the rest unchanged. However, this practice usually leads to a radically different result than the original because the interesting rules are not necessarily clustered together in the parameter search space. Consequently, this method turns out to be not much more promising than the first. Nevertheless, despite their randomness and ineffectiveness, the two methods mentioned above are commonly used to find rules containing spaceships.

A more systematic method to find spaceships is introduced by Andrew Wuensche in Collision-Based Computing [3]. He proposed that this could be achieved by measuring the variance of input entropy over time. The method also allows automatic "filtering" of cellular automata space-time patterns to show up spaceships and related emergent configurations more clearly. He claimed that cellular automata dynamics are shown to exhibit some approximate correlations with global measures on convergence in attractor basins, characterized by the distribution of in-degree sizes in their branching structure, and to the rule parameter Z.

There are also some computational methods to determine the type of the generated cellular automata. For example, Christopher Langton created a cellular automata lambda value that is computed based on the number of cells that have been born at that time step and dividing it by the total number of cellular automata cells [12]. This formula generates a decimal value between 0 and 1. The endpoints of the interval, 0 and 1, correspond to the static patterns and explosive growth respectively. Based on his classification, a lambda value within 0.1 and 0.15 indicates an interesting rule

that requires further investigation. However, the most well-known classification of cellular automata is introduced by Stephen Wolfram, which consists of four different classes: automata in which patterns stabilize into homogeneity, automata in which patterns evolve into mostly stable or oscillating structures, automata in which patterns evolve into chaos, and automata in which patterns become extremely complex [21]. Based on his classification, the fourth class is potentially computational universal and worth investigating. But neither Langton nor Wolfram established a connection between the classifications and the rules themselves.

None of these described methods have been proven to be reliable as they usually find noise or stasis. Therefore, detecting spaceships in two-dimensional outer totalistic cellular automata is an unsolved problem and this report will introduce the potential of neural networks to detect interesting rules. The main idea is that we will build machine learners, which are much more computationally capable than humans and other programs, to help determine whether the rules would be interesting. If we think of the parameter space as an ocean, then an interesting rule is like a particular depth and the spaceships are the fish at that depth. Instead of randomly choosing a depth and inspect whether fish can survive, we can quickly scan a vast volume of the sea and inspect those depths that have fish swimming in it. Given sufficient computers and memory, we have the potential to detect all the interesting rules containing spaceships.

Interestingly, Wolfram also described an uncanny systematic resemblance between neural networks and cellular automata in his book [1]. The parameters in neural networks are never explicitly set or engineered but are generated automatically. Similar things happen with cellular automata as the patterns are never artificially constructed. What differentiates between the two is that in neural networks there are learning processes, where the weights are improving according to rules of linear algebra and calculus. However, in cellular automata, the parameters of the rules are not necessarily improving, and one is forced to enumerate all possibilities.

3 Methodology

We first collected the two-dimensional cellular automata data that the machine learner could use for training, validation, and testing. This entailed designing and implementing a data-collection pipeline from scratch to generate a sequence of raw frames for each of the patterns. Then we tested several models and analyzed for the best results. We trained the data with different models including RNN and CNN, and performed hyperparameter tuning, image feature extraction, and entropy analysis.

3.1 Dataset Generation

The foremost step was to program the cellular automata evolution algorithm, which computes the states of each cell in the next generation based on the rules and the current configuration. The logic follows the rules in the "Instant birth, gradual death, no recovery" model: in the next generation, the live cells that satisfy the survival rule and the dead cells that satisfy the born rule will be alive, the live cells that do not satisfy the survival rule will become sick, the cells that are in the dying transitional (sick) states will move one step closer to death, and the remaining dead cells that do not satisfy the born rule will remain dead. The survival rule, born rule, the total number of possible states, and the neighborhood were passed in as parameters. The specific implementation of the algorithm can be found in Section A.1.

We were able to keep track of the evolution of all cells in the grid in each generation using the evolution algorithm. With the help of $CellPyLib$, which is a python package supporting the visualization of two-dimensional, k-color, adjustable neighbor cellular automata, we could save the evolution as a sequence of frames. However, we still needed to collect known rules so that we could pass them into the algorithm and train our machine learner on the generated sequence of frames later. To obtain boring rules, we manually went through random examples and collected those that died out immediately, generated static noise or boring non-spaceship patterns. Enu-

meration proved to be quite effective because most rules fall under the category of boring, and we easily collected 105 boring rules using this method. Interesting rules, on the other hand, are more rare, and thus were harder to find. Hence, we borrowed existing examples provided in Cellular Automata Rules Lexicon and recorded those with spaceships. Eventually, we successfully collected 35 rules that can be subjectively classified as interesting. The set of interesting rules we included in our dataset is described in Section A.2.

Because of the limited number of rules that we classified, we decided to apply data augmentation to increase the size of the dataset to a reasonable size exceeding 1,000. Since we also wanted to maintain a 50/50 split of the two patterns for training purposes, we reused each boring and interesting rule 10 and 30 times with different random initial configurations. Consequently, we generated a total of 1050 boring and 1050 interesting patterns. For each of the patterns, we recorded 140 consecutive generations as grayscale frames using $CellPyLib$, which was usually more than enough for the patterns to stabilize. The living and dead cells were in black and white respectively. The remaining sick cells were assigned with a grayscale color in between depending on their specific state. Figure 3.1 shows an example of a generated frame. This happened to be the most time-consuming step as it took roughly five hours to go through the lexicons and record the interesting ones, and another 12 hours to generate the frames.

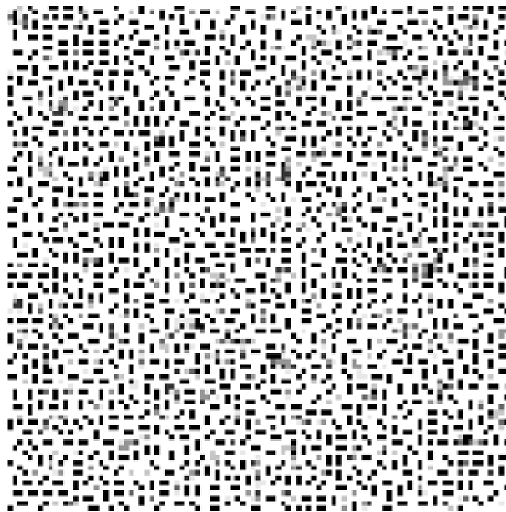


Figure 3.1: Frame 107 of a boring rule "1,5,8/2/7" with a random initial configuration.

3.2 Sequence Training with RNN

After building the dataset from scratch, we were ready to start testing different machine learners and analyze the results. Unlike other image classification tasks like distinguishing between cats and dogs, our classification task contains extra temporal information. Specifically, the sequence in which the frames were generated represents the evolution of cellular automata with respect to time, so the frames could not be processed in random order. It is very similar to a video classification task from this perspective. Hence our first approach was to use a Recurrent Neural Network (RNN), as an RNN can effectively connect information obtained from previous frames to the present frame. However, one potential problem was that RNNs only work if the gap between the relevant information and the place it is needed is small. In our case, we might need to include many consecutive frames because spaceships sometimes span across a large number of time steps. Therefore, we decided to use a Long Short-term Memory network (LSTM), which is a special kind of RNN capable of learning long-term dependencies and thus a perfect fit for our sequence classification task [5]. Since the initial configuration was totally random, we believed under most circumstances the starting generations of the cellular automata were highly randomized and would

not reflect the eventual pattern accurately. Therefore, we decided to start training the LSTM at the 80th frame, where the patterns were reasonably solidified. Each sample consisted of 41 (from 80th to 120th) consecutive frames, and each frame was of size 300 × 300 and has 1 channel as it is grayscale. The 41 selected frames were congregated into a list and the LSTM would process the entire list as one sample. The frame extraction code can be found in Section A.3. We hoped that the machine learner would consider the existence of spaceships as the decisive trait during the training process.

We tested many architectural parameters and structures to create the best model. One failed attempt was stacking a Conv2D layer on top of an LSTM layer. We thought this might work because a Conv2D layer is capable of capturing image features and LSTM can detect temporal correlations across the frames. However, the results were suboptimal and the correlation between time and space features was not captured properly by stacking the layers. Therefore, we eventually used a convolutional LSTM network, which differentiates itself from a normal LSTM in the way that it has convolutional structures in both the input-to-state and state-to-state transitions and research has shown that a ConvLSTM2D layer is better at capturing spatiotemporal information [4]. Our results did improve significantly after we made this change. Furthermore, we also tried different filter sizes, dropout rates, kernel sizes, and activations. Eventually, we used the structure described in Table 3.1 for our machine learner. The amount of time it took to train the model depended greatly on the number of training epochs and the size of the dataset, but it would not exceed an hour in our case.

ConvLSTM2D	64 filter output space, 3×3 filters, 15% dropout
max pooling	(2, 2) pooling kernels, 15% dropout
dense layer	256 nodes, ReLU activation, 15% dropout
dense layer	64 nodes, ReLU activation, 15% dropout
dense layer	2 nodes, softmax activation

Table 3.1: Structure of the RNN

The machine learner was able to achieve 93% training accuracy and 91% testing ac-

curacy on the testing set with 10% interesting data. The test recall is 98%, indicating the majority of interesting configuration has been correctly labeled as such. The high accuracy score indicates the success of the machine learner.

3.3 Data Preprocessing, Feature Extraction, Training with CNN

In the previous approach, we used RNN, specifically LSTM to train the models, which successfully processed the underlying temporal relationship of the frames. An alternative method we tried was to use a Convolutional Neural Network (CNN) by treating the frames as a typical image classification task. However, in this case, we needed to reconfigure our dataset of sequences of frames into trainable images beforehand. We also had to ensure that these reconfigured images in some way preserved the temporal information. To satisfy these requirements, we decided to stitch the images in a predetermined order into a square grid as shown in Figure 3.2. We hoped that the machine learner would be able to recognize the underlying relationship between the frames.

Figure 3.2: Example of sequence of frames in the stitched image if nine of them are included.

Another challenge we faced was to decide which frames should be included in the stitched image. To increase the algorithm's robustness and to control better for interesting configurations that have some seemingly uninteresting frames interspersed throughout their evolutions, we created two additional parameters: the starting frame and the number of frames to be included. We tested many possible numerical values of the two parameters to discover the best combination. Figure 3.3 shows two exam-

ples of the stitched images, one represents a boring rule while the other represents an interesting rule.

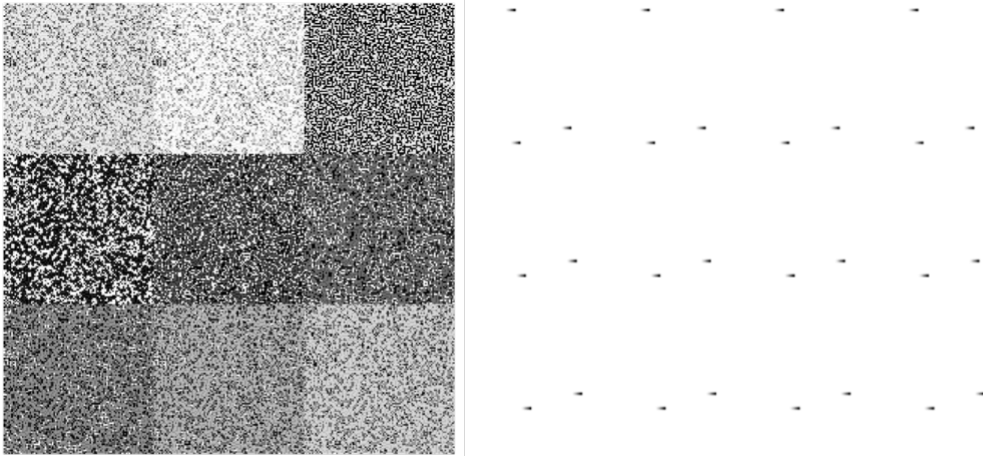


Figure 3.3: Examples of stitched up images. (Left) Nine frames are stitched together, which are generated by a boring rule "6/0,5,6,9/8". All frames are noise. (Right) Sixteen frames are stitched together, which are generated interesting rule "2/2/8". By comparing the frames in the top left and the bottom right corner, we could see the pattern translating to the right, which indicates a spaceship.

Since stitching frames is an uncanonical method of classifying cellular automata patterns, we wanted to measure the learnability of the stitched images using Brainome.ai [9] before we built the model. Because Brainome.ai accepts labeled data in CSV format, we needed to do feature extraction to the stitched images as the final preprocessing step. We first used a pre-trained NASNet-Large Model, which is a CNN that is trained on more than a million images from the ImageNet database [10]. For each of the stitched images, the model returned 1000 selected features. We then fed the data into Brainome.ai and obtained corresponding information about Decision Trees and Neural Networks. The expected generalization using Decision Tree is 2.05 bits/bit and using a Neural Network is 0.19 bits/bit. The decision tree has 1,026 parameters, and the estimated memory equivalent capacity for neural networks is 11,034 parameters. This overwhelming memory equivalent capacity indicated that the neural network would be extremely overfitting, which means that the features extracted

were barely learnable by the neural network. This poor result was reasonable in retrospect because the NASNet-Large model is specifically used for classifying and extracting features from images of common-life objects, and cellular automata patterns is not one of its targets.

As our previous feature extraction method with NASNet-Large model was unsuccessful, we decided to directly use the pixels of the stitched images as features. We fed the data into Brainome.ai and learned that the estimated memory equivalent capacity for neural networks is 3217 parameters, which was much better than the previous result even though the risk of overfitting persisted. Nevertheless, this gave us sufficient confidence to proceed with model training.

As raw data, these images were quite large given the RAM allocation of 12 Gigabytes by Google Colab [11]. Running the notebook tended to crash the kernel so we settled for lower resolution and downsampled the pixel images to 300×300. This tradeoff allowed us to manipulate and do machine learning on the data without too much computational expense. As a final preprocessing step, the [0, 255] valued matrices representing the images were normalized using simple division to [0, 1]. This improved performance greatly in practice. Many of the model architectures we tried produced sub-baseline results before this step. The next part of the optimization process was a question of model architecture and hyperparameter tuning. We tried many things to create the best model, which included altering the convolutional filter size, adding batch normalization, adding dense layers at the output, pooling the kernel size, modifying the type of pooling, and tweaking the dropout.

We found that the greatest improvements happened after adding dropout and batch normalization. There was also a significant increase in accuracy after increasing the convolutional filter size of the first convolutional layer to 5×5 from 3×3. We believed this is because 3×3 is too small to capture much of the complexity of the interesting configurations. Given a 3×3 window, many of the interesting shapes looked like noise.

We tried many things that did not work in addition to those that did. Increasing the pooling kernel size, using average pooling instead of max pooling, increasing the number of filters in the convolutional layers (from 64 in each), and increasing the second convolutional layer's filter size from 3×3 to 5×5, all resulted in worse performance by the validation accuracy metric. We found that increasing the epochs past 30 resulted in overfitting. Eventually, we used the architecture described in Table 3.2. The amount of time it took to train the model depended greatly on the number of training epochs and the size of the dataset, but it would not exceed an hour in our case.

The machine learner was able to achieve 93.44% training accuracy and 84.12% testing accuracy on the testing set with 10% interesting data. The test recall is 100%, indicating every interesting configuration has been correctly labeled as such.

Conv2D	64 filter output space, 5×5 filters, ReLU activation. Batch normalization prior to ReLU
max pooling	(2, 2) pooling kernels, 15% dropout
Conv2D	64 filter output space, 3×3 filters, ReLU activation. Batch normalization prior to ReLU
max pooling	(2, 2) pooling kernels, 15% dropout
dense layer	64 nodes, ReLU activation, 15% dropout
dense layer	10 nodes, ReLU activation, 15% dropout
output layer	1 node, sigmoid activation

Table 3.2: Structure of the CNN

3.4 Entropy Analysis

Due to the varying degrees of information density in cellular automata patterns, entropy is an adequate measure to use since there is likely a correlation between the label (boring and interesting) and the degree of randomness in the images created by the

cellular automata. There are many existing research on this topic and the common consensus is that the problem of computing or even approximating the topological entropy of a given cellular automata is algorithmically undecidable [7, 8]. Therefore, we decided to try something that has not yet been attempted before. Namely, we computed the cross-entropy values of the stitched images generated in the preprocessing step of Section 3.3. In this case, boredom can be understood as either static or complete noise, which correspond to extremely low and high entropy values.

We iterated through all the stitched frames and computed their entropies using the cross-entropy algorithm, then plotted the entropy values of the boring and interesting images in Figure 3.4.

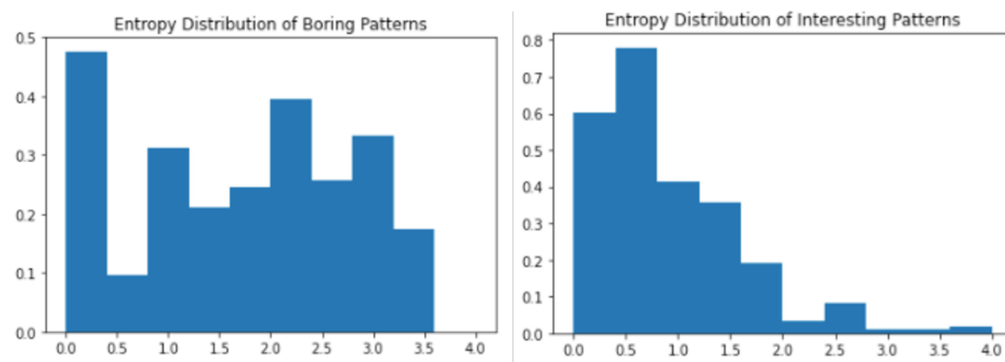


Figure 3.4: Entropy distribution of boring (left) and interesting (right) patterns.

Boring images had entropy values that spread roughly evenly from 0.0 to 3.5, with a small gap between 0.5 and 0.75. Many had entropy values close to 0.0, which is reasonable because they would likely correspond to patterns that die out. On the other hand, interesting images had entropy values concentrated in the 0.0 to 2.0 range, especially between 0.5 and 0.75. This intuitively makes sense because interesting images had less noise and entropy compared to boring images on average. It should be noted the minimum entropy for boring images was 0.0 while the minimum entropy for interesting images was 0.0318. This is because frames that had no live cells were always labeled as boring. The entropy values suggest adding features identifying if the image entropy is above 2.0 or equals to exactly 0 may be beneficial for the model accuracy.

4 Spaceship Discoveries

After we finished training the machine learners, we put them into practice and used them to classify patterns and find spaceships in those that are interesting. Since cellular automata with two possible states, like the Game of Life, have already been widely explored, we focused mainly on those with three states. We ran the machine learner on the sequences of frames generated from random combinations of survival and born rules, and then manually inspected the few rules which the learner classified as interesting. Overall this was a very time-consuming process since for each new rule, we had to generate a sequence of frames so that the machine learner can have something to train on. However, we believed that the situation could be ameliorated in the future thanks to the process being entirely parallelizable. More relavant information is discussed in Section 5. The evolution of the spaceships in this section will be provided as a sequence of figures. The numbers give the generations and the exact movement of each is depicted by its shifting position in the enclosing grids. Code used to generate the figures can be found in Sections A.11 and A.12.

4.1 Spaceships in three-state cellular automata

Our machine learner has found several new interesting rules with three states (the dead state, the live state, and one sick state) that have not been previously discovered, which are "4,6/2/3", "4/2,4/3", "4,6/2,4/3", "2,4,6/2,4/3", "4/2,5/3", "3,6/2,6/3", and "5,6/2,6/3". The common trait of these newly discovered rules is that dead cells will be born with two alive cells in their neighborhoods. These rules all generate the same family of spaceships, where the members are all led by a two-by-two spaceship and followed with a distinct *tagalong*, where tagalong is defined as a pattern that is not a spaceship itself but can be attached to the back of a spaceship to form a larger spaceship [17]. We decided to call this two-by-two leading structure the "leading block" (top left corner in Figure 4.1). The leading block is the smallest found spaceship in

all patterns with three possible states. It has a period of one and a speed of c. Since all members in the family are led by the leading block, they all have a uniform speed of c. However, the members can have different periods. The one-period members have zero or more one-by-two blocks (we named it the supplemental block), which are the smallest possible tagalongs, attached to either side of the leading block. We can enumerate the number of one-period spaceships with at most two supplemental blocks. There is one member with no supplemental block attached, namely the leading block, two members with one, and ten members with two. These spaceships are shown in Figure 4.1. There are infinitely many one-period members in the family because any arbitrary number of supplemental blocks can be attached.

However, one caveat is that there are some patterns, like the one shown in 4.2, that have the supplemental blocks attached to the leading block but are not actually spaceships. This means that the tagalongs are very delicate and the slightest difference in their structure can lead to a massive change in the eventual outcome. Most patterns that have almost the same tagalong as one of the basic forms with only a few different cells will not turn out to be a spaceship. Thus, it is very hard to artificially engineer a spaceship and it proves Wolfram's philosophy that we should not try to build anything at all.

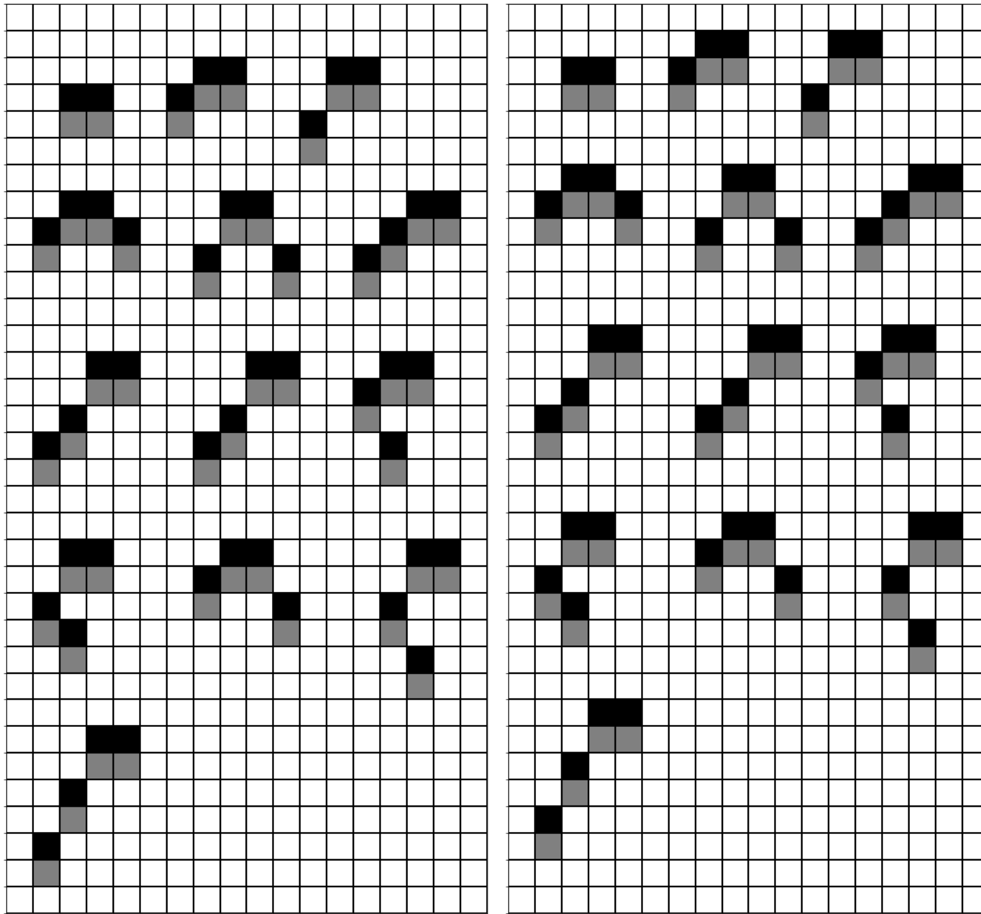


Figure 4.1: All one-period members with zero, one, or two supplemental blocks of the spaceship family that appears in rules “4,6/2/3”, “4,6/2,4/3”, “2,4,6/2,4/3”, “3,6/2,6/3”, “5,6/2,6/3”. Code is provided in A.12.5.

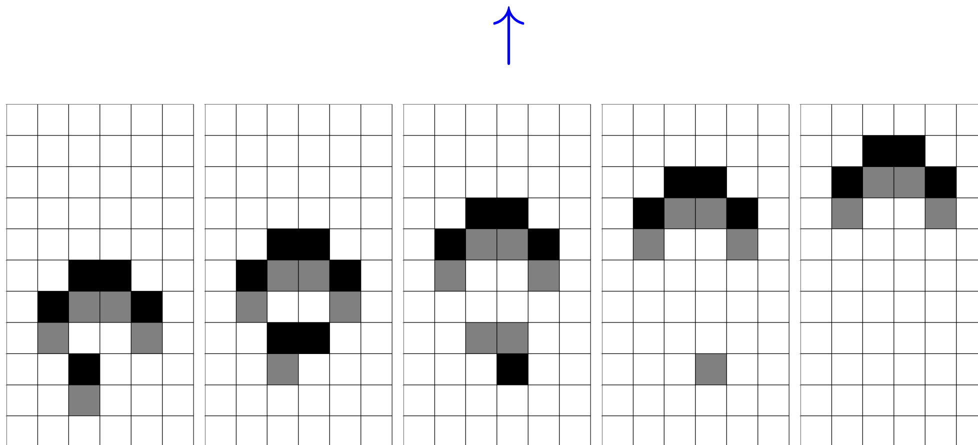


Figure 4.2: A pattern that is not a spaceship in rules "4,6/2/3", "2,4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3" despite consisting of the leading block and three supplemental blocks. However, it does transform into another two-period spaceship in four steps. Code is provided in A.12.6.

We also found many family members with a period of two. Their structures are more complicated since their tagalongs are no longer entirely made of the supplemental blocks. They emit one unit of vanishing "exhaust" when moving across the grid, whereas the rest of their body remains unchanged. We have discovered five most basic two-period spaceships that appear in rules "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", and "5,6/2,6/3" in Figure 4.3, each with a distinct tagalong. For simplicity, they will be referred to as "two-period spaceship A, B, C, D, and E" in the rest of the report. Furthermore, we have also observed some two-period spaceships that have the exact same tagalongs as the five basic forms. For example, in Figure 4.5 is another spaceship that looks virtually the same as the "two-period spaceship E" and only differs in the way that its tagalong is relatively moved towards the right by one unit. We have already seen such room for diversity in the one-period members. Additionally, like the one-period members, there are also infinitely many two-period spaceships in the family. Their tagalongs can get arbitrarily large as they can have a connecting bridge consisting of any arbitrary number of supplemental blocks between the basic tagalong and the two-by-two leading block. However, all of these tagalongs are essentially extensions of one of the five most basic forms in Figure 4.4. This means that

their tagalongs can be reduced by stripping away one or more supplemental blocks. Figure 4.6 depicts some possible extensions of the basic five two-period spaceships.

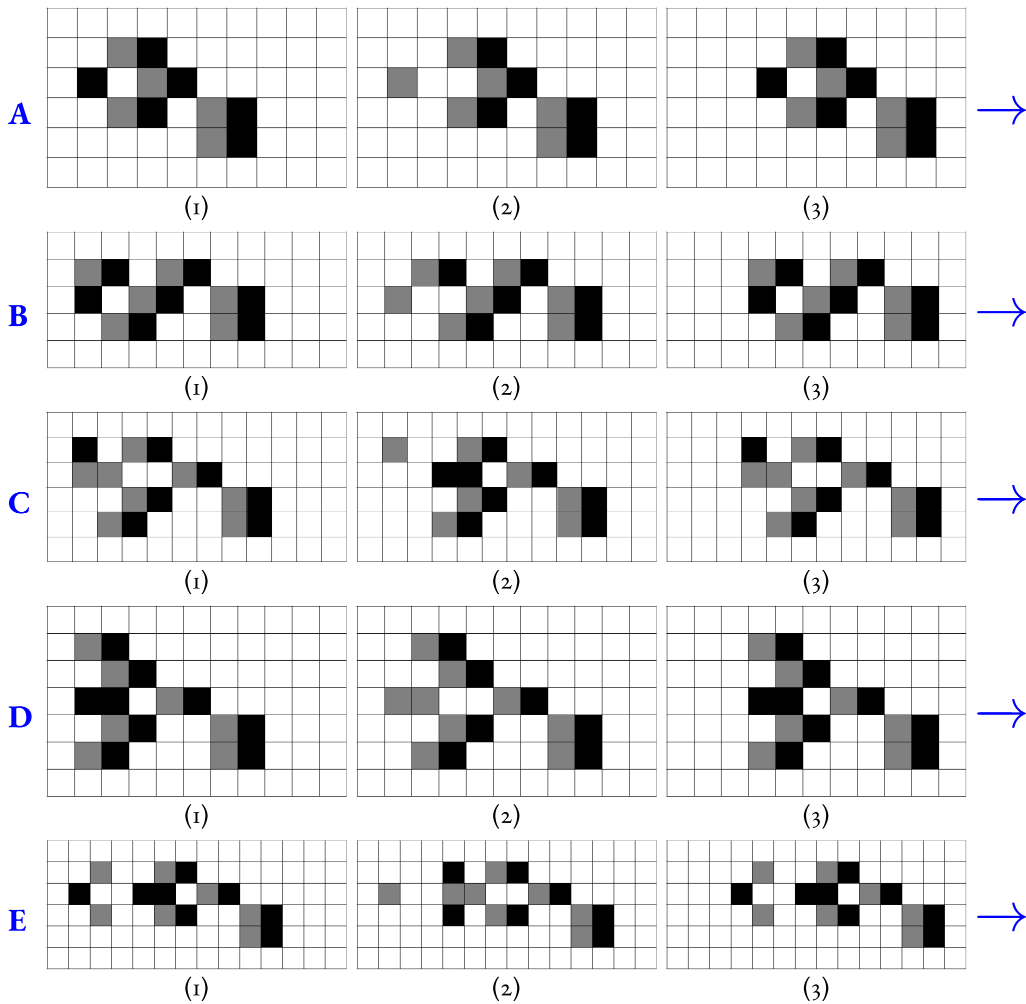


Figure 4.3: The five basic two-period members of the spaceship family that exist for rules "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they are referred to as "two-period spaceship A, B, C, D, and E" respectively. Code is provided in A.12.7, A.12.8, A.12.9, A.12.10, and A.12.11.

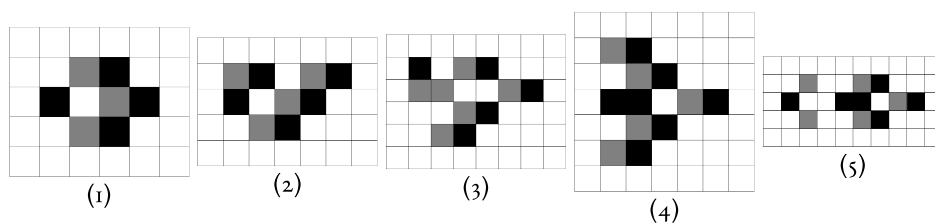


Figure 4.4: The five basic tagalongs of two-period spaceships.

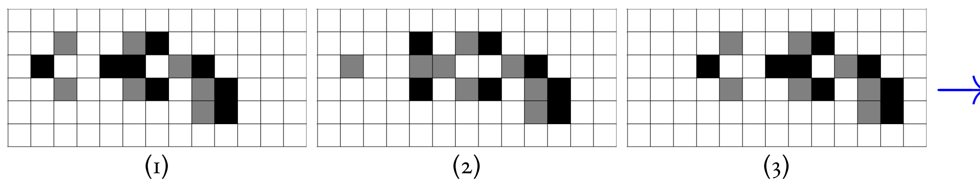


Figure 4.5: The spaceship that is almost the same as "two-period spaceship E" except its tagalong is moved relatively towards right by one unit. Code is provided in A.12.12.

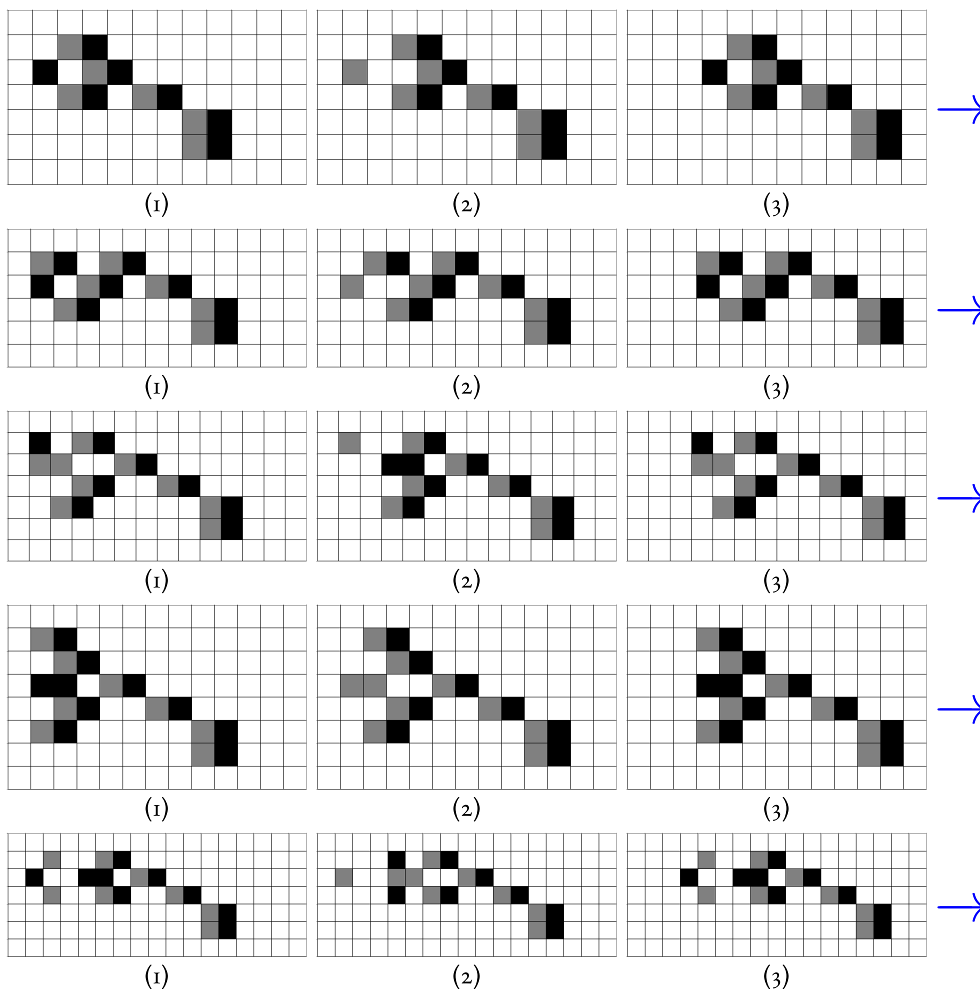


Figure 4.6: Examples of extended two-period members of the spaceship family that exist for rules "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". Their tagalongs are the same as the five basic forms in Figure 4.4 except there is an additional supplemental block connecting them to the leading block. The leading block is like the tractor towing something behind it and this one simply has a longer tow cable. Other extended members have more connecting supplemental blocks and thus even longer tow cables. Code is provided in A.12.13, A.12.14, A.12.15, A.12.16, and A.12.17.

We have also found family members with a period of four. Superficially, they do not look much different from the two-period members except their tagalongs are larger and more complicated. They emit a more noticeable and larger amount of vanishing exhaust when they move across the grid. We identified six basic four-period members shown in Figure 4.8 and 4.9, each with a distinct tagalong in Figure 4.7. For simplicity, they will be referred to as "four-period spaceship A, B, C, D, E and F" in the rest of the report. The longest period members we found are the ones with periods of eight in Figure 4.10. We have discovered a total of two such spaceships, which we will name as "eight-period spaceship A" and "eight-period spaceship B". Their tagalongs are depicted in Figure 4.11. Superficially, there is an uncanny resemblance between "eight-period spaceship A" and "four-period spaceship C", and "eight-period spaceship B" and "four-period spaceship A". The two eight-period spaceships both generate a maximum of ten units' exhaust. We believe it is highly likely that there are more with unique tagalongs that are yet undiscovered, as they are more rare and thus much harder to find than the other members in the family. With the same argument we made with the two-period spaceships, there are infinitely many four and eight-period spaceships in the family.

The family members introduced so far have only one tail. However, these spaceships, unlike normal species in real life, have unlimited potential to mutate, combine, and have arbitrarily complex structures. However, some members in the family have found a way to combine a few basic tagalongs to form a larger one. Figure 4.12 shows two Frankenstein spaceships whose tagalong is a combination of the basic forms we have introduced earlier. One combines "two-period spaceship C" and "four-period spaceship A". The other combines "two-period spaceship A" and two "four-period spaceship B"s. Their existence proves that two or more tagalongs can be combined to form a larger tagalong. The period of the resulting Frankenstein spaceship is determined by the longer period of its components. The diversity of the family members is thus beyond imaginable as the tagalongs can get arbitrarily complex and it is impos-

sible to enumerate all possibilities.

The most interesting family members we found are rakes, which are cellular automata that leave behind a trail of non-vanishing debris of a stream of spaceships [18]. Their structures are much more complex than the other family members. We have observed a total of three rakes, one with a period of four and two with eight in Figure 4.13, 4.14, and 4.15. For simplicity, they will be referred to as "rake A, B, and C" in the rest of the report. Their periods are equivalent to the number of steps it takes them to generate a new spaceship. All three rakes generate a stream of one-period spaceships, whose moving directions are not the same as the rakes. Rake A generates a stream of two-by-two leading blocks, which moves in the opposite direction as the rake itself. Rake B generates a stream of one-period members with one supplemental block, which also moves in the opposite direction as the rake itself. Rake C generates a stream of one-period members with two supplemental blocks, which move perpendicular to the rake.

We discovered one special spaceship depicted in Figure 4.16 that does not contain the leading block and hence is not a member of the family. It maintains a total of three live and three sick cells in all generations. Furthermore, the spaceships we have introduced so far either traverse horizontally or vertically. But just like Conway's the Game of Life spaceship, this spaceship moves diagonally across the grid with a period of four. Its speed is also $\frac{c}{4}$, since it takes four generations for a given state to be translated by one cell. We decided to call this the new Life with three states.

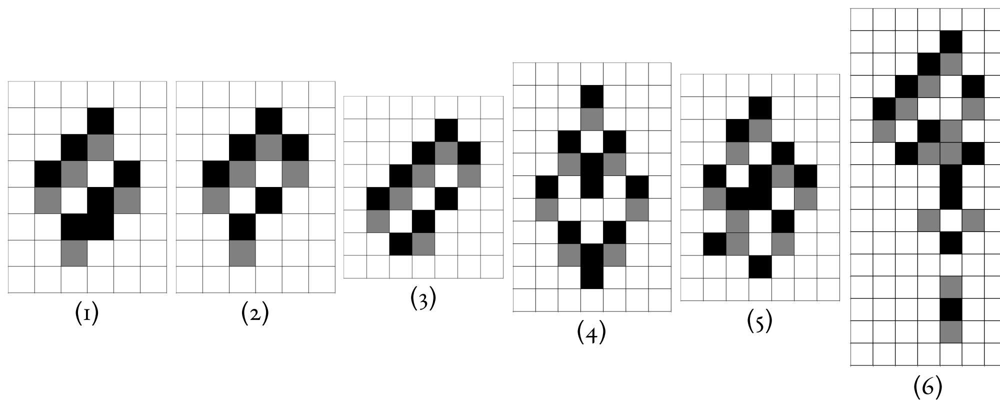


Figure 4.7: The six basic tagalongs of the four-period spaceships.

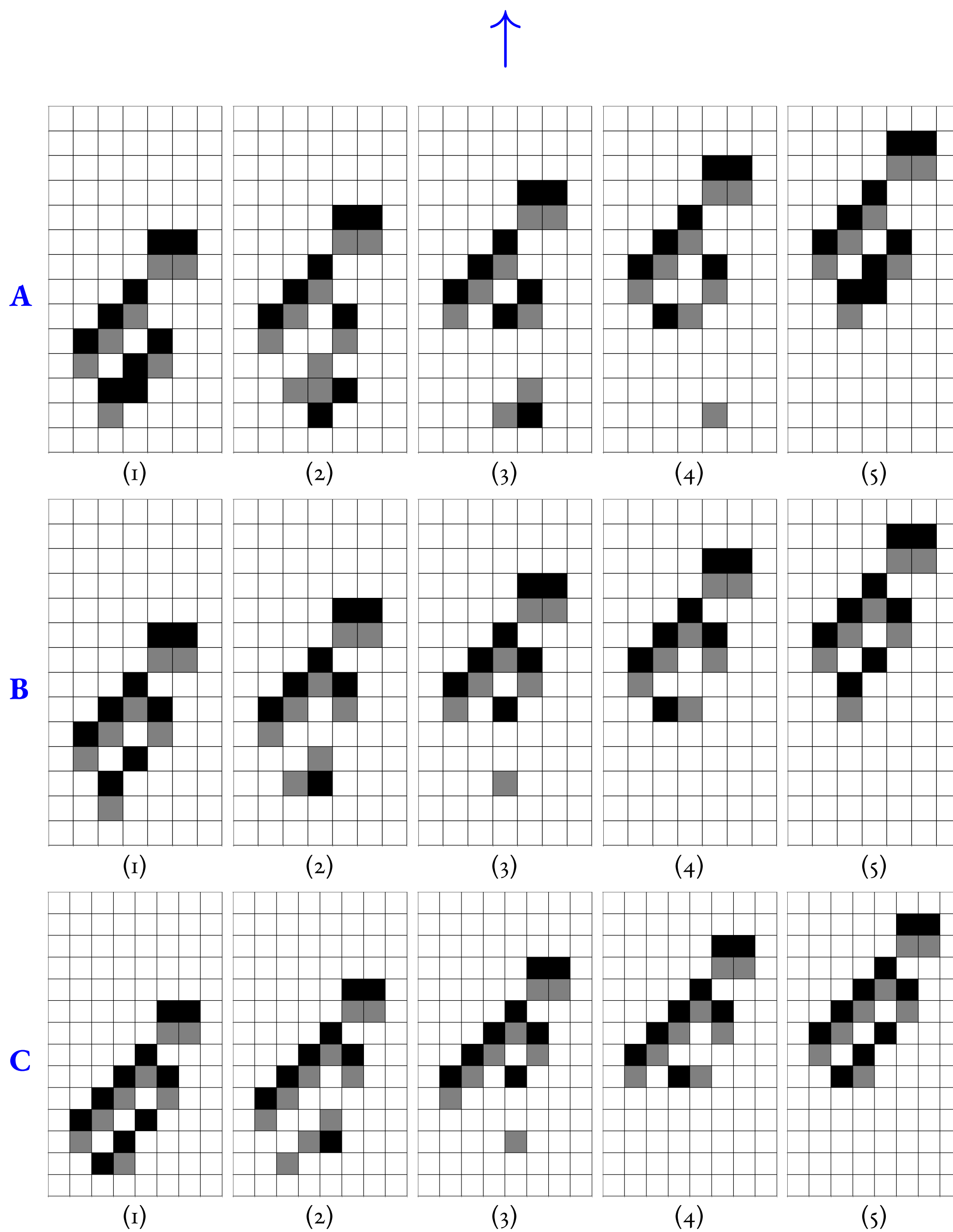


Figure 4.8: Three of the six basic four-period members of the spaceship family. All these spaceships exist for rules "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they are referred to as "four-period spaceship A, B, and C" respectively. Code is provided in A.12.18, A.12.19, A.12.20.

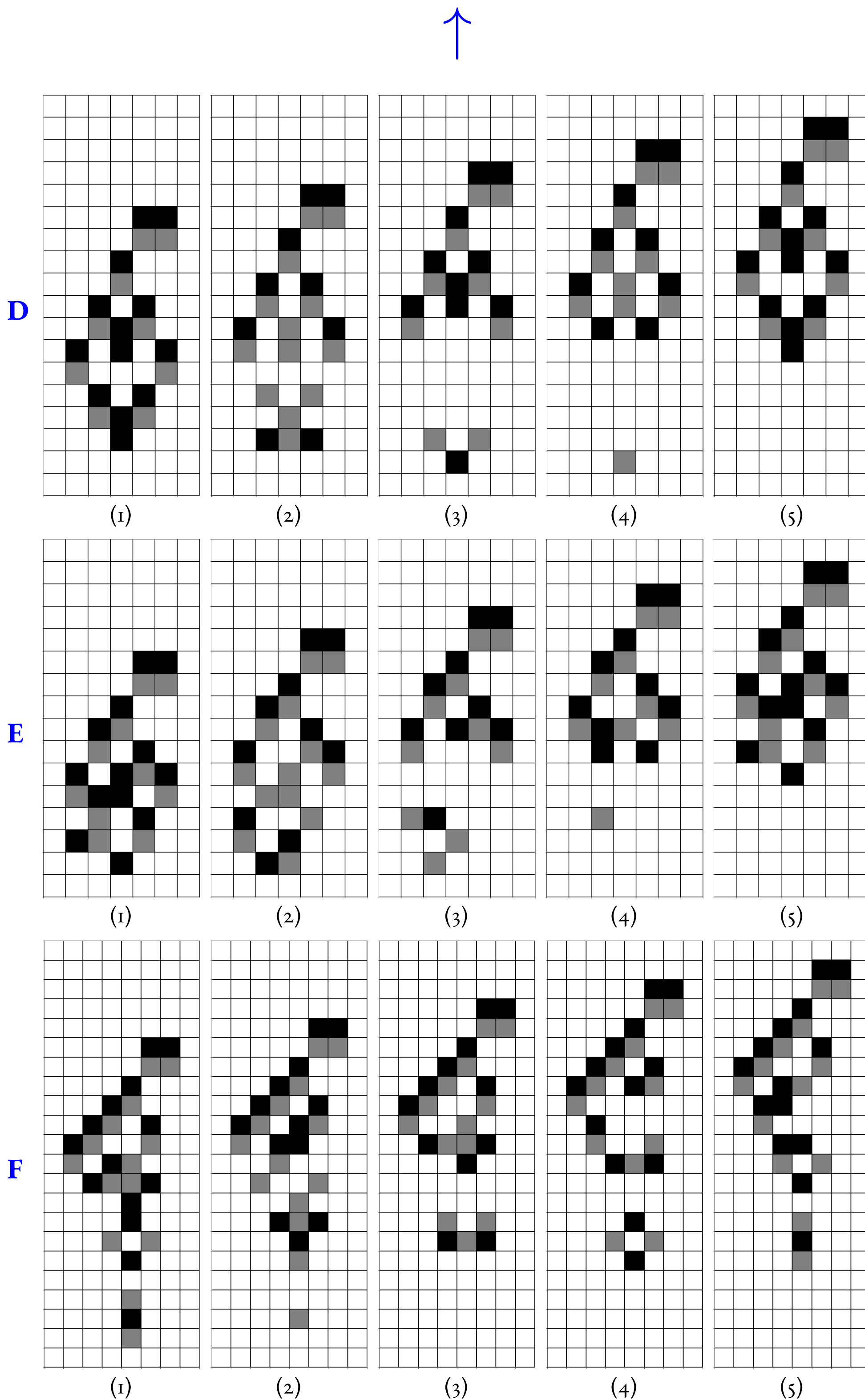


Figure 4.9: The other three of the six basic four-period members of the spaceship family. The first one exists for "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3", the other two only exist for "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they are referred to as "four-period spaceship D, E, and F" respectively. Code is provided in A.12.21, A.12.22, and A.12.23.

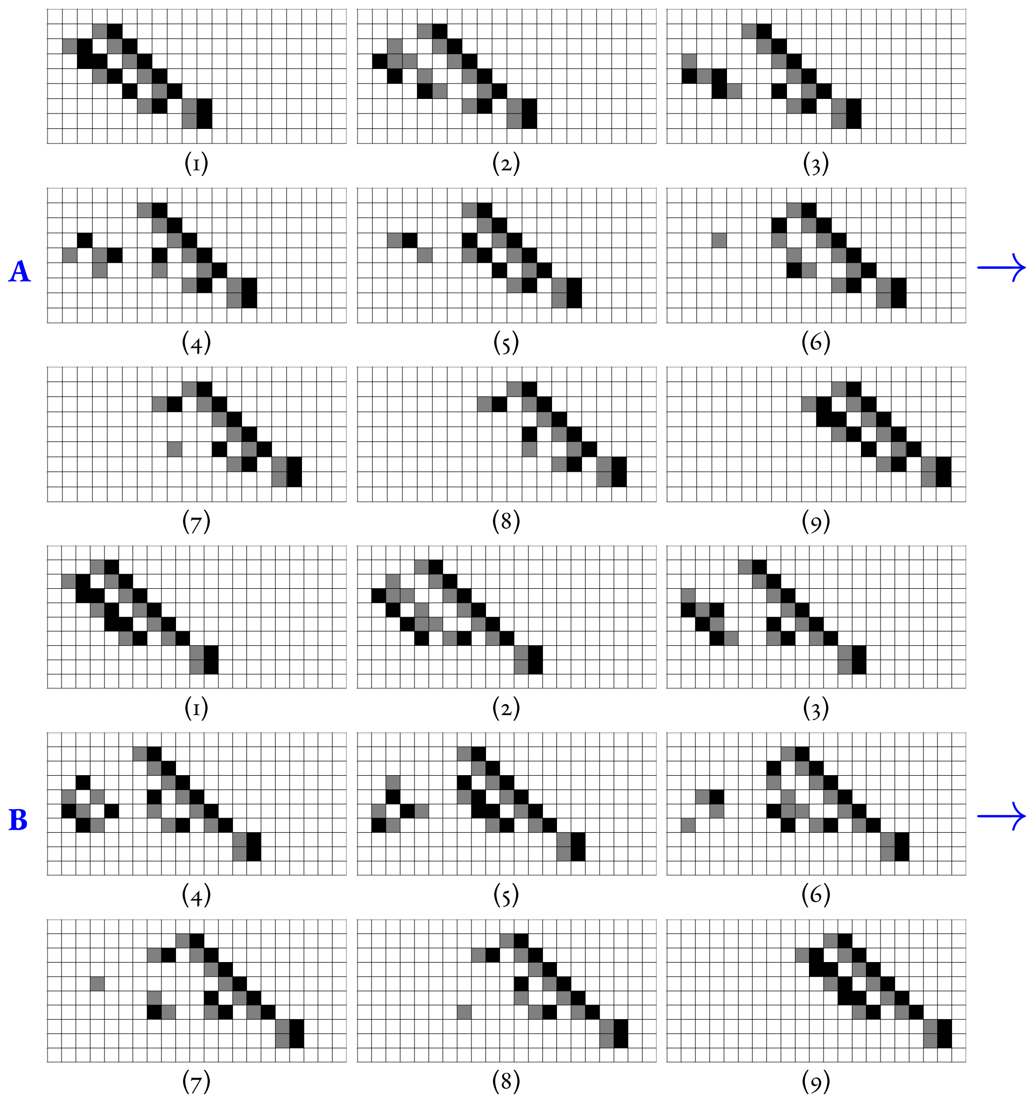


Figure 4.10: Two eight-period members of the spaceship family that exists for rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they will be referred to as "eight-period spaceship A" and "eight-period spaceship B". They resemble "four-period spaceship C" and "four-period spaceship A" respectively. They both emit a maximum of ten units' exhaust. Code is provided in A.12.24, and A.12.25.

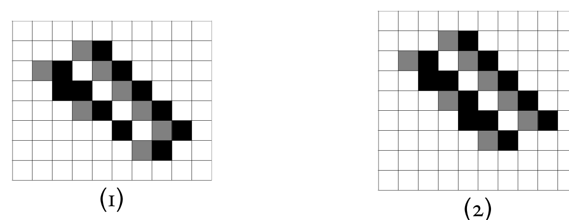


Figure 4.11: The basic tagalongs of the two eight-period spaceships.

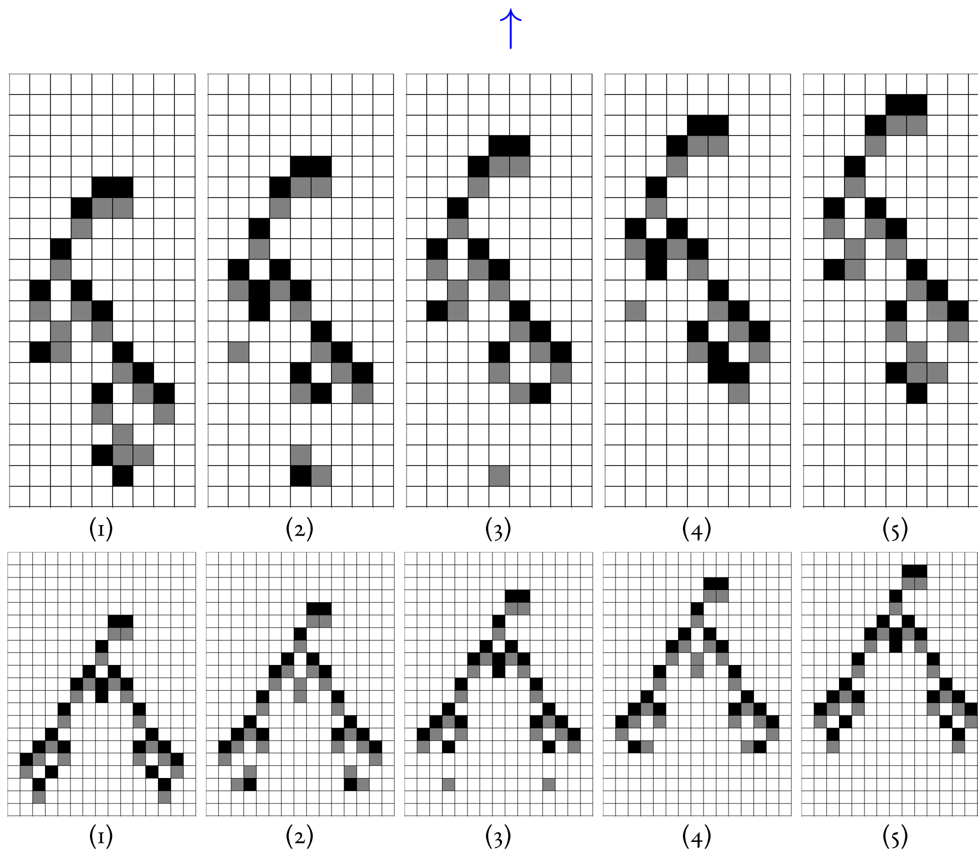


Figure 4.12: Two Frankenstein spaceships with two and three tails that exists for rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". Both have periods of four. The top combines "two-period spaceship C" and "four-period spaceship A". The bottom combines "two-period spaceship A" and two "four-period spaceship B" s. Code is provided in A.12.26, and A.12.27.

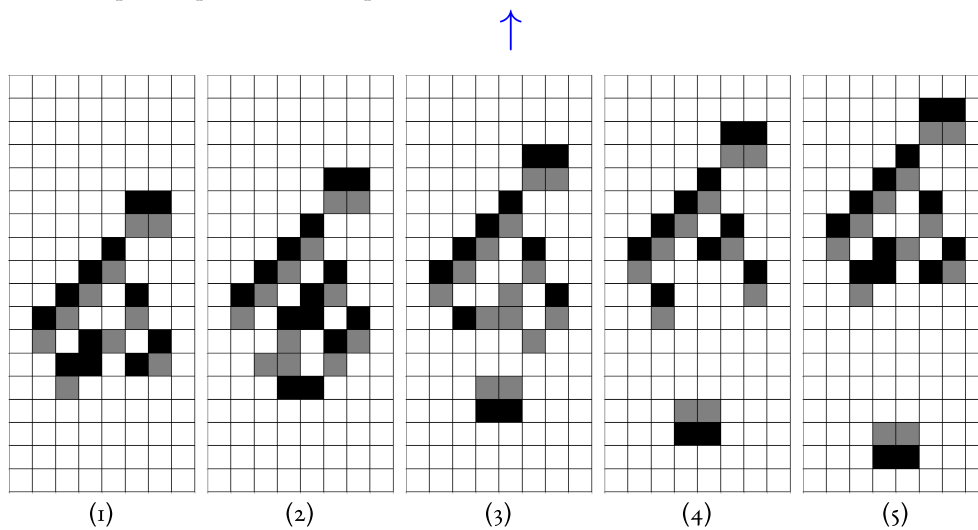


Figure 4.13: A four-period rake that exists for rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, it will be referred to as "rake A". It generates a leading block every four steps. Code is provided in A.12.28.

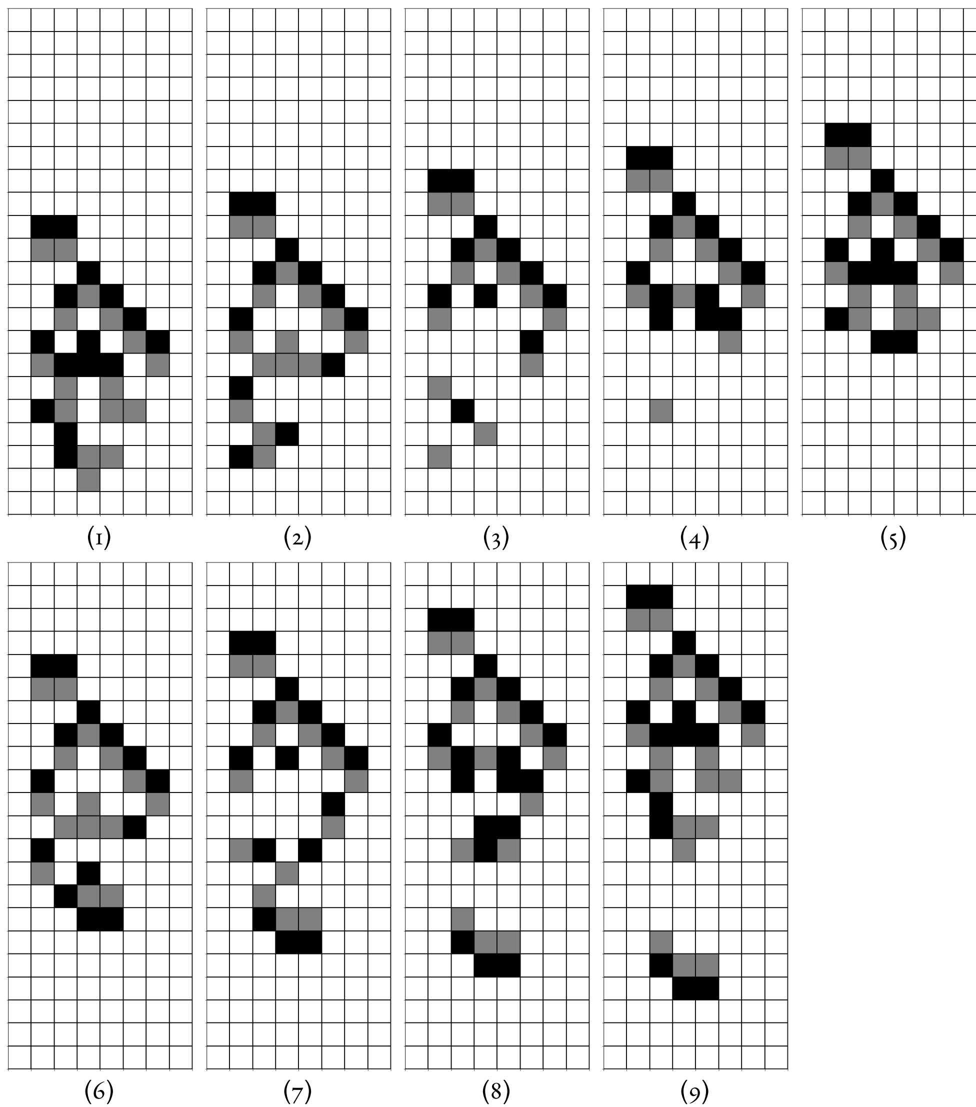


Figure 4.14: An eight-period rake that exists for rules "6/2/3", "4,6/2/3", "5,6/2,6/3". For simplicity, it will be referred to as "rake B". It generates a one-period family member with one supplemental block every eight steps. Code is provided in A.12.29.

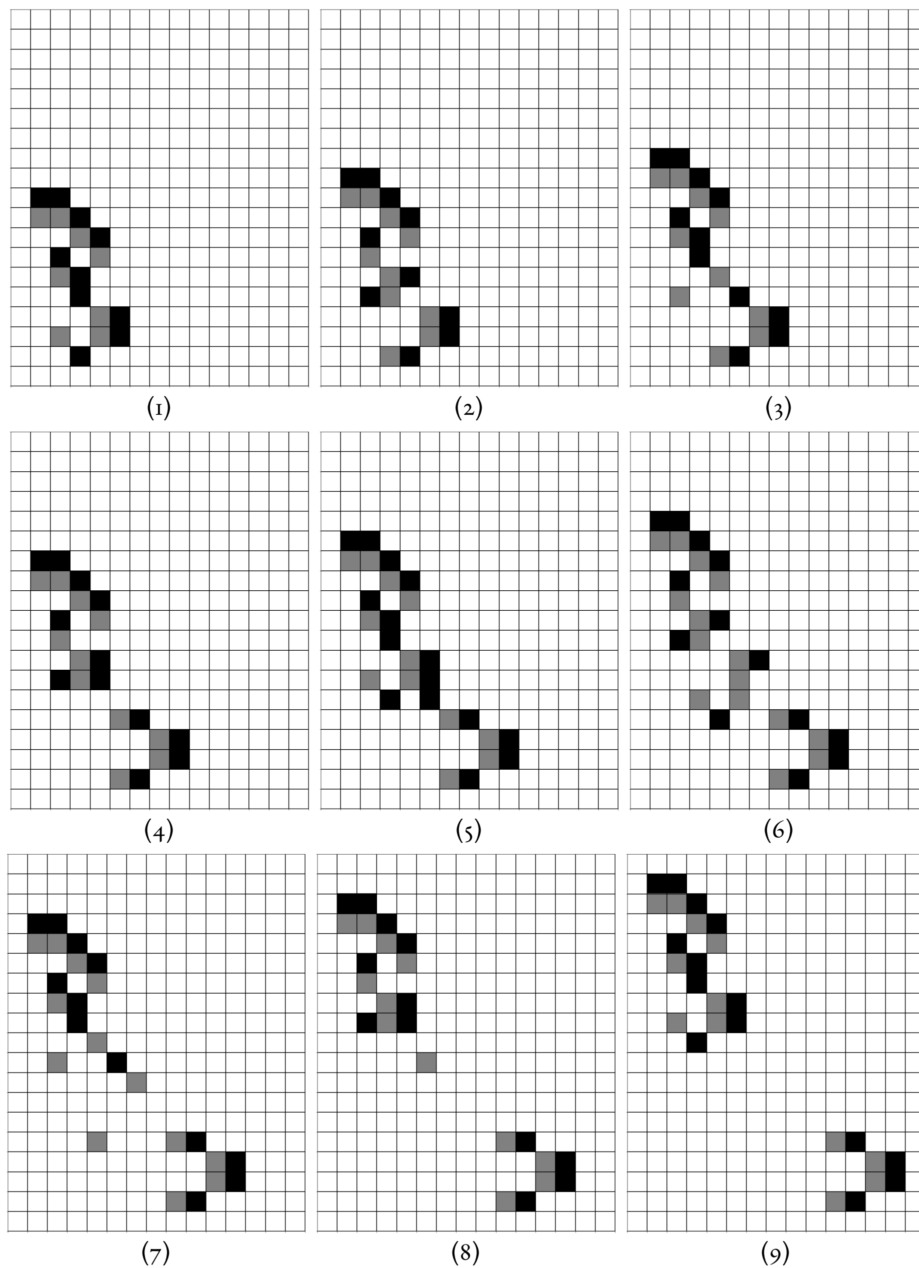


Figure 4.15: An eight-period rake that exists for rule "6/2,4,6/3". For simplicity, it will be referred to as "rake C". It generates a one-period member with two supplemental blocks every eight steps. Code is provided in A.12.30.

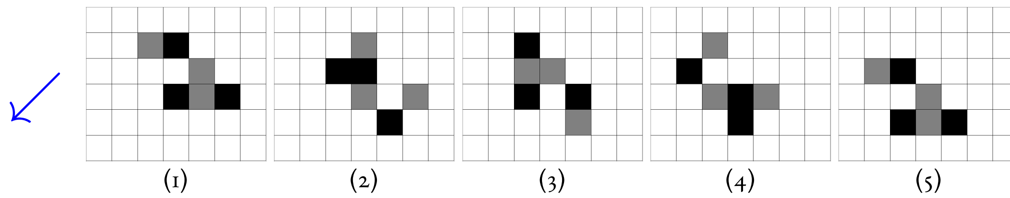


Figure 4.16: The new Life with three states that exists for rules "4,6/2/3", "4,6/2,4/3", "2,4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3", and "3/2,5/3". It moves diagonally with a speed of $\frac{c}{4}$. Code is provided in A.12.31.

4.2 Spaceship collision behaviors in three-state cellular automata

Collisions between spaceships are very common on a two-dimensional grid. The collisions, however, are not of physical nature. When that happens, a chemistry-like reaction will take place between them. Their internal structures become intertwined with each other and temporarily unrecognizable from their previous forms. Eventually, the patterns become clearer, and a new equilibrium has been reached. There are three most frequent cases.

1. Shown in Figure 4.17, the two spaceships are simultaneously destroyed completely in the collision, leaving the grid empty.
2. Shown in Figure 4.18, the collision generates a new spaceship that has a different structure.
3. Shown in Figure 4.19 and 4.20, one of the two spaceships is "murdered" during the process, which means exactly one spaceship survives unscathed after the collision, while the other completely disappears. This is like the survival of the fittest in the animal world and the classic "who would win, a giant squid or a killer whale, a crocodile or a cheetah" questions where only one can survive.

Generally, it is very difficult to predict the outcome when two spaceships collide with each other and there are edge cases that are not included. However, the three listed cases constitute the majority.

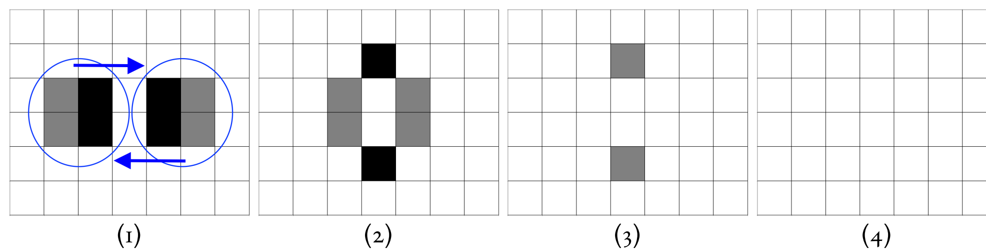


Figure 4.17: Two leading blocks in rule "4,6/2/3" collide with each other head on and are both destroyed in three steps, leaving the grid completely empty. Code is provided in A.12.32.

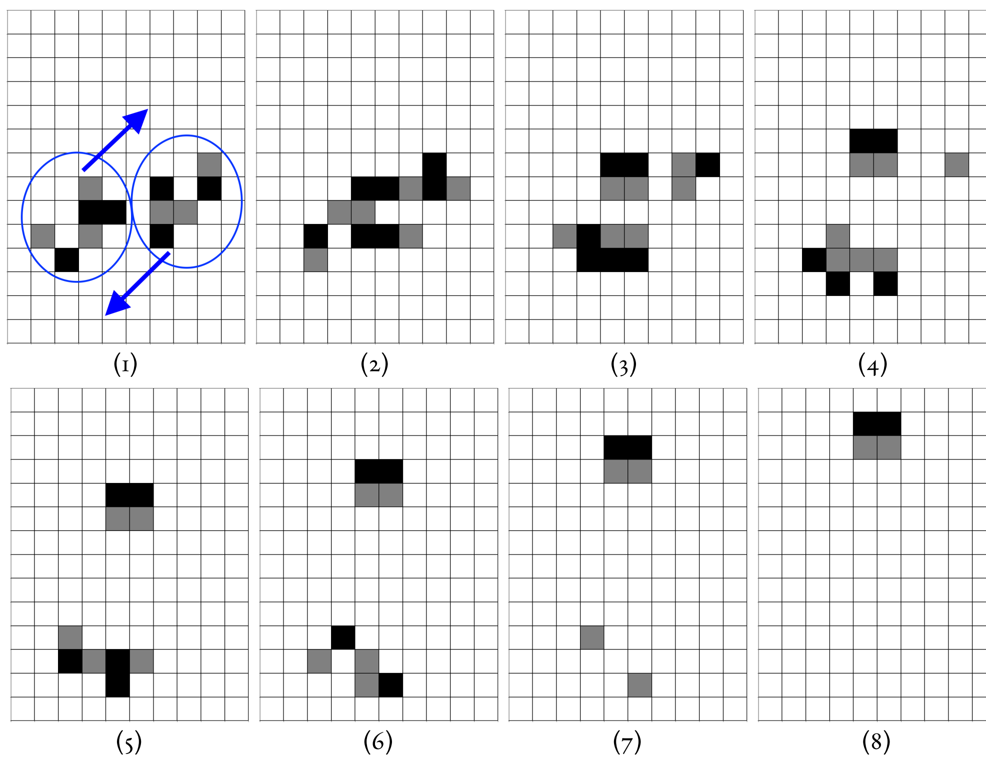


Figure 4.18: Example of a "combined spaceship" that appears in rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". Two new Life's collide and they generate a fundamental block (an interesting demonstration of simplicity coming from complexity). Code is provided in A.12.33.

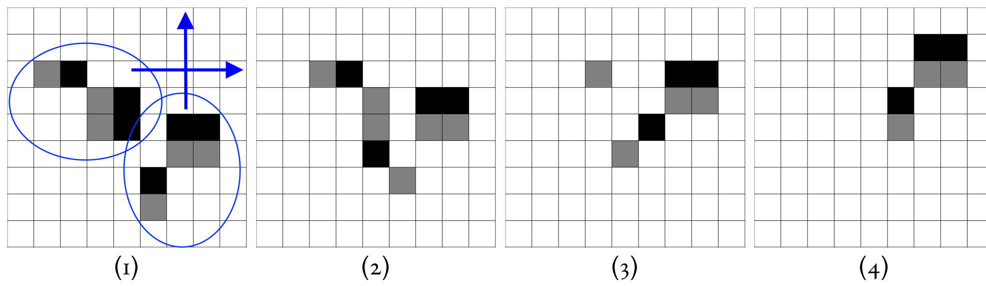


Figure 4.19: Example of a "murdered spaceship" that appears in rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". Two identical one-period members of the spaceship family with one supplemental block collide with each other, and only the bottom one survived the clash. Code is provided in A.12.34.

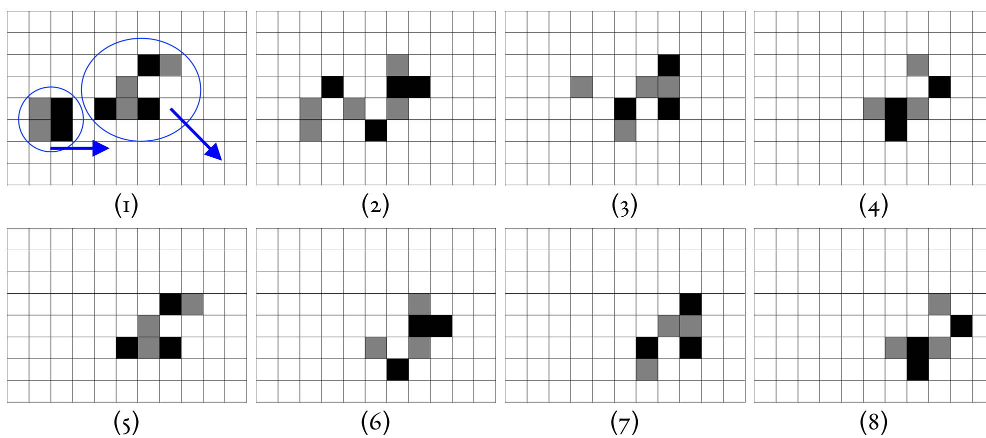


Figure 4.20: Example of a "murdered spaceship" that appears in rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". A fundamental block spaceship collides with the new Life. The fundamental block is murdered in three steps. Code is provided in A.12.35.

4.3 Other interesting discoveries

As the number of states increases, the complexity of the spaceships increases correspondingly. In cellular automata with four possible states, we have found a more visually appealing rake like a spaceship emitting gas in Figure 4.21. The rake has a period of two and a speed of c. It generates a constant stream of one-period spaceships, one every two steps.

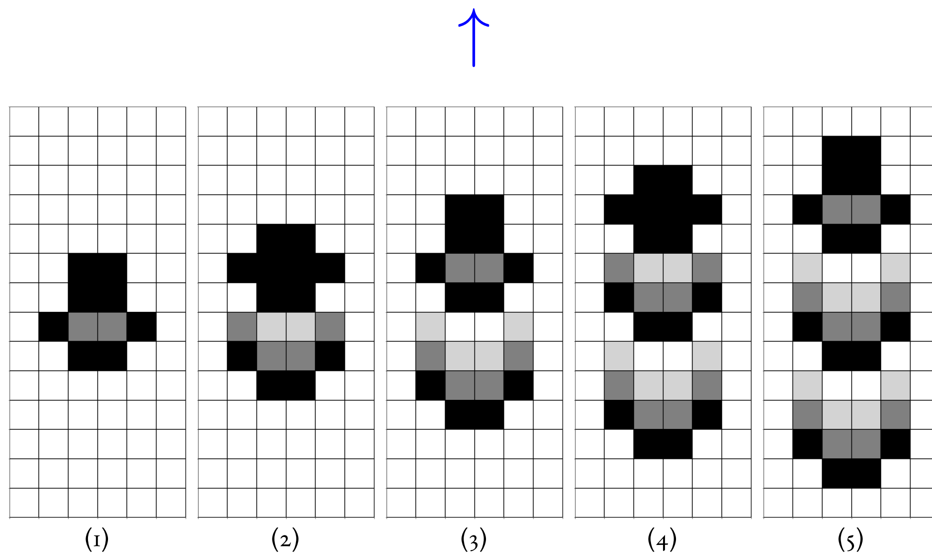


Figure 4.21: A rake that appears in rule "3,4,5/2/4". It has a period of two and a speed of c. It generates a constant stream of one-period spaceships, one every two steps. Code is provided in A.12.36.

We have also identified similar interactions between spaceships when they collide. There are countless cases where the two spaceships are both destroyed during the process. On the other hand, the other two cases are much rarer. We recorded one case of each kind in Figure 4.22 and Figure 4.23.

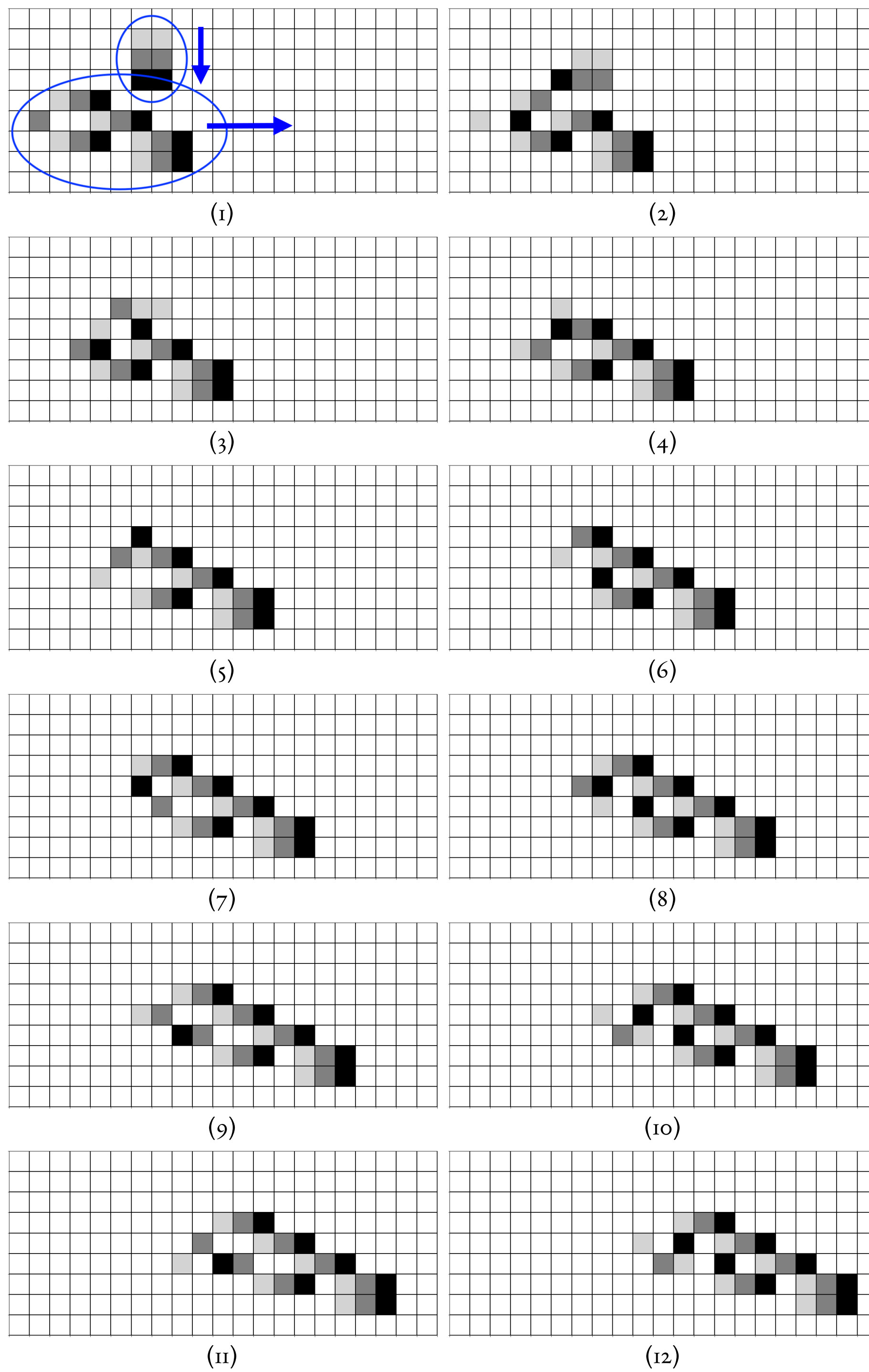


Figure 4.22: Example of a “combined spaceship” that appears in rule “3/2/4”. Code is provided in A.12.37.

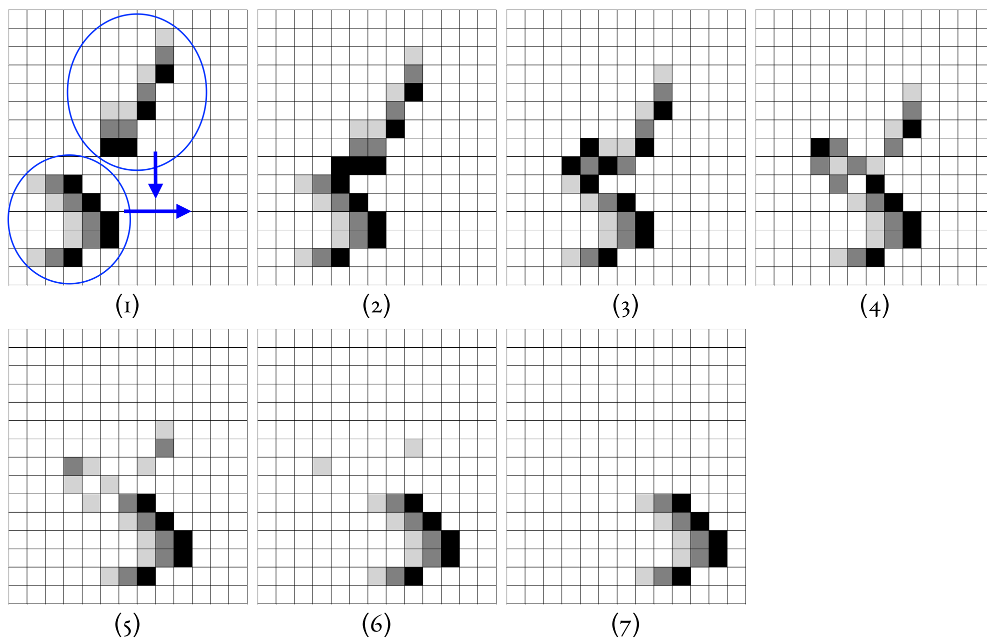
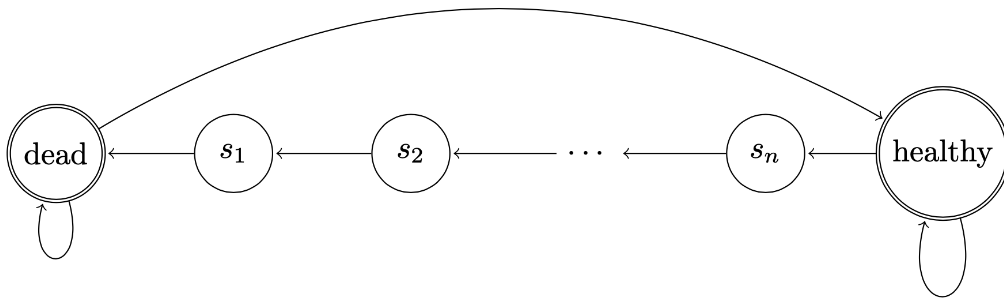


Figure 4.23: Example of a "murdered spaceship" that appears in rule "3/2/4". The bottom left spaceship murders the top right one in six steps. Code is provided in A.12.38.

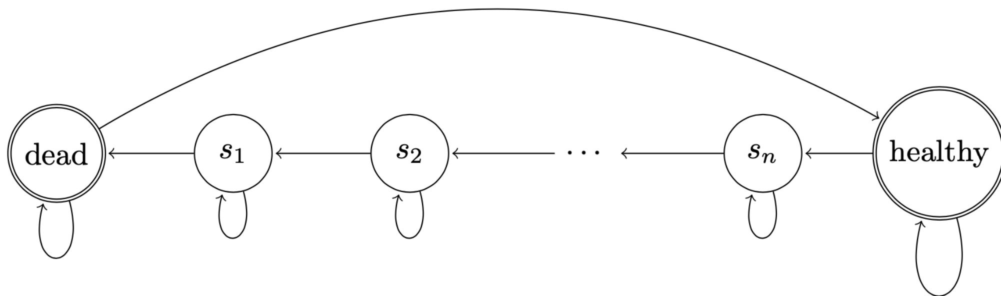
5 Future Work

So far this entire research has been performed in Google Colab using Python. However, once the pipeline has been properly set up, the entire process is highly parallelizable and thus can be run on many machines simultaneously. This can greatly increase the efficiency of the pipeline and minimize the overall time required for detecting more interesting rules. Eventually, we plan to migrate the code to C, which should speed up the process of simulation by at least a factor of 45 [15].

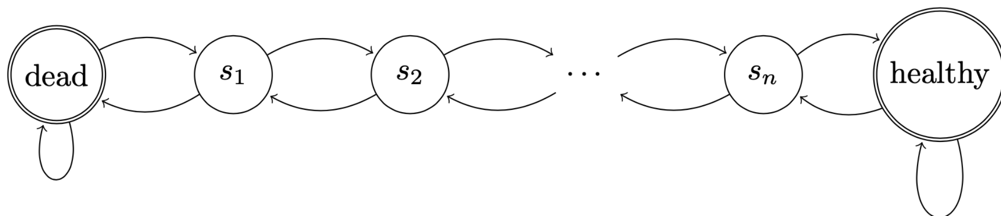
Furthermore, this report is only exploring cellular automata in the "instant birth, gradual death, no recovery" model, which is arguably the simplest model. This means that live (healthy) cells, once they fall sick, can never recover and can only be one step closer to death at each generation. However, there are many other viable models (shown in Figure 5.1) where the sick cells can have other outcomes, and additional rules are required to define the achievable behavior of the sick cells. Even though we cannot engineer the behavior of the cells directly, we can artificially engineer the models to simulate various real-life scenarios and harness them for our purposes. One simple example is that if we want the cells to imitate the basic behavior of human beings, then the sick cells should be allowed to remain on the same level of sickness or recover. If we also want to include the possible scenario of a deadly virus able to kill a healthy person instantly, then any sick and healthy cells should be allowed to die at any generation. We can even construct an imaginary chaotic world where all the cells are allowed to go to any other state at any generation. This report only studied one model, but there are many other possibilities we have not even enumerated, all of which are worth exploring and might contain spaceships and other "life" forms we have never dreamed of.



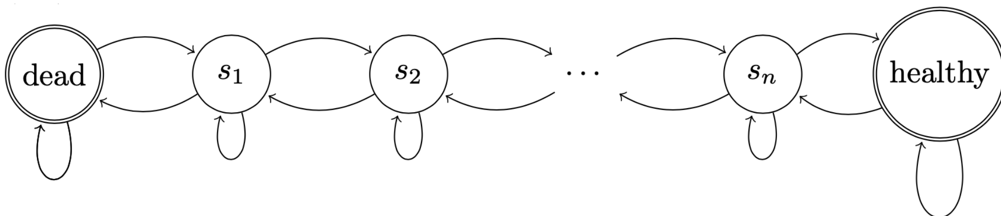
(1) "Instant birth, gradual death, no recovery" Model. Live (healthy) cells can get sick. Sick cells are not able to recover, and they will be one step closer to death at each step. Dead cells are always born healthy and cannot be born sick.



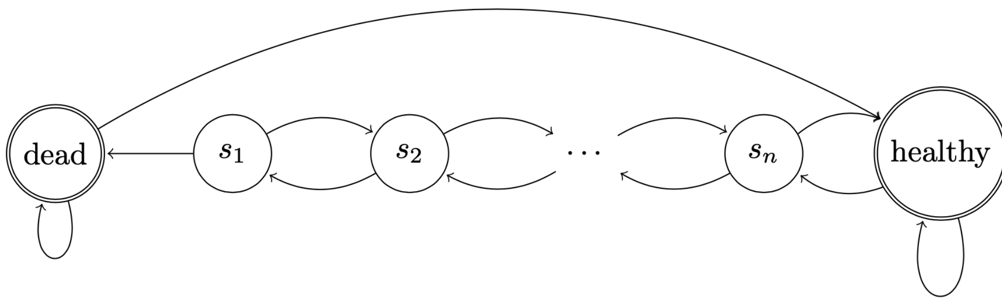
(2) "Instant birth, gradual death, stay sick, no recovery" Model. Live cells can get sick. Sick cells are not able to recover, and they will be either one step closer to death or stay the same at each step. Dead cells cannot be born sick.



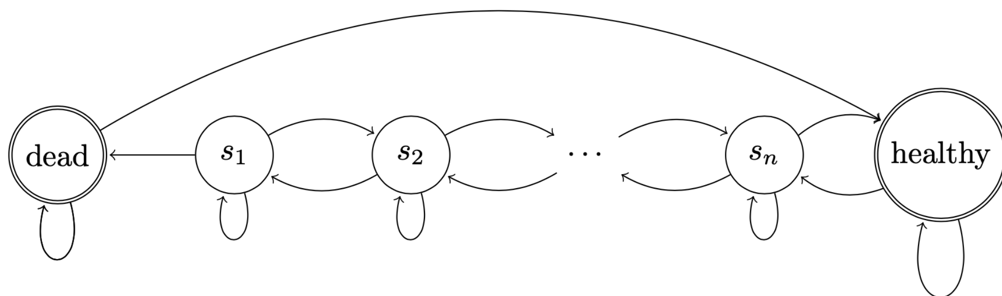
(3) "Gradual birth, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life at each step. Dead cells will be born sick.



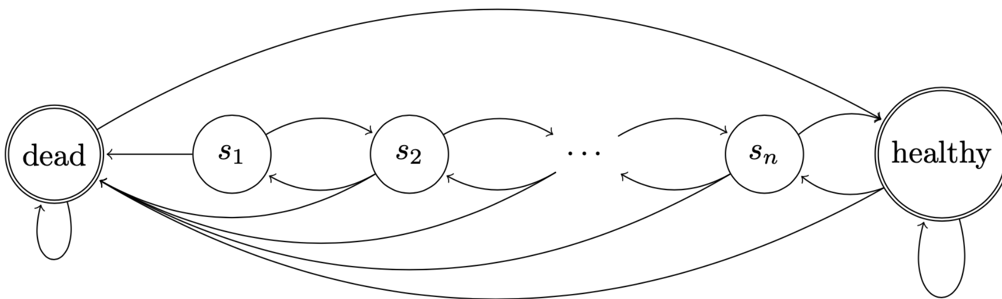
(4) "Gradual birth, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life or stay the same at each step. Dead cells will be born sick.



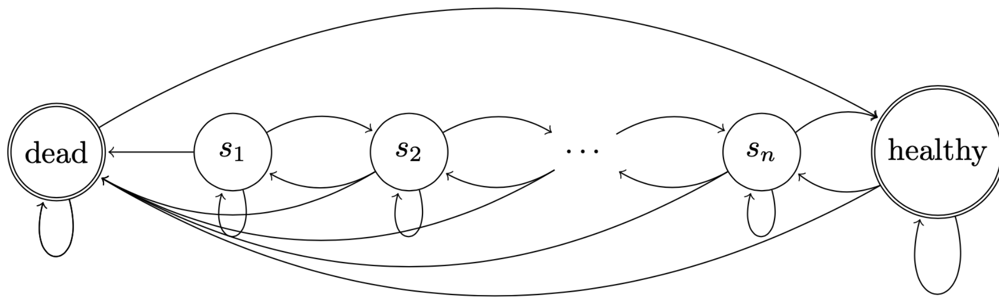
(5) "Instant birth, gradual death, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they will be one step closer to either death or life at each step. Dead cells cannot be born sick.



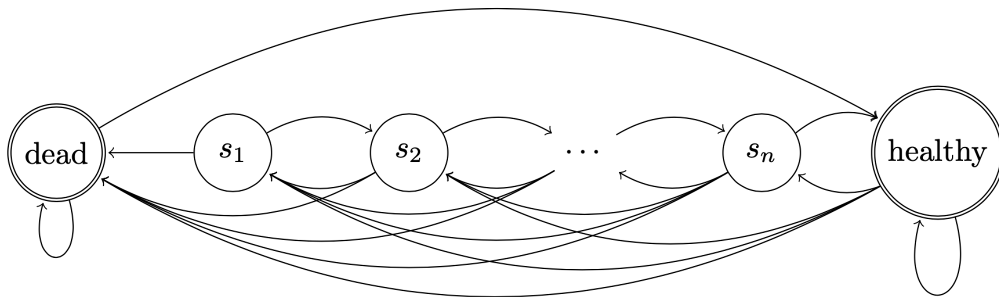
(6) "Instant birth, gradual death, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they will be one step closer to death or life or stay the same at each step. Dead cells cannot be born sick.



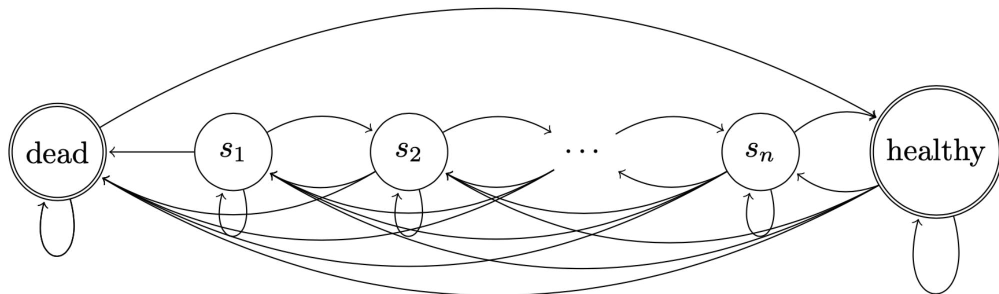
(7) "Instant birth, gradual and instant death, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life at each step. But they can also die immediately in the next step, imitating sudden death in real life. Dead cells cannot be born sick.



(8) "Instant birth, gradual and instant death, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life or stay the same at each step. But they can also die immediately in the next step, imitating sudden death in real life. Dead cells cannot be born sick.



(9) "Instant birth, any-level sicker, gradual recovery" Model. Live cells can get sick. Sick cells can recover as they can get one step closer to life at each step. Sick cells can also get sicker to any worse levels including death at each step. Dead cells cannot be born sick.



(10) "Instant birth, any-level sicker, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover as they can get one step closer to life at each step. They can also get sicker to any worse levels including death at each step. They can also stay the same. Dead cells cannot be born sick.

Figure 5.1: Examples of possible cellular automaton state transition diagrams ranked from the simplest to the most complicated. The vertices labeled with "s" represent the sick states.

Even within the current "instant birth, gradual death, no recovery" model we are exploring, there is still a huge potential search space. This report mostly explores cel-

lular automata with three states: the live (healthy) state, the dead state, and one sick state. We have identified a family of spaceships and the new Life, but there could be more that have not yet been found. Besides, one barely understands what cellular automata would look like if there are more than three possible states, i.e., if two or more sick states are included. We still do not know the answer to the questions like how many interesting rules are there, or what the spaceships would look like, or if there is any correlation between the rule and the structure of the spaceships. It is expected that as the number of sick cells increases, so would the complexity of the patterns and spaceships. The discovery of the beautiful rocket-like rake in Figure 4.21 gives us confidence that it is highly likely that there are more visually appealing spaceships with more states. Furthermore, this report focuses solely on spaceships and only classifies those rules that generate spaceships as interesting. However, there are many other interesting patterns in cellular automata that are worth investigating. For example, we can follow the same steps to find rules that generate oscillators.

An alternative future direction of this research is to perform temperature simulation with cellular automata. Namely, the closest analogy with cellular automata in real life is arguably particles and their behavior is controlled by temperature, which acts as the neighborhood rules in the space of cellular automata. For example, the particles are static in an environment of absolute zero. Hence, one possible direction of this research is to create neighborhood rules in cellular automata to simulate gradient temperature; one might find creatures huddling around ”hot springs”.

One other limitation of the current study is the relatively small size of the dataset. Because of the scarcity of cellular automata rules that have been classified, we had to perform data augmentation to make the dataset large enough for the machine learner. Specifically, we ran the same set of rules with random initial configurations to obtain diversified data. Ideally, this step could be skipped if we had sufficiently many classified rules available. Finding boring rules is relatively easier since they are the majority. The difficult part is finding the interesting ones, which would require much time and

labor. There are only a limited few provided by the Cellular Automata Lexicon and other sources [6], and the rest is constituted of an indefinitely long process of random guessing and brute-forcing. Theoretically, a good way to quickly enlarge our dataset is to use the machine learner we just trained to detect or at least narrow down more interesting rules and include them in the dataset. With more available data, the machine learner is expected to be more robust and perform better. Another potential method to improve the performance of the machine learner is to invest more in hyperparameter tuning and include the results from entropy analysis as features.

In Wolfram's 15-year view on "A New Kind of Science" [13], he claimed that with every passing year, he understood more about what the book was really about and why it was so important. The core of the book would go far beyond science and into many areas that will be increasingly important in defining our whole future. The book was fundamentally about something profoundly abstract: the theory of all possible theories, and the universe of all possible universes. For Wolfram, one of the biggest achievements was the realization that one can explore such fundamental things concretely—by doing actual experiments in the computational universe of possible programs. And in the end, we have a collection of what might at first seem like quite alien pictures made just by running very simple such programs. This report covers only a tip of an iceberg and points to the direction of doing systematic and controlled experiments in the infinite universe of theories and possibilities. There is so much left unknown, and so much to learn.

6 Conclusion

Despite the large amount of effort and study that has been put into cellular automata, there is still much that is unknown. The space of possible rules is infinite, so the task of determining the interesting ones that leads to spaceships is pertinent to the continued development of cellular automata theory. Spaceships have piqued the interest of many researchers since they literally represent life, and have proven to be beneficial to many biology and physics models [21]. Currently, there is no systematic method to automatically detect interesting rules. Existing methods are either too inefficient and expensive due to the enormous search space, or they are relatively fast but with poor accuracy. This report explores the possibility of using neural networks to find interesting rules in the "instant birth, gradual death, no recovery" model using the Moore neighborhood. Due to their capability to automatically learn and adapt during the training phase, they have the potential to approach the task that is too expensive for human labor and normal computer programs.

We created the data generation algorithm to compute the state of each cell at every step. After collecting the known interesting and boring rules from lexicons, we applied data augmentation to create a sufficiently large training and testing set by re-running the rules with different initial configurations. Each of the outcomes was recorded using the python package $CellPyLib$ as a sequence of grayscale frames. The first several generations were ignored because they depended highly on the initial configuration and therefore did not reflect the final stable pattern accurately. With many possible ways to explore the dataset, our final production pipeline used RNN, CNN, feature extraction, and entropy analysis.

The RNN approach resembles a video classification task. Each pattern is represented by its corresponding sequence of frames and each sequence is treated as one single instance. We performed hyperparameter tuning and eventually decided to use ConvLSTM2D, which led to the best accuracy. CNN on the other hand is the best at clas-

sifying images. Therefore, we had to stitch some of the frames together into one congregated image in advance. Each of the stitched images was considered as an instance. Different selections of frames that were included in the image have been tested. However, since this is an uncanonical approach, we wanted to find out the learnability of our dataset using Brainome.ai. To achieve this, we performed feature extraction using a pre-trained NASNet-Large CNN Model that is trained on a million images from the ImageNet database. For each of the instances, the machine learner extracts 1000 features and puts them in an organized CSV file. However, the results shown by Brainome.ai were suboptimal. Therefore, we eventually used the image pixels directly as features and this time, Brainome.ai showed us promising results. This gave us sufficient confidence to proceed with model training.

After the models are properly trained with extensive hyperparameter tuning, we used them to classify rules with three states and random survival and born rules and find spaceships in those that are classified as interesting. We discovered a handful of new interesting rules and found an entire family of spaceships, the new Life, and several rakes.

Based on the conclusions of this study, we think neural networks are adequate to be used to detect interesting cellular automata rules. The results that our machine learner found were undoubtedly interesting and worth further exploration.

7 References

[1] Wolfram S. *A New Kind of Science*. Wolfram Media, Champaign Il, 2002.

[2] Bays C. *Gliders in Cellular Automata*. In: Meyers R. (eds) Encyclopedia of Complexity and Systems Science. Springer, New York, NY, 2009. DOI: https://doi.org/10.1007/978-0-387-30440-3_249.

[3] Wuensche A. *Finding Gliders in Cellular Automata*. In: Adamatzky A. (eds) Collision-Based Computing. Springer, London, 2002. DOI: https://doi.org/10.1007/978-14471-0129-1_13.

[4] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, Wang-chun Woo. *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*. 2015, arXiv:1506.04214 [cs.CV].

[5] Christopher O. *Understanding LSTM Networks*. 2015, `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[6] *Cellular Automata Lexicon*. `http://psoup.math.wisc.edu/mcell/rullex_gene.html`.

[7] D'amico M, Manzini G, Margarac L. *On computing the entropy of cellular automata*. In: Theoretical Computer Science Volume 290, Issue 3, 3 January 2003, Pages 1629-1646. DOI: https://doi.org/10.1016/S0304-3975(02)00071-3.

[8] P. Hurd L, Kari J, Culik K. *The topological entropy of cellular automata is uncomputable*. In: Ergodic Theory and Dynamical Systems, Volume 12, Issue 2, June 1992, pp. 255 - 265. DOI: https://doi.org/10.1017/S0143385700006738.

[9] Brainome - measure and improve the learnability of your data. (2021, April 14). Retrieved May 10, 2021. `https://www.brainome.ai/`.

[10] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition (pp. 248–255).

[11] Google Colaboratory. (n.d.). Retrieved May 10, 2021. https://colab.research.google.com.

[12] David J.Eck. *Introduction to The Edge of Chaos*. Department of Mathematics and Computer Science, Hobart and William Smith Colleges. http://math.hws.edu/xJava/CA/EdgeOfChaos.html.

[13] Wolfram S. *A New Kind of Science: A 15-Year View*. May 16, 2017. https://writings.stephenwolfram.com/2017/05. /a-new-kind-of-science-a-15-year-view/.

[14] Martin G. *MATHEMATICAL GAMES The fantastic combinations of John Conway's new solitaire game "life"*. In Scientific American 223 (October 1970): 120-123. https://www.ibiblio.org/lifepatterns/october1970.html.

[15] Xie P. *How Slow is Python Compared to C*. https://medium.com/codex/how-slow-is-python-compared-to-c-3795071ce82a

[16] "Cellular automaton." Merriam-Webster.com Dictionary. (2021, May 10). Merriam-Webster. https://www.merriam-webster.com/dictionary/cellular%20automaton.

[17] Spaceship (cellular automaton). (2021, April 23). In Wikipedia. https://en.wikipedia.org/wiki/Spaceship_(cellular_automaton).

[18] Rake (cellular automaton). (2021, January 13). In Wikipedia. https://en.wikipedia.org/wiki/Rake_(cellular_automaton).

[19] Speed of light (cellular automaton). (2020, April 12). In Wikipedia. https://en.wikipedia.org/wiki/Speed_of_light_(cellular_automaton).

[20] The Game of Life. (2021, April 21). In Wikipedia. https://en.wikipedia.org/wiki/The_Game_of_Life.

[21] Cellular automaton. (2021, April 26). In Wikipedia. https://en.wikipedia.org/wiki/Cellular_automaton.

[22] Rule 30. (2021, January 13). In Wikipedia. https://en.wikipedia.org/wiki/Rule_30.

[23] Glider (Conway's Life). (2021, April 20). In Wikipedia. https://en.wikipedia.org/wiki/Glider_(Conway's_Life).

[24] Gun (cellular automaton). (2021, January 31). In Wikipedia. https://en.wikipedia.org/wiki/Gun_(cellular_automaton).

A Appendix

All of the written code and generated gifs can be found at the GitHub repository: https://github.com/Liaoqitian/MLCA.

A.1 Cellular Automata Generation Algorithm

```
def generate(neighborhood,max_state,survive_arr,born_arr):
    center_cell = neighborhood[1][1]
    live_cells_count = np.sum((neighborhood == max_state).
   astype(int))
    if center_cell == max_state:
        for num_neighbors in survive_arr:
            if live_cells_count - 1 == num_neighbors:
                return center_cell
            return center_cell - 1
    else if center_cell != 0 and center_cell != max_state:
        return center_cell - 1
    else:
        for num_neighbors in born_arr:
            if total == num_neighbors:
                return max_state
        return 0
```

- Input: The neighborhood (a two-dimensional array), the total number of states, the survival rule (a one-dimensional array), and the born rule (a one-dimensional array).
- Output: The state of the center cell in the next generation.
- Description: Computes the state of the center cell in the next generation.
- Example call:

```
neighborhood = [[0, 0, 0], [0, 1, 1], [0, 0, 0]]
max_state = 2
survive_arr = [2, 3]
born_arr = [3]
generate(neighborhood, max_state, survive_arr, born_arr)
# returns 0
```

A.2 The 35 Selected Interesting Rules

Survival Rules	Death Rules	Number of States
3, 4, 5	2, 4	25
6	2, 4, 6	3
0, 2, 3, 5, 6, 7, 8	3, 4, 6, 8	9
2, 3, 5, 6, 7, 8	3, 4, 6, 8	9
2	1, 3	21
0, 3, 5, 6, 7, 8	2, 4, 5, 6, 7, 8	7
0, 3, 5, 6, 7, 8	2, 4, 5, 6, 7, 8	5
3, 4, 5	3	6
3	2	4
3, 4, 5	3, 4	6
3, 4, 6, 7	2, 6, 7, 8	6
0, 3, 4, 6, 7	2, 5	6
2, 3	3, 4	8
0, 3, 4, 5	2, 6	6
3, 4, 5	3, 4, 6, 7, 8	5

Survival Rules	Death Rules	Number of States
2, 4, 5	3, 6, 8	2
2, 3, 6, 7	3, 4, 5, 7	5
3, 4, 6, 7	2, 5	6
6	2	3
1, 2, 5	3, 6	2
3, 4, 6, 7	2	4
N/A	2	3
2	2	8
2, 3	2	8
2, 3	3	2
2, 3	3, 6	2
2, 3, 8	3, 6, 8	2
2, 3, 8	3, 5, 7	2
2, 5, 6	2, 4, 5	5
3, 4, 5	2	4
4, 5, 6, 7	2, 3, 5, 8	5
3	2, 5	3
0	2, 6	4
0, 4, 7, 8	2, 3, 5, 6	5
3, 4, 5	2, 6	5

A.3 Frame Extraction

```
def frame_extraction(file_path):
    frames_list = []
    for x in range(80, 120):
        image = cv2.imread(file_path+str(x)+".png", cv2.
    IMREAD_GRAYSCALE)
        image = cv2.resize(image, (IMG_SIZE, IMG_SIZE))
        frames_list.append(image)
```

- Input: a file path of a folder containing the images.
- Output: a list of images.
- Description: appends a selection of images of a pattern into a list.

A.4 Recurrent Neural Network Implementation

```
model = Sequential()
model.add(ConvLSTM2D(filters = 64, kernel_size = (5, 5),
    return_sequences = False, data_format = "channels_last",
    input_shape = X.shape[1:]))
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.15))
model.add(Flatten())
model.add(Dense(256, activation="relu"))
model.add(Dropout(0.15))
model.add(Dense(64, activation="relu"))
model.add(Dropout(0.15))
model.add(Dense(2, activation = "softmax"))
model.add(Activation("sigmoid"))
model.compile(loss='categorical_crossentropy', optimizer=opt
    , metrics=["accuracy"])
```

- Input: nothing
- Output: an RNN model
- Description: detailed implementation of the RNN.

A.5 Image Stitching Function

```
def stitch_images(file_path, file_name, start_frame,
    num_frames, save_DIR):
    images = [Image.open(image) for image in [file_path + "/
    " + file_name + str(x) + ".png" for x in range(
    start_frame, start_frame + num_frames)]]
    widths, heights = zip(*(i.size for i in images))
```

```
    dimension = int(math.sqrt(num_frames))
    total_width = int(sum(widths) / dimension)
    total_height = int(sum(heights) / dimension)
    new_image = Image.new("RGB", (total_width, total_height)
    )
    for index in range(0, num_frames):
        image = images[index]
        new_image.paste(image, ((index \% dimension ) *
    image.size[0], math.floor(index / dimension) * image.size
    [1]))
    save_DIR = save_DIR + "combined_" + file_name + ".png"
    new_image.save(save_DIR)
return
```

- Input: a file path leading to the images, the name of the file, the starting frame, the number of frames, the path to save the output.
- Output: nothing.
- Description: stitches a selection of images together into a square.

A.6 Image Feature Extraction with NASNet-Large

```
model_name="nasnetalarge"
model=pretrainedmodels.__dict__[model_name](num_classes
    =1000, pretrained='imagenet')
model.eval()
load_img = utils.LoadImage()
tf_img = utils.TransformImage(model)
features_file = open("file.csv", "ab")
feature_data = []
for i in range(len(image_paths)):
    input_img = load_img(image_paths[i])
    input_tensor = tf_img(input_img)
    input_tensor = input_tensor.unsqueeze(0)
    input = torch.autograd.Variable(input_tensor,
    requires_grad=False)
    output_logits = model(input)
    output_features = model.features(input)
    output_logits = model.logits(output_features)
    output_logits = output_logits[0].detach().numpy()
    row_data = np.append(output_logits, labels[i])
    feature_data = np.append(feature_data, row_data)
```

- Input: a file path leading to the images.
- Output: 1000 extracted features for each of the image.
- Description: extract features from images with NASNet-Large.

A.7 Image Feature Extraction with Image Pixels

```
def extract_features(IMAGE_DIR):
    img_array = cv2.imread(IMAGE_DIR, cv2.IMREAD_GRAYSCALE)
    feature = np.reshape(new_array,(new_array.shape[0]*
   new_array.shape[1]))
feature_extraction_data.append([feature, class_num])
```

- Input: a file path leading to the images.
- Output: a number of extracted features depending on the size of the image.
- Description: extract features from images with pixles.

A.8 Convolutional Neural Network Implementation

```
model = keras.Sequential()
model.add(Conv2D(64, (5, 5), input_shape=tempx.shape[1:]))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.15))

model.add(Conv2D(64, (3, 3)))
model.add(BatchNormalization())
model.add(Activation("relu"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.15))

model.add(Flatten())
model.add(Dense(64))
model.add(Activation("relu"))
model.add(Dropout(0.15))

model.add(Dense(10))
model.add(Activation("relu"))
model.add(Dropout(0.15))

model.add(Dense(1))
model.add(Activation("sigmoid"))
model.compile(loss="binary_crossentropy", optimizer="rmsprop
   ", metrics=["accuracy"])
```

- Input: nothing
- Output: a CNN model
- Description: detailed implementation of the CNN.

A.9 Image Cross-Entropy Computation

```
def COMPUTE_ENTROPY(signal)
    lensig = signal.size
    symset = list(set(signal))
    probpab = [np.size(signal[signal == i])/(1.0*lensig) for
    i in symset]
    entropy = np.sum([p * np.log2(1.0 / p) for p in  propab
   ])
    return entropy

label_entropies = {'Boring': [], 'Interesting': []}
for i, instance in enumerate(X):
    instance_1d = instance.ravel()
    entropy = compute_entropy(instance_1d)
    label_id = y[i]
    if label_id == 0:
        label_entropies['Boring'].append(entropy)
    else:
        label_entropies['Interesting'].append(entropy)
```

- Input: individual images.
- Output: cross-entropy value of images.
- Description: computes the cross-entropy value of the patterns. The entropy values are divided into interesting and boring for further analysis.

A.10 Maximum Memory Capacity Prediction

```
data: array of length i containing vectors x with
   dimensionality d
labels: a column containing 0 or 1
function COMPUTE_MEC(data, labels)
  thresholds = 0
  loop over i: table[i] = \sigma x[i][d], label[i]
  sorted table = sort(table, key = column 0)
  class = 0
  loop over i: if not sortedtable[i][1] == class then
      class = sortedtable[i][1]
      thresholds = thresholds + 1
  end
  maxcapreq = threshold * d + thresholds + 1
  expcapreq = log2 (threshold + 1) * d
  return maxcapreq, expcapreq
```

- Input: labeled data
- Output: the maximum and expected capacity requirement of the machine learner.

- Description: computes the maximum and expected memory capacity of the machine learner.

A.11 Spaceship Image and GIF generation

```
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np
import os
import numpy as np
import matplotlib.pyplot as plt
import imageio

def count_neighbors(data, i, j):
    res = 0
    max_state = number_states - 1
    if i > 0 and data[i - 1][j] == max_state:
        res += 1
    if i < len(data) - 1 and data[i + 1][j] == max_state:
        res += 1
    if j > 0 and data[i][j - 1] == max_state:
        res += 1
    if j < len(data[0]) - 1 and data[i][j + 1] == max_state:
        res += 1
    if i > 0 and j > 0 and data[i - 1][j - 1] == max_state:
        res += 1
    if i > 0 and j < len(data[0]) - 1 and data[i - 1][j + 1]
    == max_state:
        res += 1
    if i < len(data) - 1 and j > 0 and data[i + 1][j - 1] ==
    max_state:
        res += 1
    if i < len(data) - 1 and j < len(data[0]) - 1 and data[i
    + 1][j + 1] == max_state:
        res += 1
    return res

def evolve(data, survival_arr, born_arr):
    copy = [[0 for j in range(length)] for i in range(width)
   ]
    max_state = number_states - 1
    for i in range(width):
        for j in range(length):
            if data[i][j] > 0 and data[i][j] < max_state:
                copy[i][j] = data[i][j] - 1
            else:
                count = count_neighbors(data, i, j)
                if data[i][j] == max_state and count in
   survival_arr:
                    copy[i][j] = data[i][j]
```

```
41                  if data[i][j] == max_state and count not in
    survival_arr:
42                      copy[i][j] = data[i][j] - 1
43                  elif data[i][j] == 0 and count in born_arr:
44                      copy[i][j] = max_state
45
46 filenames = []
47 length = len(data[0])
48 width = len(data)
49
50 for step in range(period):
51     if number_states == 3:
52         cmap = colors.ListedColormap(['white', 'gray', '
    black'])
53         bounds = [-0.5,0.5,1.5,2.5] # White: 0, Gray: 1,
    Black: 2
54     elif number_states == 4:
55         cmap = colors.ListedColormap(['white', 'lightgray',
    'gray', 'black'])
56        # White:0, lightgray:1, Gray:2, Black:3
57         bounds = [-0.5,0.5,1.5,2.5,3.5]
58     norm = colors.BoundaryNorm(bounds, cmap.N)
59     fig, ax = plt.subplots()
60     ax.imshow(data, cmap=cmap, norm=norm)
61
62     # draw gridlines
63     ax.grid(which='major', axis='both', linestyle='-', color
    ='k', linewidth=2)
64     ax.set_xticks(np.arange(-.5, length, 1));
65     ax.set_yticks(np.arange(-.5, width, 1));
66
67     frame1 = plt.gca()
68     frame1.axes.xaxis.set_ticklabels([])
69     frame1.axes.yaxis.set_ticklabels([])
70     plt.rcParams["figure.figsize"] = (20,20)
71     plt.savefig(f'{step}.png', bbox_inches='tight')
72     filenames.append(f'{step}.png')
73
74     plt.tick_params(axis = "x", which = "both", bottom =
    False, top = False)
75     data = evolve(data, survival_arr, born_arr)
76
77 with imageio.get_writer('mygif.gif', mode='I', duration =
    0.5) as writer:
78     for filename in filenames:
79         image = imageio.imread(filename)
80         writer.append_data(image)
```

- Input: the initial configuration, the number of states, the survival rule, the born rule, and the number of generations.

- Output: a set of images whose number depends on the number of generations and a gif.
- Description: visualize the generations of cellular automata with four or fewer states with images and gif.

A.12 Initial configuration of the gliders

This section contains a list of initial configurations and the corresponding parameters of the spaceships mentioned in this report. One can pass these as parameters into the function in Section A.11 to reproduce the images and gif.

A.12.1 Code for figure 1.3, the Game of Life

```
survival_arr = [2, 3]
born_arr = [3]
number_states = 2
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.2 Code for figure 1.3, the light-weight spaceship

```
survival_arr = [2, 3]
born_arr = [3]
number_states = 2
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.3 Code for figure 1.3, the mid-weight spaceship

```
survival_arr = [2, 3]
born_arr = [3]
number_states = 2
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.4 Code for figure 1.3, the heavy-weight spaceship

```
survival_arr = [2, 3]
born_arr = [3]
number_states = 2
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.5 Code for figure 4.1

```
survival_arr = [6]
born_arr = [2,4,6]
number_states = 3
period = 2
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 2, 2, 0, 0, 0, 0],
    [0, 0, 2, 2, 0, 0, 2, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
    [0, 0, 2, 2, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 2, 2, 0],
    [0, 2, 1, 1, 2, 0, 0, 0, 1, 1, 0, 0, 0, 0, 2, 1, 1, 0],
    [0, 1, 0, 0, 1, 0, 0, 2, 0, 0, 2, 0, 0, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0, 0, 0, 0, 2, 2, 0, 0, 0, 2, 2, 0, 0],
    [0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 2, 1, 1, 0, 0],
    [0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 2, 1, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 2, 2, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 1, 1, 0, 0, 0, 2, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0],
    [0, 2, 0, 0, 0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 2, 0, 0, 0],
    [0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.6 Code for figure 4.2

```
survival_arr = [6]
born_arr = [2,4,6]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 2, 2, 0, 0],
    [0, 2, 1, 1, 2, 0],
    [0, 1, 0, 0, 1, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.7 Code for figure 4.3

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0],
    [0, 0, 0, 1, 1, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 2, 1, 2, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.8 Code for figure 4.3

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0],
    [0, 0, 2, 2, 0],
    [0, 0, 1, 1, 0],
    [0, 2, 0, 0, 0],
    [0, 1, 2, 0, 0],
    [0, 0, 1, 2, 0],
    [0, 2, 0, 1, 0],
    [0, 1, 2, 0, 0],
    [0, 0, 0, 0, 0]
]
```

A.12.9 Code for figure 4.3

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0],
    [0, 0, 0, 1, 1, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 2, 0, 2, 0, 0],
    [0, 1, 0, 1, 2, 0],
    [0, 0, 1, 0, 1, 0],
    [0, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.10 Code for figure 4.3

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 2, 0, 0],
    [0, 2, 1, 2, 1, 2, 0],
    [0, 1, 0, 2, 0, 1, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.11 Code for figure 4.3

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0],
    [0, 0, 0, 1, 1, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 2, 0, 2, 0, 0],
    [0, 1, 2, 1, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.12 Code for figure 4.5

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0],
    [0, 0, 2, 1, 1, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 2, 0, 2, 0, 0],
    [0, 1, 2, 1, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.13 Code for figure 4.6 (1/5)

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 2, 1, 2, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.14 Code for figure 4.6 (2/5)

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0],
    [0, 0, 0, 1, 1, 0],
    [0, 0, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0],
    [0, 2, 0, 0, 0, 0],
    [0, 1, 2, 0, 0, 0],
    [0, 0, 1, 2, 0, 0],
    [0, 2, 0, 1, 0, 0],
    [0, 1, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.15 Code for figure 4.6 (3/5)

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0],
    [0, 2, 0, 2, 0, 0, 0],
    [0, 1, 0, 1, 2, 0, 0],
    [0, 0, 1, 0, 1, 0, 0],
    [0, 2, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.16 Code for figure 4.6 (4/5)

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 2, 0, 2, 0, 0, 0],
    [0, 2, 1, 2, 1, 2, 0, 0],
    [0, 1, 0, 2, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.17 Code for figure 4.6 (5/5)

```
survival_arr = [4]
born_arr = [2]
number_states = 3
period = 3
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0],
    [0, 2, 0, 2, 0, 0, 0],
    [0, 1, 2, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.18 Code for figure 4.8 (1/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [0, 2, 1, 0, 2, 0, 0],
    [0, 1, 0, 2, 1, 0, 0],
    [0, 0, 2, 2, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.19 Code for figure 4.8 (2/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 2, 1, 2, 0, 0],
    [0, 2, 1, 0, 1, 0, 0],
    [0, 1, 0, 2, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.20 Code for figure 4.8 (3/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 1, 2, 0, 0],
    [0, 0, 2, 1, 0, 1, 0, 0],
    [0, 2, 1, 0, 2, 0, 0, 0],
    [0, 1, 0, 2, 0, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.21 Code for figure 4.9 (4/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0],
    [0, 0, 2, 0, 2, 0, 0],
    [0, 0, 1, 2, 1, 0, 0],
    [0, 2, 0, 2, 0, 2, 0],
    [0, 1, 0, 0, 0, 1, 0],
    [0, 0, 2, 0, 2, 0, 0],
    [0, 0, 1, 2, 1, 0, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.22 Code for figure 4.9 (5/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 2, 1, 0, 0, 0],
    [0, 0, 1, 0, 2, 0, 0],
    [0, 2, 0, 2, 1, 2, 0],
    [0, 1, 2, 2, 0, 1, 0],
    [0, 0, 1, 0, 2, 0, 0],
    [0, 2, 1, 0, 1, 0, 0],
    [0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.23 Code for figure 4.9 (6/6)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 1, 0, 0, 0],
    [0, 0, 2, 1, 0, 2, 0, 0],
    [0, 2, 1, 0, 0, 1, 0, 0],
    [0, 1, 0, 2, 1, 0, 0, 0],
    [0, 0, 2, 1, 1, 2, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 1, 0, 1, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.24 Code for figure 4.10 (1/2)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 9
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 2, 1, 2, 0, 0],
```

```
    [0, 0, 0, 2, 1, 0, 1, 0, 0],
    [0, 0, 2, 1, 0, 2, 0, 0, 0],
    [0, 2, 1, 0, 2, 0, 0, 0, 0],
    [0, 1, 0, 2, 1, 0, 0, 0, 0],
    [0, 0, 2, 2, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.25 Code for figure 4.10 (2/2)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 9
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 2, 1, 0, 0, 0, 0],
    [0, 0, 0, 2, 1, 0, 2, 0, 0, 0],
    [0, 0, 2, 1, 0, 2, 1, 0, 0, 0],
    [0, 2, 1, 0, 2, 2, 0, 0, 0, 0],
    [0, 1, 0, 2, 1, 0, 0, 0, 0, 0],
    [0, 0, 2, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.26 Code for figure 4.12 (1/2)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 2, 0, 0, 0, 0],
    [0, 0, 0, 1, 1, 0, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0],
    [0, 2, 0, 2, 0, 0, 0, 0, 0],
    [0, 1, 0, 1, 2, 0, 0, 0, 0],
    [0, 0, 1, 0, 1, 0, 0, 0, 0],
    [0, 2, 1, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 2, 0, 0],
    [0, 0, 0, 0, 2, 0, 1, 2, 0],
    [0, 0, 0, 0, 1, 2, 0, 1, 0],
    [0, 0, 0, 0, 0, 2, 2, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.27 Code for figure 4.12 (2/2)

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 0, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 1, 2, 1, 2, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 2, 1, 2, 0, 0, 0, 0, 0, 2, 1, 2, 0, 0],
    [0, 2, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 2, 0],
    [0, 1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0],
    [0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0],
    [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.28 Code for figure 4.13

```
survival_arr = [4,6]
born_arr = [2]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 2, 1, 0, 0, 0],
    [0, 0, 2, 1, 0, 2, 0, 0],
    [0, 2, 1, 0, 0, 1, 0, 0],
    [0, 1, 0, 2, 1, 0, 2, 0],
    [0, 0, 2, 2, 0, 2, 1, 0],
    [0, 0, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.29 Code for figure 4.14

```
survival_arr = [6]
born_arr = [2]
number_states = 3
period = 9
data = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 2, 2, 0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 0, 0, 0],
    [0, 0, 2, 1, 2, 0, 0, 0],
    [0, 0, 1, 0, 1, 2, 0, 0],
    [0, 2, 0, 2, 0, 1, 2, 0],
    [0, 1, 2, 2, 2, 0, 1, 0],
    [0, 0, 1, 0, 1, 0, 0, 0],
    [0, 2, 1, 0, 1, 1, 0, 0],
    [0, 0, 2, 0, 0, 0, 0, 0],
```

```
    [0, 0, 2, 1, 1, 0, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.30 Code for figure 4.15

```
survival_arr = [6]
born_arr = [2,4,6]
number_states = 3
period = 9
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.31 Code for figure 4.16

```
survival_arr = [6]
born_arr = [2,4,6]
number_states = 3
period = 5
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 2, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 2, 1, 2, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.32 Code for figure 4.17

```
survival_arr = [4, 6]
born_arr = [2]
number_states = 3
period = 4
data = [
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 1, 2, 0, 2, 1, 0],
    [0, 1, 2, 0, 2, 1, 0],
    [0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0]
]
```

A.12.33 Code for figure 4.18

```
survival_arr = [4, 6]
born_arr = [2]
number_states = 3
period = 8
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0, 0, 2, 0, 2, 0],
    [0, 0, 0, 2, 2, 0, 1, 1, 0, 0],
    [0, 1, 0, 1, 0, 0, 2, 0, 0, 0],
    [0, 0, 2, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.34 Code for figure 4.19

```
survival_arr = [4, 6]
born_arr = [2]
number_states = 3
period = 4
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 2, 0, 0, 0, 0],
    [0, 0, 0, 1, 2, 0, 2, 2, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 0],
    [0, 0, 0, 0, 0, 2, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.35 Code for figure 4.20

```
survival_arr = [4, 6]
born_arr = [2]
number_states = 3
period = 8
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
```

```
    [0, 1, 2, 0, 2, 1, 2, 0, 0, 0, 0],
    [0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```

A.12.36 Code for figure 4.21

```
survival_arr = [3, 4, 5]
born_arr = [2]
number_states = 4
period = 5
data = [
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 3, 3, 0, 0],
    [0, 0, 3, 3, 0, 0],
    [0, 3, 2, 2, 3, 0],
    [0, 0, 3, 3, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]
```

A.12.37 Code for figure 4.22

```
survival_arr = [3]
born_arr = [2]
number_states = 4
period = 12
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 2, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 1, 2, 3, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
```

```
    [0, 0, 0, 0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
   0, 0, 0]
]
```

A.12.38 Code for figure 4.23

```
survival_arr = [3]
born_arr = [2]
number_states = 4
period = 7
data = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 1, 3, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 1, 1, 3, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 3, 3, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0],
    [0, 1, 2, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
]
```