Abstract

Cellular automata is a collection of colored cells on a grid that evolves according to a set of neighborhood rules. The rules, which consists of a born rule, a survive rule, and the number of possible states, are applied iteratively for as many times steps as desired to generate new grid configurations. There are many possible configurations and this paper specifically explores celluluar automata using Moore neighborhood in the deterministic model of "Instant birth, gradual death, no recovery", meaning that sick cells are not able to recover and have to move one step closer to death at each generation.

A challenge that two-dimensional cellular automata face is a huge parameter search space to generate patterns within. The number of total possible combinations of parameters in the rules can easily exceed billion. In this paper, we define the rules that generate gliders as interesting, not other rules that can be subjectively labeled as interesting, such as intricate still life and oscillating patterns. Existing research has not discovered a clear pattern among rules which generate gliders (oscillating translators that move across the grid).

Manually searching for the interesting rules would be unrealistic as users may have to randomly go through hundreds if not thousands of random rules before finding an interesting one. The introduction of neural networks has revolutionized a variety of classification tasks. This paper explores the potential of using neural networks to detect interesting cellular automata rules. Specifically, we will discuss our approach to detect interesting rules using Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), feature extraction, entropy analysis, and other techniques. We then put the trained machine learners into practice and detected several interesting rules with three possible states. We found an entire family of gliders of different periods and many other interesting results.

Acknowledgements

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 paper. 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 paper. 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, 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 paper 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.

Contents

1 Introduction

A cellular automaton is a collection of cells on a grid of a specified shape that evolves through discrete time steps according to a set of rules [7]. The grid can be in any finite number of dimensions, and the evolution rules usually consist of constraints on the current state of the cell and the states of the cells in the neighborhood. The most common neighborhood configurations are von Neumann neighborhood and Moore neighborhood in Figure 1.1, which includes the surrounding four and eight cells respectively. Cellular automata have attracted much attention among scientists and they have been regarded by Stephen Wolfram as the "new kind of science" [4]. What makes cellular automata so special is that 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 the 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 think of their dynamics as a microworld where the cells constitute their own ecosystems.

The arguably most famous cellular automaton is John Conway's the Game of Life [6] that was initially revealed to the public in a 1970 Scientific American article. The Game of Life is a two-dimensional cellular automaton with two possible states, live and dead, using Moore neighborhood and the following set of rules:

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.

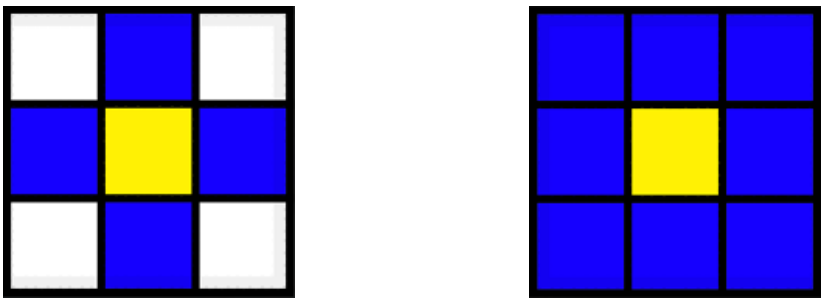


FIGURE 1.1: (Left) Neumann Neighborhood includes the surrounding four cells of a center cell. (Right) Moore Neighborhood includes the surrounding eight cells of the center cell.

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 2/9 and 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 is joined to the vertex representing the state reached after one step in the evolution. 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.

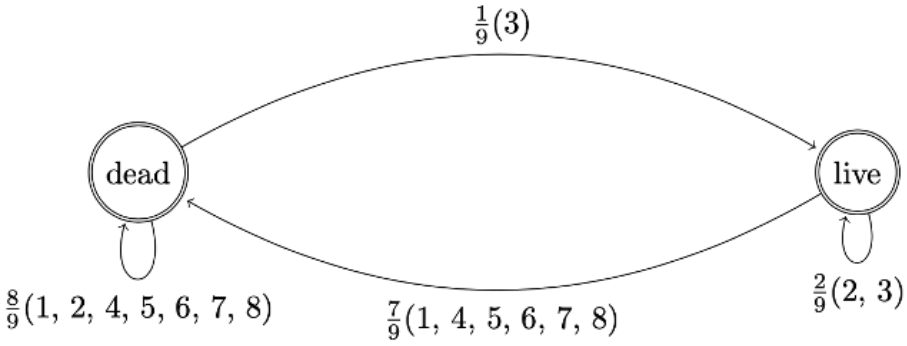


FIGURE 1.2: Transitional state diagram of the Game of Life.

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 these sick cells interact with the other. In the rest of the paper, healthy and live cells refer to the same thing and are thus used in-

terchangeably. These newly added rules, combined with the survival and born rules, constitute a new set of evolution rules.

The feature that makes the Game of Life so well-known is undoubtedly the discovery of "gliders" [9]. Some of the most frequently generated patterns 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 [7]. "Gliders", also known as translating oscillators, "spaceships" 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 [2]. They are usually considered the most interesting pattern and are widely used for modelling complicated nonlinear systems in computational science, physics, chemistry, and biology. If we think of cellular automata as an ecosystem, then the gliders are a unique kind of independent life form within. Finding gliders can potentially help us answer questions like how close we are simulating life, or whether we can make the life forms intelligent so they can adapt to changing environment.

Period and speed are the two frequently utilized metric to describe a glider. Period refers to the number of ticks a pattern must iterate through before returning to its initial configuration [6]. The speed of the glider is expressed in terms of the metaphorical speed of light, c [1,3]. The speed of light is a propagation rate across the grid of exactly one step, either horizontally, vertically, or diagonally, per generation. Because 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 glider 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 glider found by Conway is the first one in Figure 1.3 with period of 4 and a speed of $c/4$, as it takes four generations for a given state to be translated by

one cell diagonally. Obviously, there are many other cellular automata rules besides the Game of Life that can produce gliders. In Figure 1.4 are some of the famous gliders that have been generated with other sets of rules featured in Cellular Automata Rules Lexicon.

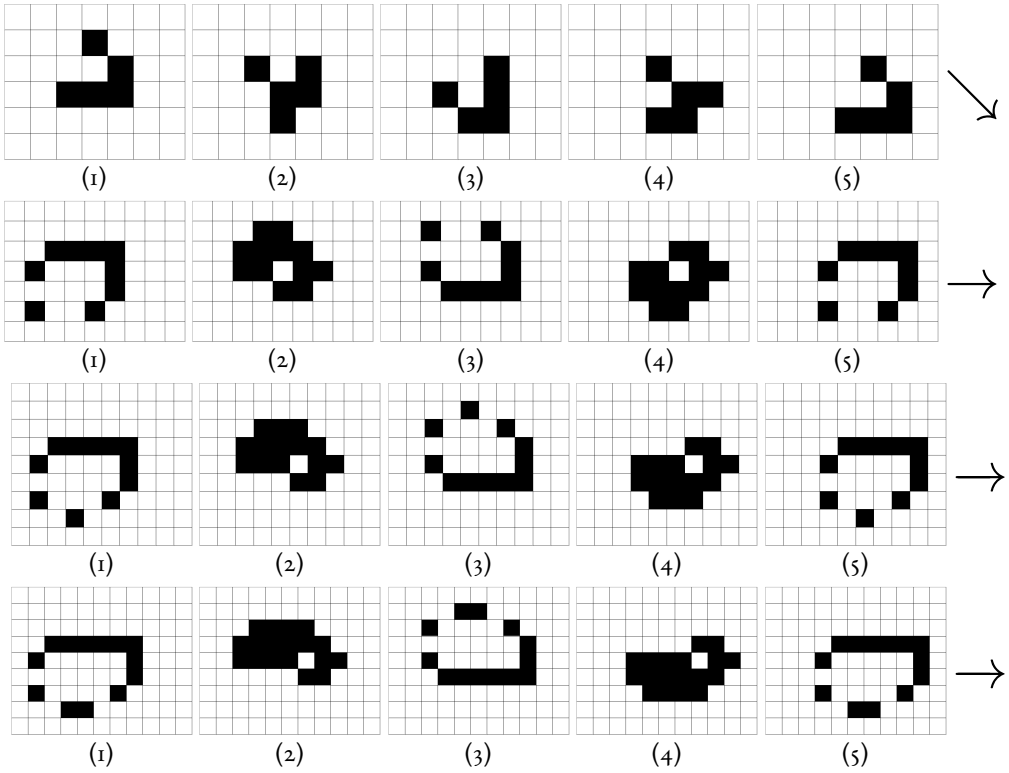


FIGURE 1.3: Gliders generated by the Game of Life "2,3/3/2" rule. The top is the original glider first found by Conway. The rest are the light-weight, mid-weight and heavy-weight spaceships respectively. All four gliders have a period of four. The light-weight, mid-weight and heavy-weight spaceship have a speed of $c/2$, as it takes four generations for a given state to be translated by two cells.

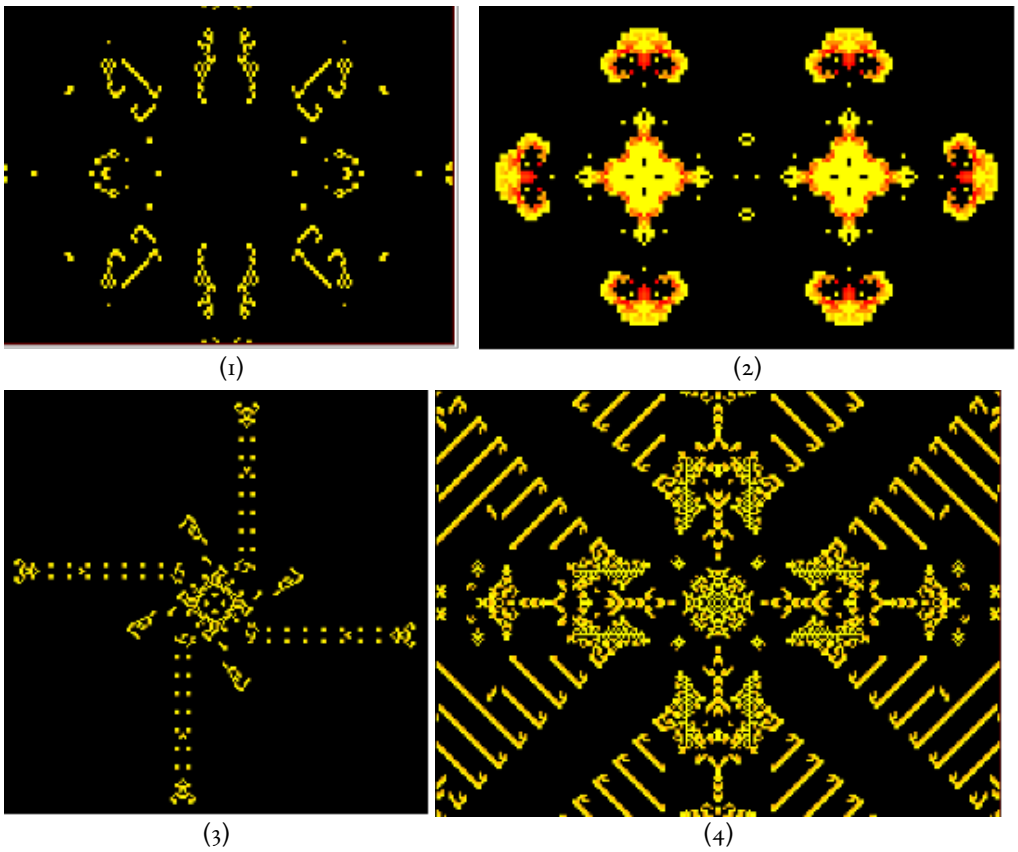


FIGURE 1.4: Four examples of discovered glider patterns with different rules. (Top Left) Brian's Brain with rule "/2/3". (Top Right) Burst with rule "0,2,3,5,6,7,8/3,4,6,8/9". (Bottom Left) Brain6 with rule "6/2,4,6/3". (Bottom Right) Star Wars with rule "3,4,5/2/4".

The gliders in Figure 1.4 belong to the category of outer totalistic generations of two-dimensional cellular automata, which means the state of the cell at time t depends on both its own state and the total of its neighbors at time t. They are generated using Moore neighborhood with the "Instant birth, gradual death, no recovery" model depicted in Figure 1.5.

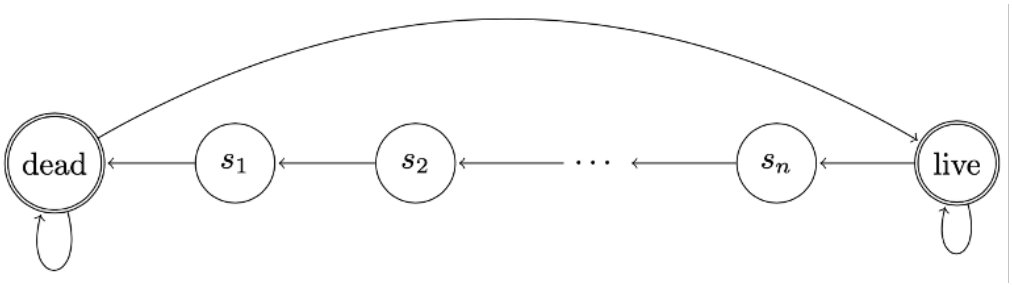


FIGURE 1.5: "Instant birth, gradual death, no recovery" Model. Healthy cells can get sick. Sick cells are not able to recover, and they will be one step closer to death at each step. Dead cells cannot be born sick.

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 which 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, there are three parameters that constitute the rules of the model:

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

The canonical way to represent the evolution rules controlling 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 [13]. 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 1.3. 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 gliders 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. Therefore, we shall only consider gliders for stable rules that exhibit bounded growth and eventu-

ally 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 [12]. However, statistically it has an extremely low probability if the initial configuration is randomly generated. In this paper, we decided to define the rules that satisfy these requirements using the "Instant birth, gradual death, no recovery" model with Moore neighborhood as interesting. Specifically, the definition of interesting rules in this paper includes:

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

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 gliders, are classified as boring. Finding out what the interesting rules are and what the gliders 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, under the assumption that we have a maximum of 10 possible states, there are 29¬ survival rules, 29¬ born rules, which leads to a total of 218¬ combinations of rules, which is virtually an impossible task with normal computer software. Because of this gigantic number of possibilities, automatic detection of interesting rules will be very helpful, which makes sure that users do not have to manually go through the process.

This paper explores the possibility of using deep neural networks to detect these interesting 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 techniques. 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 are 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 of 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 arguably the most tedious and time-consuming step. All we have do eventually is to manually inspect the rules that have been classified as interesting by our machine learner and record the gliders within the patterns if they have been classified correctly.

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 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 another great scientist Stephan Wolfram published "A new kind of science" in 2002, which is also regarded as the encyclopedia of cellular automata [4]. 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 computational universe and the scarcity of the discovered rules gives us the potential to mine the interesting rules and harness for our purposes. Wolfram describes the process of looking for something interesting in the space of cellular automata very different from our accustomed approach of building models step by step while ensuring that we have control over their behaviors. Instead, he makes the rather counter-intuitive claim that we should not try building anything at all and just define what we want and then search for it in the computational universe. It is sometimes very easy and fast to find we want. For example, Wolfram quickly came across with rule 30 [8] 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.

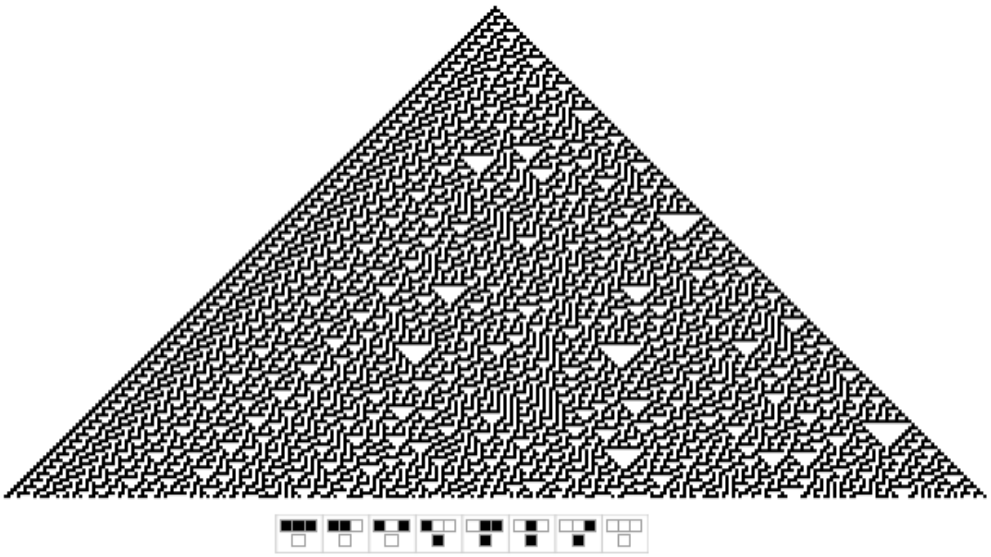


FIGURE 2.1: Wolfram's Rule 30

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 gliders. 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 gliders. 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 gliders.

A more systematic method to find gliders is introduced by Andrew Wuensche in Collision-Based Computing [10]. He proposed that this could be achieved by a measure of the variance of input-entropy over time. The distribution of rule classes in rule-space is discovered. The method also allows automatic "filtering" of cellular automata space-time patterns to show up gliders and related emergent configurations more clearly. Cellular automata dynamics is 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 [14]. 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 Stephan Wolfram, which consists of four different classes: automata in which patterns stabilize into homogeneity, automata in which patterns evolves into mostly stable or oscillating structures, automata in which patterns evolves into chaos, and automata in which patterns become extremely complex [7]. 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 gliders in two-dimensional outer totalistic cellular automata is an unsolved problem and this paper 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 and the interesting rules as living fish, then we are not trying to catch a random fish with a hook, but rather using a large fishing net to quickly scan a vast volume of sea and inspect whatever that looks like a fish. Given sufficient computers and memory, we have the potential to detect all the interesting rules containing gliders.

Interestingly, Wolfram also described an uncanny systematic resemblance between neural networks and cellular automata in his book [6]. The parameters in neural networks are never explicitly set or engineered, but they are generated automatically. Similar thing happens 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 might have to enumerate all possibilities.

3 Methodology

We first collected the two-dimensional cellular automata data which the machine learner could use for training, validation, and testing. This entailed 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 is 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 are passed in as parameters. The specific implementation of the algorithm can be found in 6.1.

We were able to keep track of the evolution of all cells in the grid in each generation using the evolution algorithm, and with the help of a proper visualization tool, we can 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-glider patterns. Enumeration proved to be quite effective because most rules fall under the category of boring, and we easily collected 105 bor-

ing rules using this method. Interesting rules, on the other hand, are much rarer and thus harder to find. Hence, we borrowed existing examples provided in Cellular Automata Rules Lexicon and recorded those with gliders. Eventually, we successfully collected 35 rules that are classified can be subjectively classified as interesting. The set of interesting rules we included in our dataset are described in 6.2.

To obtain the annotator agreement for interesting rules, we calculated Cohen's kappa when labeling rules from lexicons and other sources. These sources contained a total of 147 potentially interesting rules. We both agreed to include 35 of those rules that had clear glider patterns and excluded 107. There were 3 rules member #1 wanted to include and the other did not, and 2 rules member #2 wanted to include that was not included by member #1. This gives us 96.59% agreement and a Cohen's k equal to 0.91, signifying almost perfect or perfect agreement.

Because of the limited number of rules that we classified, we decided to apply data augmentation to increase the size of the dataset. The boring and interesting rules were each reused 10 and 30 times with different random initial configuration. Consequently, we generated a total of 1050 boring and 1050 interesting patterns, making the classes perfectly balanced. For each of the patterns, we recorded 140 consecutive generations as grayscale frames using CellPyLib, which is a python package supporting the visualization of two-dimensional, k-color, adjustable neighbor cellular automata. The living and dead cells are in black and white respectively. The remaining sick cells are assigned with a grayscale color in between depending on their specific state. Figure 3.1 shows an example of a generated frame.

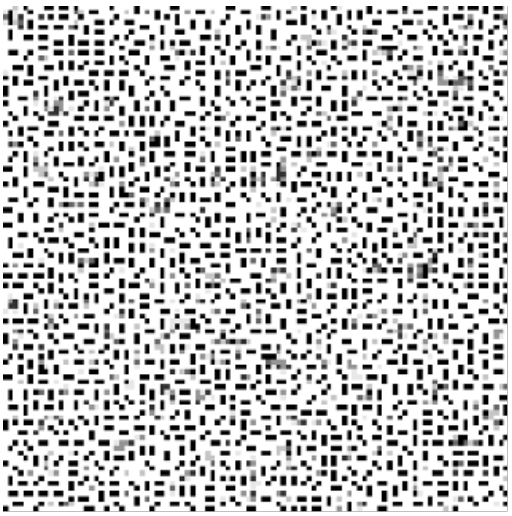


FIGURE 3.1: Frame 107 of a boring rule "1,5,8/2/7" with a random initial configuration.

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 are generated represents the evolution of cellular automata with respect to time, so the frames cannot 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 is 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 gliders 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 [15]. Since the initial configuration is totally random, we believe under most circumstances the starting generations of the cellular automata are highly randomized and will not reflect the

eventual pattern accurately. Therefore, we decided to start training the LSTM at the 80th frame, where the patterns are reasonably solidified. Each sample consists of 41 (from 80th to 120th) consecutive frames and each frame is of size 300 × 300 and has 1 channel as it is grayscale. The 41 selected frames are congregated into a list and the LSTM would process the entire list as one sample. Corresponding code can be found in 6.3. We hoped that the machine learner would consider the existence of gliders 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 were 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 [11]. Our results did improve significantly after we used ConvLSTM2D layer instead stacking a Conv2D layer with an LSTM layer. Furthermore, we also tried different filter sizes, dropout rates, kernel sizes, and activations. Eventually, we used the structure described in FIGURE 3.2 for our machine learner.

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 accuracy 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, and 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 preserve 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 3.3. 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 of the 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. The number of frames is constrained to be a square number because eventually we need to stitch the selected frames into a

square grid. We tested many possible numerical values of the two parameters to discover the best combination. Figure 3.4 shows two examples of the stitched images, one represents a boring rule while the other represents an interesting rule.

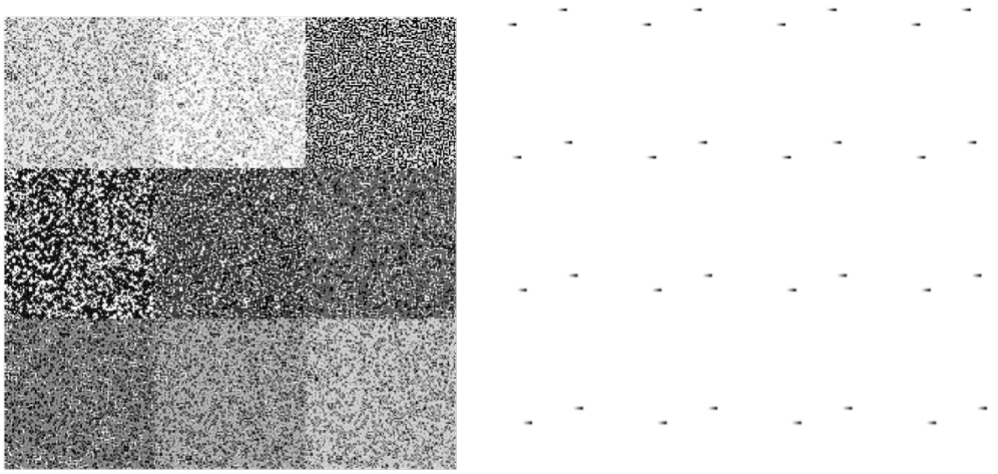


FIGURE 3.3: Examples of stitched up images. (Left) Nine frames are stitched together, which are generated by a boring rule "6/0,5,6,9". All frames are noise. (Right) Sixteen frames are stitched together, which are generated interesting rule "2/2/8". By comparing the frames in the top left and the bottom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

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 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 pre-processing 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. For each of the stitched images, the model returns 1000 selected features. We then fed the data into Brainome.ai and obtained corresponding information about Decision Trees and Neural Networks. Expected generalization using Decision Tree is 2.05 bits/bit and using a Neural Network is 0.19 bits/bit. The decision tree has 1026 parameters, and the estimated memory equivalent capacity for neural networks is 11034 parameters. This overwhelming memory equivalent capacity indicates that the neural network would

be extremely overfitting, which means that the features extracted are 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 the targeted objects.

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 is 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. Running the notebook tended to crash the kernel so we settled for less resolution and down sampled 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.

For our model architecture, we implemented a CNN, which is a logical choice given the task being to classify images. The machine learner aims to predict whether the generated cellular automata will be interesting or boring given a set of rules and an initial configuration. We tried many architectural parameters and hyperparameters to create the best model, including convolutional filter size, dense layers at output, pooling kernel size, type of pooling, dropout, and batch normalization.

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 look 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 FIGURE 3.5.

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 random nature of cellular automata patterns, entropy is an adequate metric to compute since it is likely that there is correlation between the label (boring and

interesting) and the statistical measure of randomness in the images. We used cross-entropy as the default entropy function. 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 3.6.

FIGURE 3.6: Entropy distribution of different 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. There are many which had entropy values close to 0.0, which is reasonable because they would likely correspond to patterns which die out. On the other hand, interesting images have entropy values concentrated in the 0.0 to 2.0 range, especially between 0.5 and 0.75. This is intuitively true because interesting images have 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 boring 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 Glider Activities

After finished training the machine learners, we put them into practice and used them to classify patterns and find gliders in those that are interesting. Since cellular automata with two possible states, like the Game of Life, have already been widely explored and played with, we focused mainly on those with three states. We ran the machine learner on sequence of frames generated random combinations of survival and born rules, and then manually inspect the few rules which the learner classified as interesting.

4.1 Gliders 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", "4,6/2,4/3", "4/2,5/3", "3,6/2,6/3", "5,6/2,6/3". The common trait of these newly discovered rules is that the born rule contains two. These rules all generate the same family of gliders, where the members are all led by a two-by-two glider and followed with a distinct tagalong, where tagalong is defined as a pattern that is not a glider itself but can be attached to the back of a glider to form a larger glider [1]. 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 glider in all patterns with three possible states. It has a period of one and 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 gliders with at most two supplemental blocks. There is one member with no supplemental block attached, namely the leading block, four members with one, and eighteen members with two.

These gliders are shown in Figure 4.1. It is obvious that there are infinitely many one-period members in the family because any arbitrary number of supplemental blocks can be attached. 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 6.12.

However, one caveat is that there are some patterns that have the supplemental blocks attached to the leading block but are not actually gliders. This means that the tagalongs are very delicate and the slightest difference in their structure can lead to 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 glider. Thus, it is very hard to artificially engineer a glider.

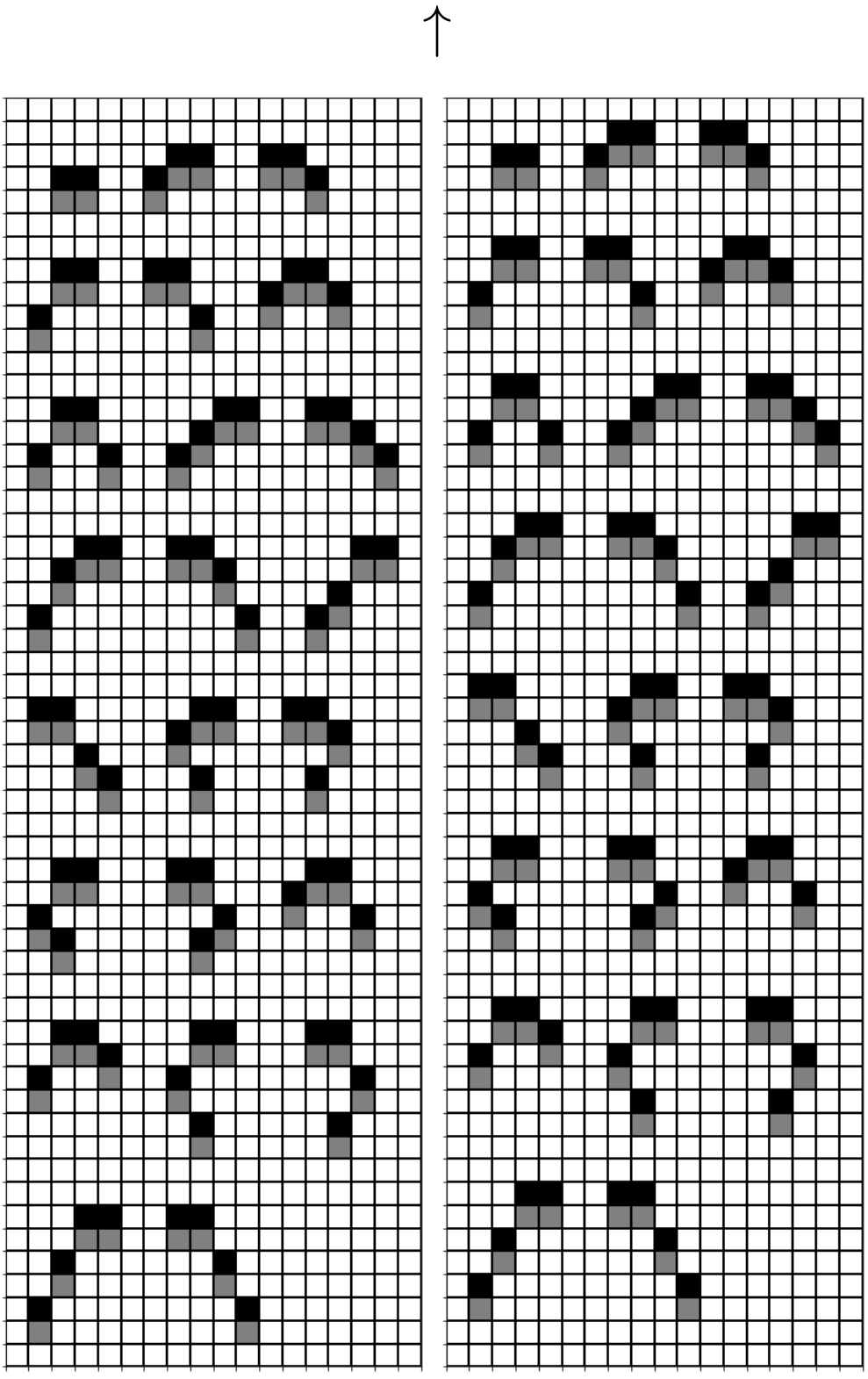


FIGURE 4.1: All one-period members with zero, one, or two supplemental blocks of the glider family that appears in rules "4,6/2/3", "2,4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". Code is provided in 6.12.1.

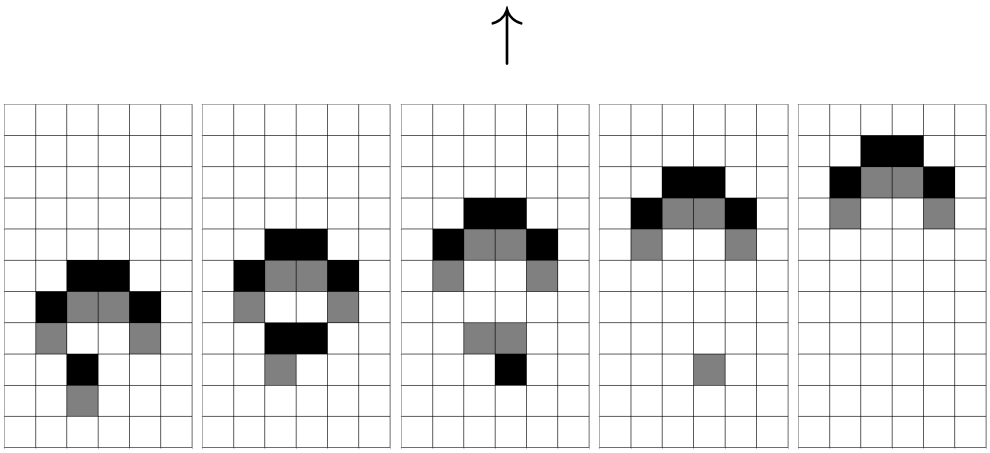


FIGURE 4.2: A pattern that is not a glider in rules "4,6/2/3", "(2)4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3" despite consisting of the leading block and three supplemental blocks. However, it does transform into another two-period glider in four steps. Code is provided in 6.12.2.

We also found many family members with 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 gliders that appear in rules "4,6/2/3", "4,6/2,4/3", "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 glider A, B, C, D, and E" in the rest of the paper. Furthermore, we have also observed some two-period gliders that have the exact same tagalongs as the five basic forms. For example, in Figure 4.5 is another glider that look virtually the same as the "two-period glider E" and only differs in the way that its tagalong is relatively moved towards 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 gliders in the family. Their tagalongs can get arbitrarily large as they can have a connecting bridge consisting of any arbitrary number of supplemental 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 gliders.

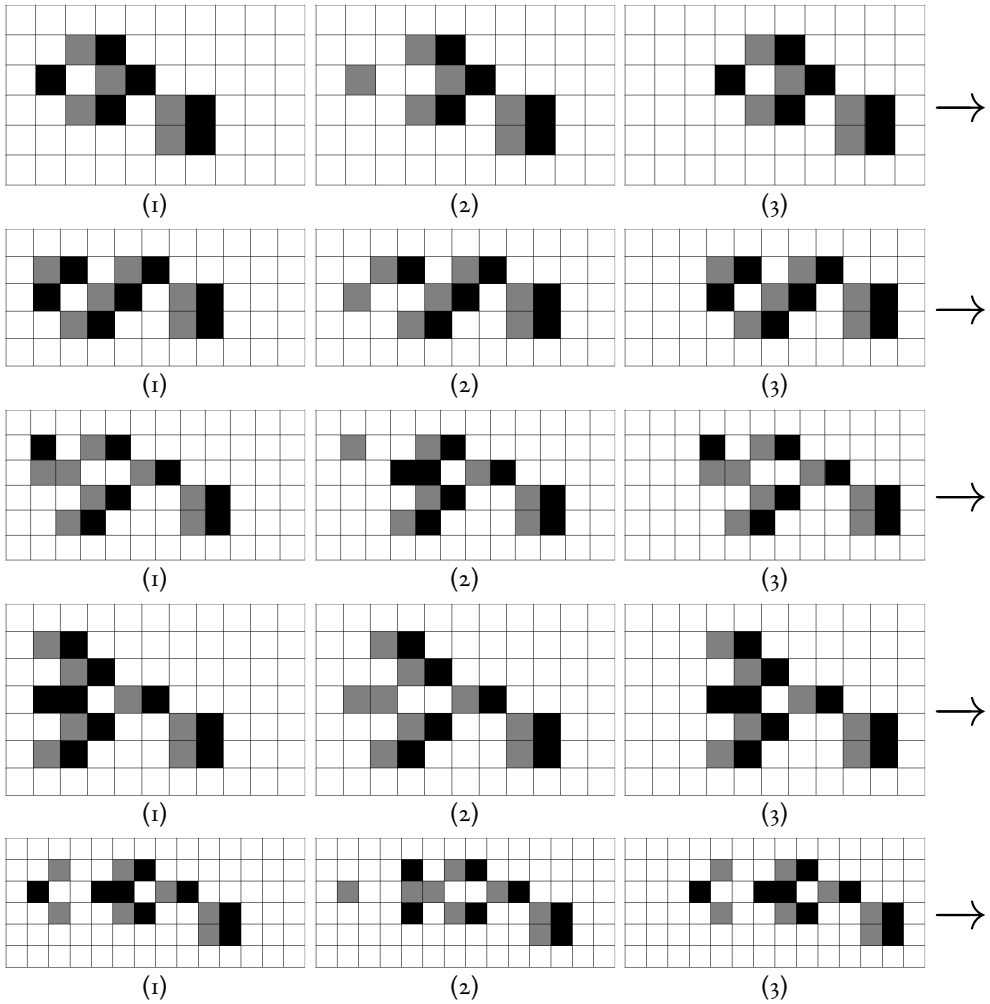


FIGURE 4.3: The five basic two-period members of the glider family that exist for rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they are referred to as "two-period glider A, B, C, D, and E" respectively. Code is provided in 6.12.3, 6.12.4, 6.12.5, 6.12.6, and 6.12.7

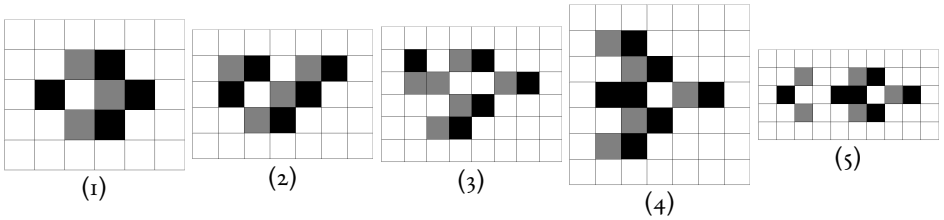


FIGURE 4.4: The five basic tagalongs of two-period gliders.

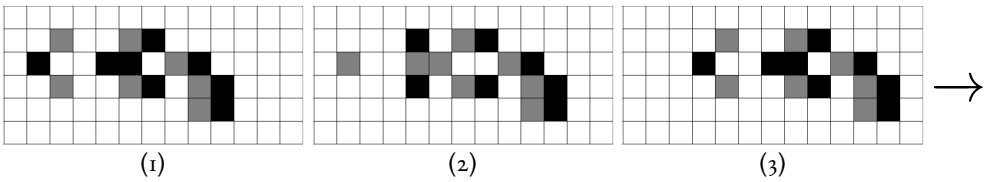


FIGURE 4.5: The glider that is almost the same as "two-period glider E" except its tagalong is moved relatively towards right by one unit. Code is provided in 6.12.8.

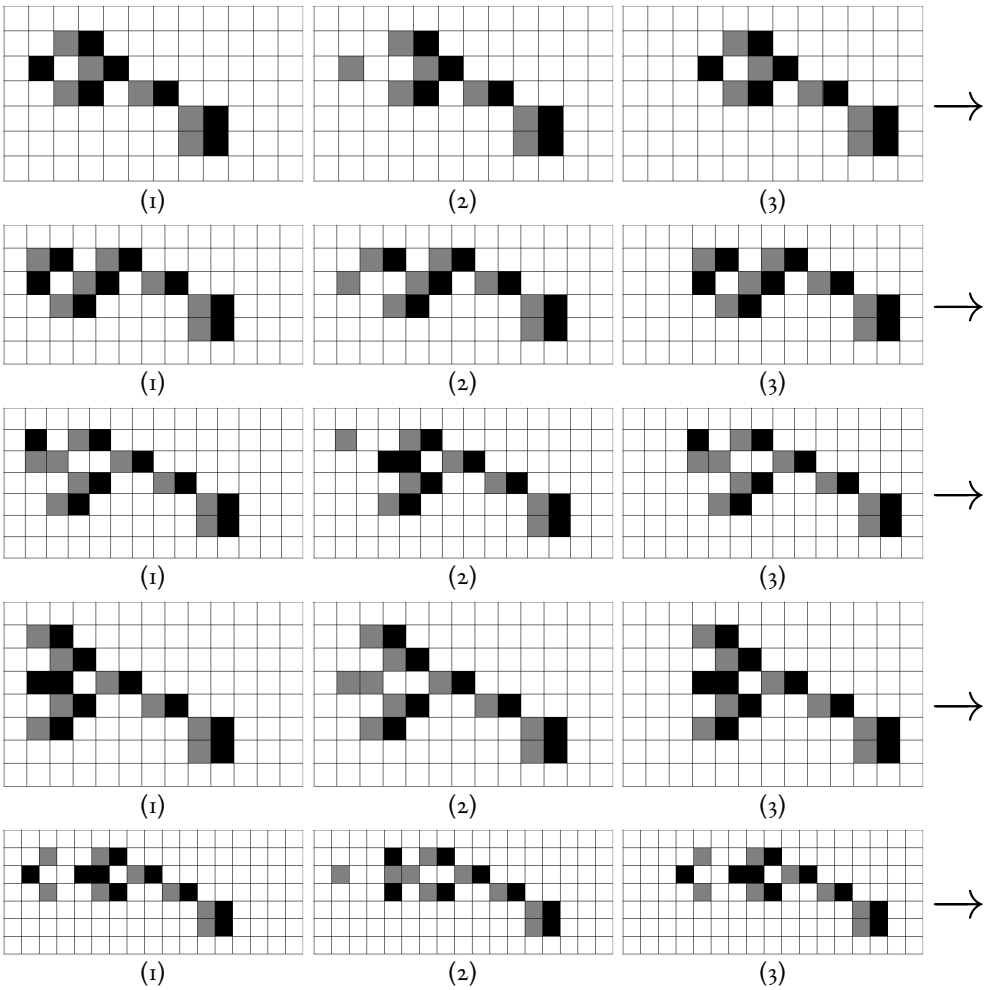


FIGURE 4.6: Examples of extended two-period members of the glider family that exist for rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". Their tagalongs are the same as the five basic forms in Figure 4.4 except there is an additional supplemental block connecting them to the leading block. Other extended members have more connecting supplemental blocks. Code is provided in 6.12.9, 6.12.10, 6.12.11, 6.12.12, and 6.12.13.

We have also found family members with period of four. Superficially, they do not

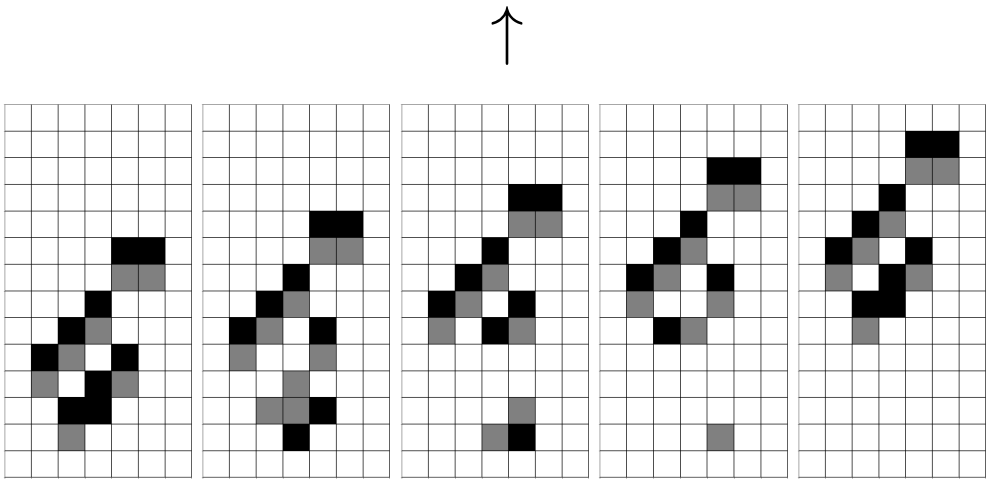
look much different from the two-period members except their tagalongs are larger and more complicated. They emit a more noticeable and larger amount of vanishing exhaust when they move across the grid. We identified six basic four-period members shown in Figure 4.9, each with a distinct tagalong in Figure 4.7. For simplicity, they will be referred to as "four-period glider A, B, C, D, E and F" in the rest of the paper. 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 gliders, which we will name as "eight-period glider A" and "eight-period glider B". Their tagalongs are depicted in Figure 4.8. Superficially, there is an uncanny resemblance between "eight-period glider A" and "four-period glider C", and "eight-period glider B" and "four-period glider A". The two eight-period gliders 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 much rarer and thus much harder to find than the other members in the family. With the same argument we made with the two-period gliders, there are infinitely many four and eight-period gliders in the family.

The family members introduced so far have only one tail. However, these gliders, 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 some of the basic tagalongs to form a larger one. Figure 4.11 shows two Frankenstein gliders whose tagalong is a combination of the basic forms we have introduced earlier. One combines "two-period glider C" and "four-period glider A". The other combines "two-period glider A" and two "four-period glider B" s. Their existence prove that two or more tagalongs can be combined to form a larger tagalong. The period of the resulting Frankenstein glider 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 it is impossible to enumerate all possible tagalongs.

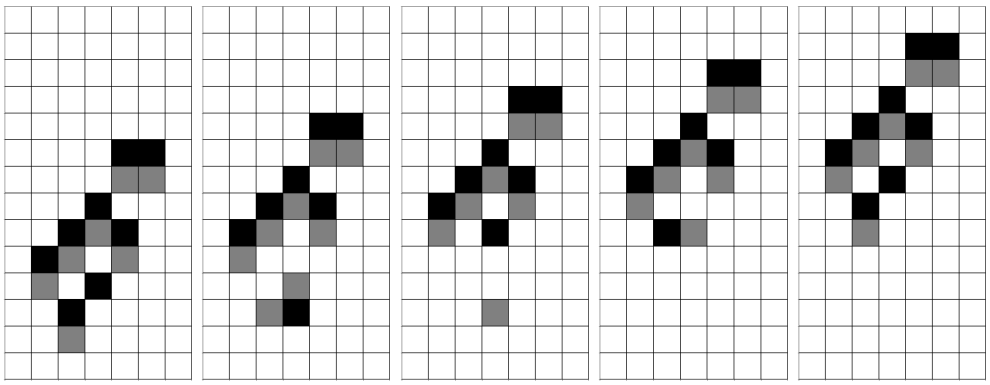
The most interesting family members we found are rakes, which are cellular automata

that leaves behind a trail of non-vanishing debris of a stream of gliders [2]. 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 4.12, Figure 4.13, and Figure 4.14. For simplicity, they will be referred to as "rake A, B, and C" in the rest of the paper. Their periods are equivalent to the number of steps it takes them to generate a new glider. All three rakes generate a stream of one-period gliders, 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 moves perpendicular to the rake.

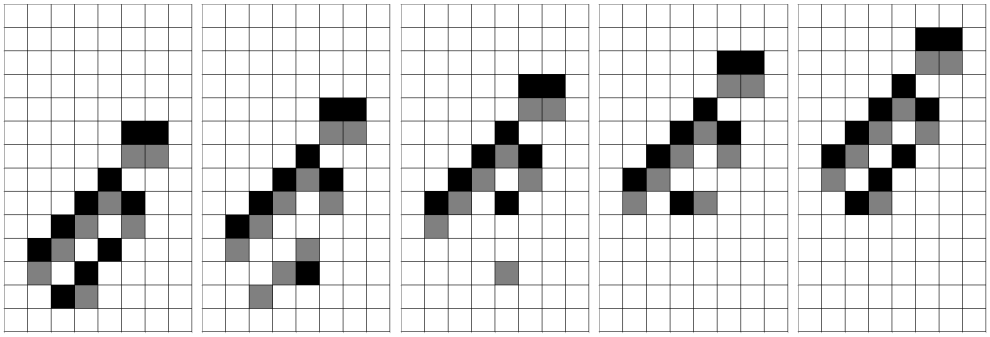
We discovered one special glider depicted in Figure 4.15 that does not contain the leading block and hence is not a member of the family. It maintains a total of three live and three sick cells in all the generations. Furthermore, the gliders we have introduced so far either traverse horizontally or vertically. But just like Conway's Game of Life glider, this glider moves diagonally across the grid with a period of four. Its speed is also 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.



(1) (2) (3) (4) (5)



(1) (2) (3) (4) (5)



(1) (2) (3) (4) (5)

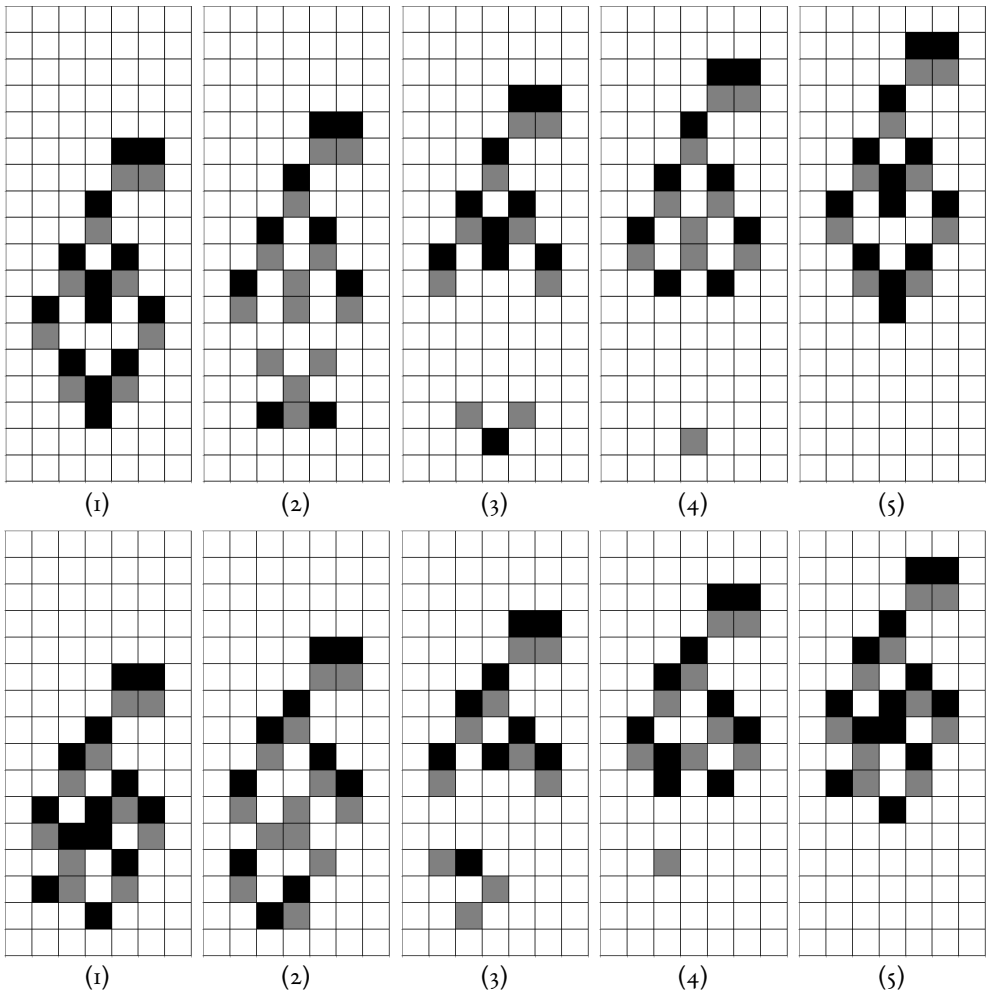


FIGURE 4.7: The six basic four-period members of the glider family. All these gliders exist for rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3", except the last two ones which do not exist for rules "4,6/2,4/3" and "4,6/2,4/3". For simplicity, they are referred to as "four-period glider A, B, C, D, and E" respectively. Code is provided in 6.12.14, 6.12.15, 6.12.16, 6.12.17, 6.12.18, and 6.12.19.

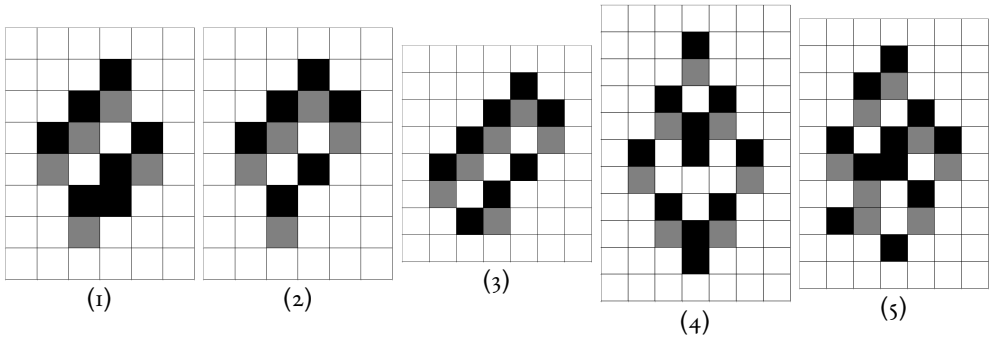


FIGURE 4.8: The basic tagalongs of the five four-period gliders.

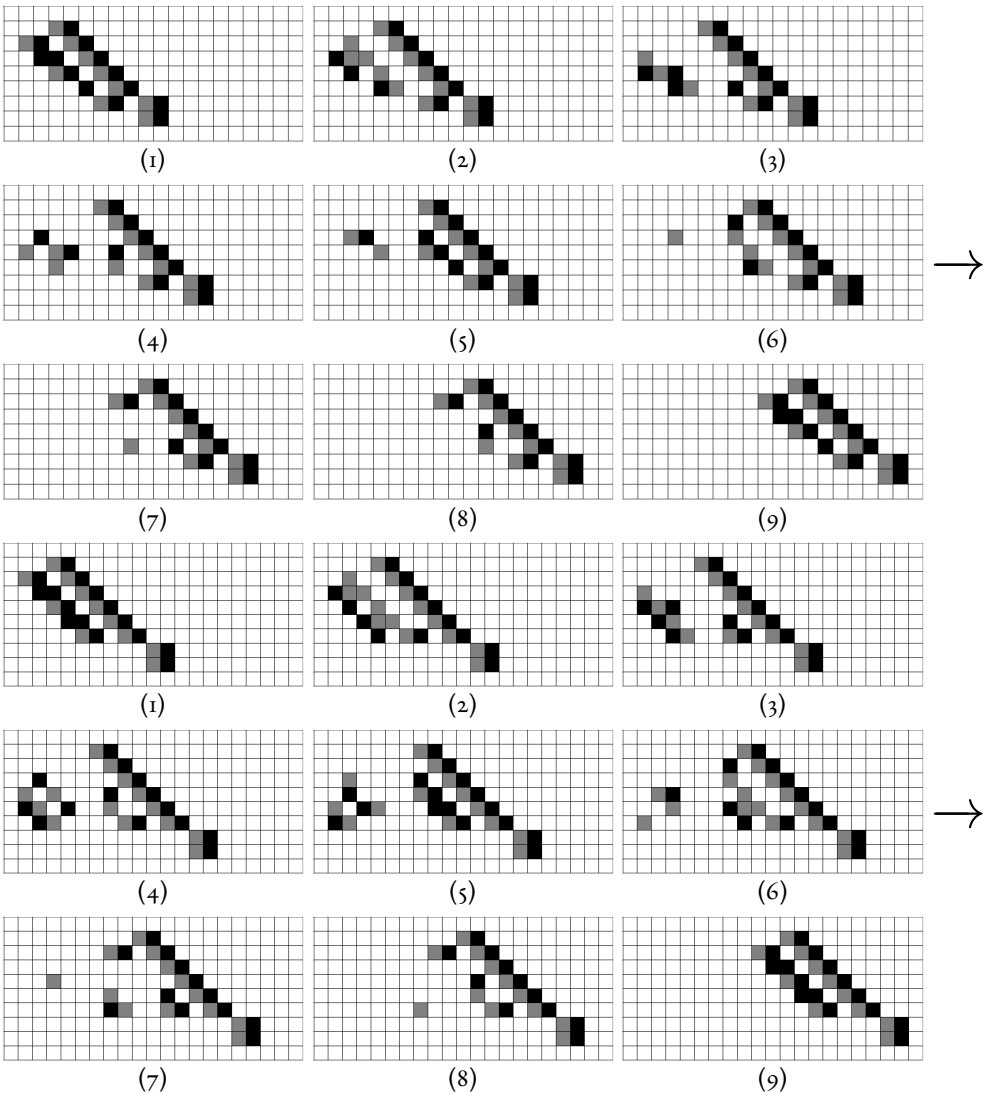


FIGURE 4.9: Two eight-period members of the glider family that exists for rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, they will be referred to as "eight-period glider A" and "eight-period glider B". They resemble "four-period glider C" and "four-period glider A" respectively. They both emit a maximum of ten units' exhaust. Code is provided in 6.12.20, and 6.12.21.

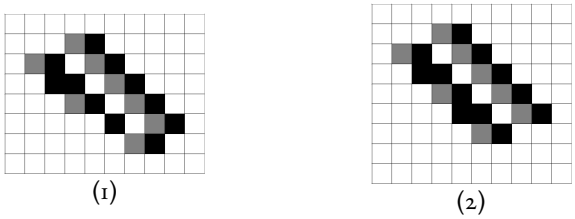


FIGURE 4.10: The basic tagalongs of the two eight-period gliders.

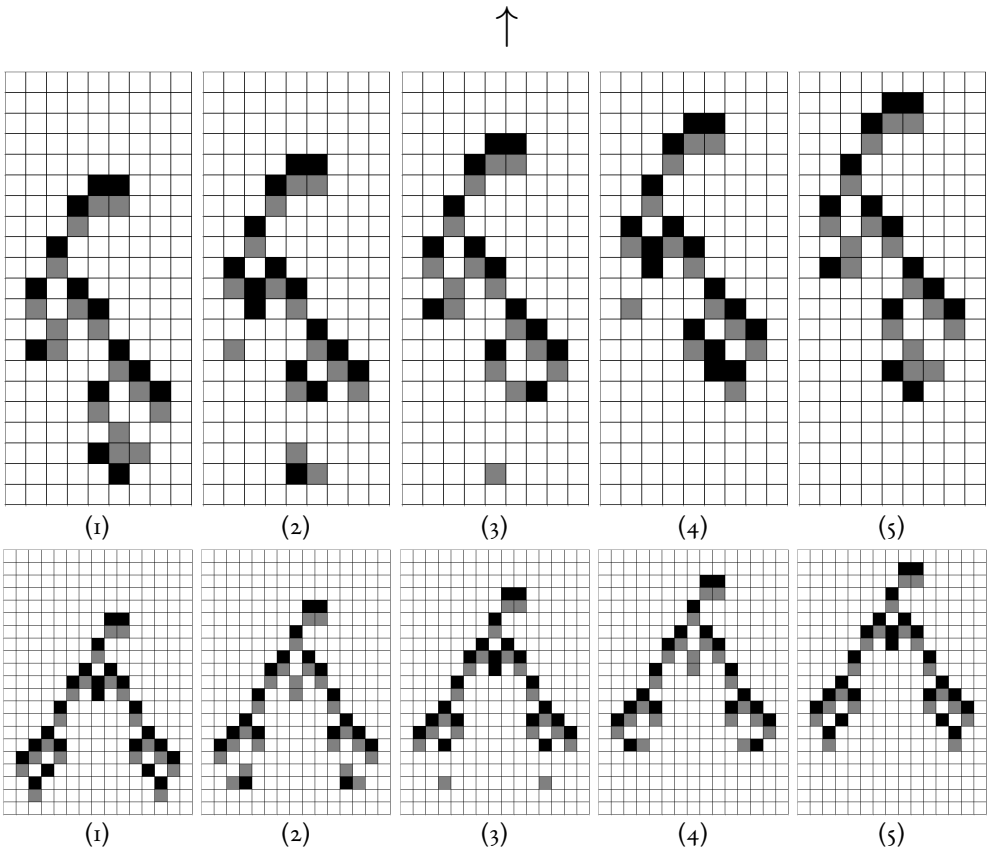


FIGURE 4.11: Two Frankenstein gliders with two and three tails that exists for rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". Both have periods of four. The top combines "two-period glider C" and "four-period glider A". The bottom combines "two-period glider A" and two "four-period glider B" s. Code is provided in 6.12.22, and 6.12.23.

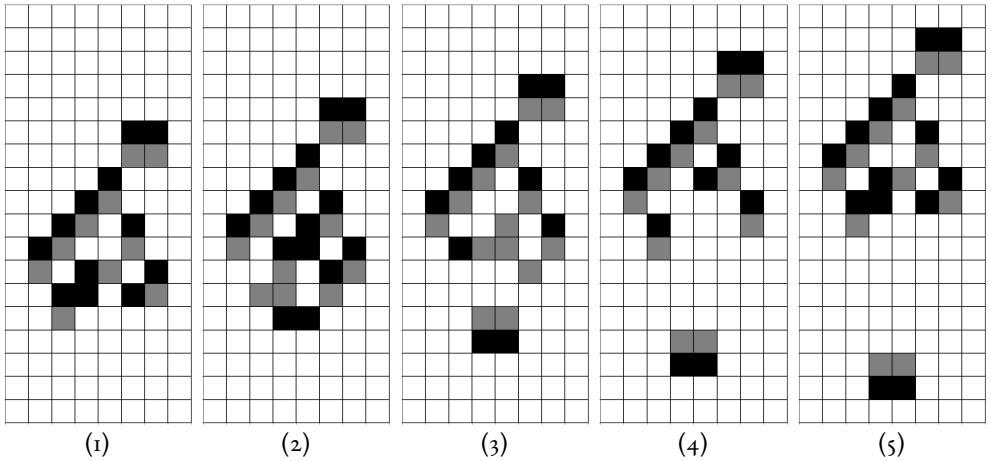


FIGURE 4.12: A four-period rake that exists for rules "4,6/2/3", "3,6/2,6/3", "5,6/2,6/3". For simplicity, it will be referred to as "rake A". It generates a leading block every four steps. Code is provided in 6.12.24.

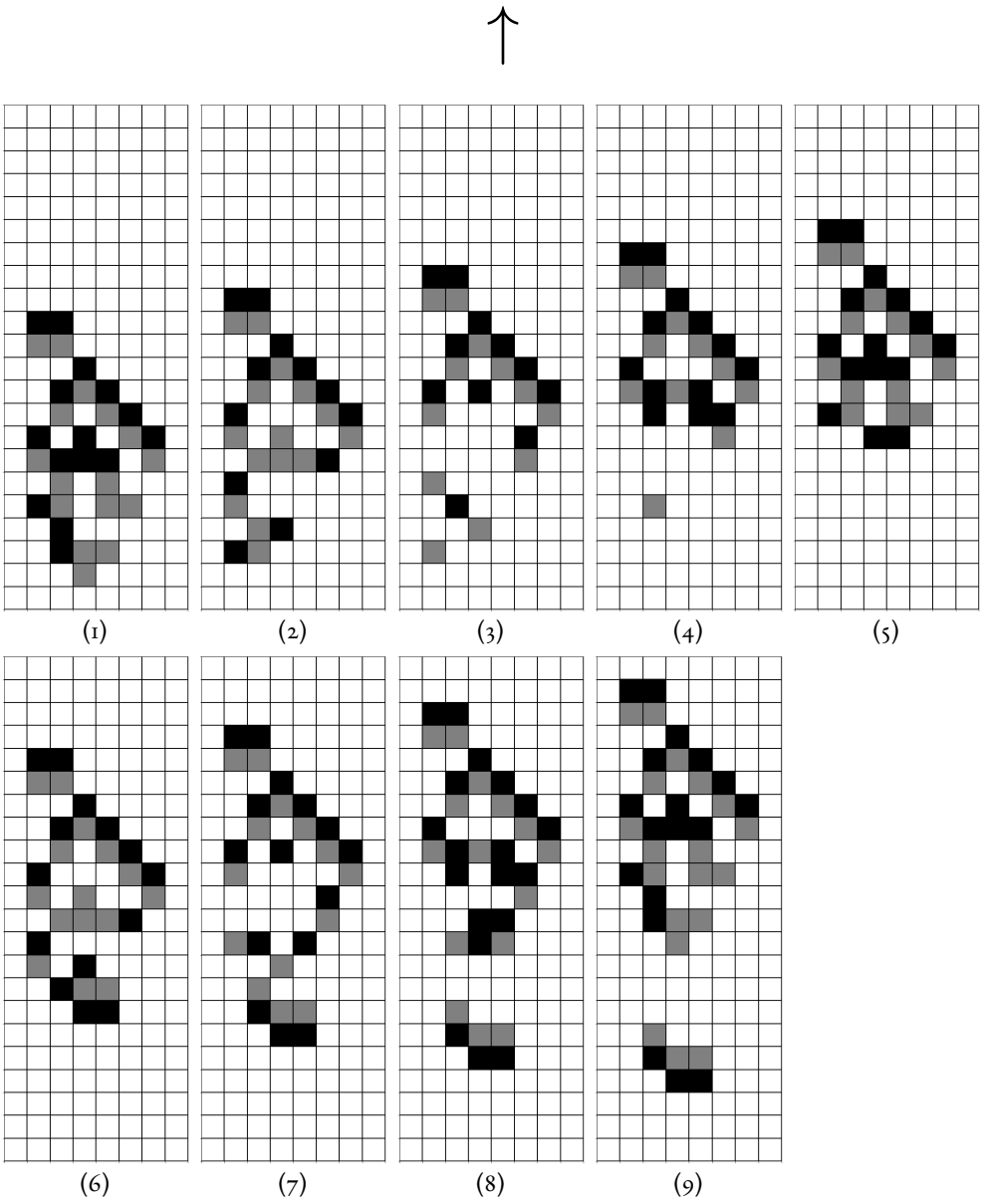


FIGURE 4.13: An eight-period rake that exists for rules "6/2/3", "4,6/2/3", "5,6/2,6/3". For simplicity, it will be referred to as "rake B". It generates a one-period family member with one supplemental block every eight steps. Code is provided in 6.12.25.

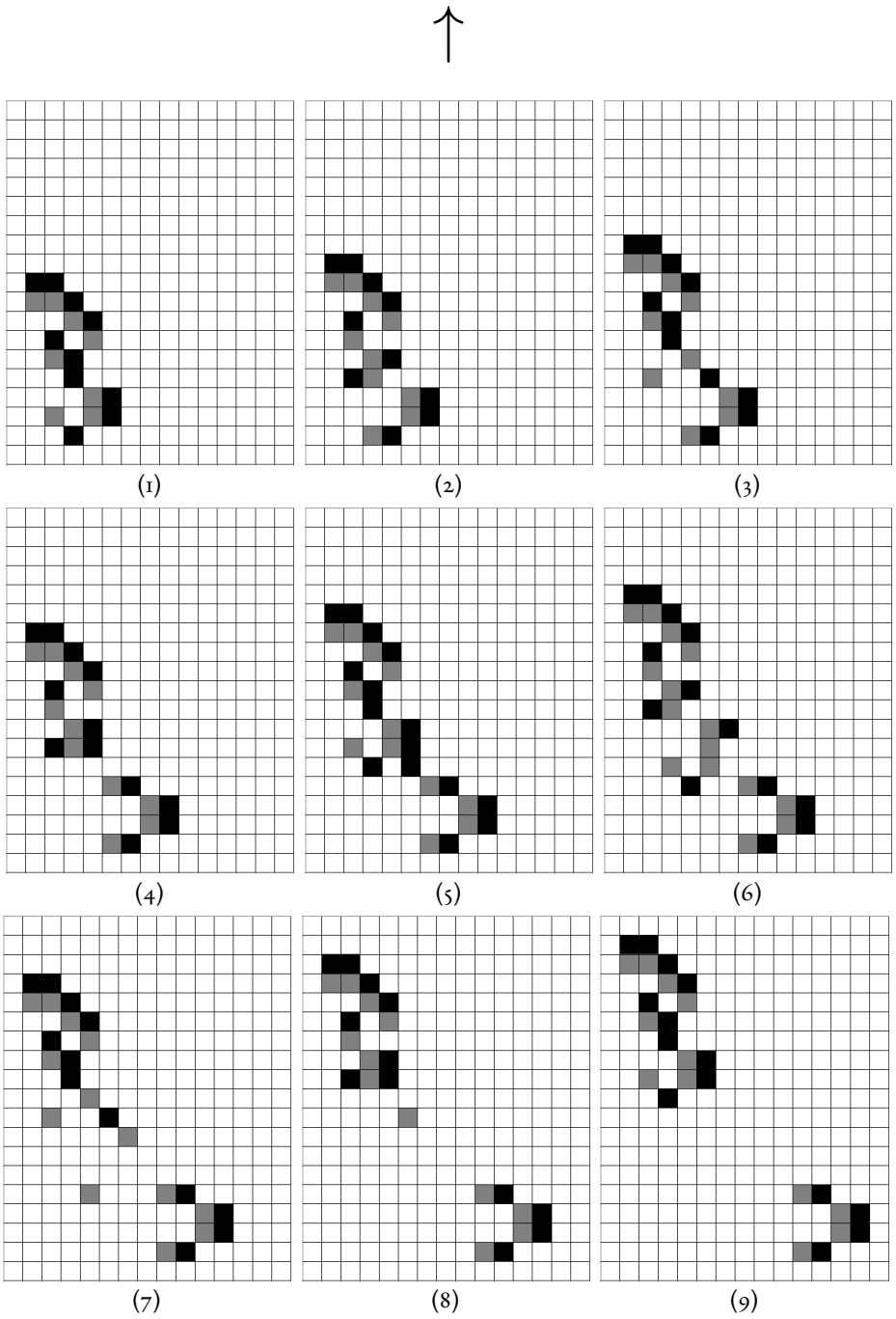


FIGURE 4.14: An eight-period rake that exists for rule "6/2,4,6/3". For simplicity, it will be referred to as "rake C". It generates a one-period member with two supplemental blocks every eight steps. Code is provided in 6.12.26.

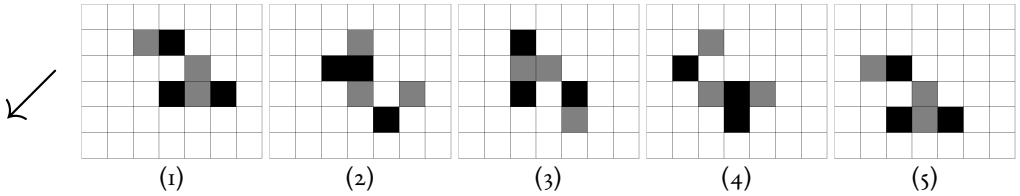


FIGURE 4.15: The new Life with three states that exists for rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3", and "3/2, 5/3". It moves diagonally with a speed of $c/4$. Code is provided in 6.12.27.

4.2 Glider Collision Behaviors in three-state cellular automata

Collisions between gliders are very usual 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 4.16, the two gliders are simultaneously destroyed completely in the collision, leaving the grid empty.
2. Shown in Figure 4.17, the collision generates a new glider that has a different structure.
3. Shown in Figure 4.18, one of the two gliders is "murdered" during the process, which means exactly one glider survive unscathed after the collision, while the other completely disappears.

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

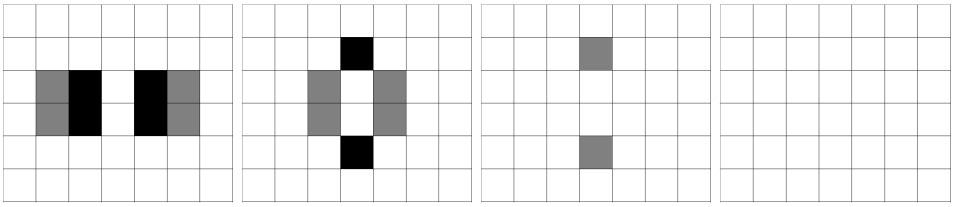


FIGURE 4.16: Two leading blocks in rule "4,6/2/3" collide with each other head on and are both destroyed in three steps, leaving the grid completely empty. Code is provided in 6.12.28.

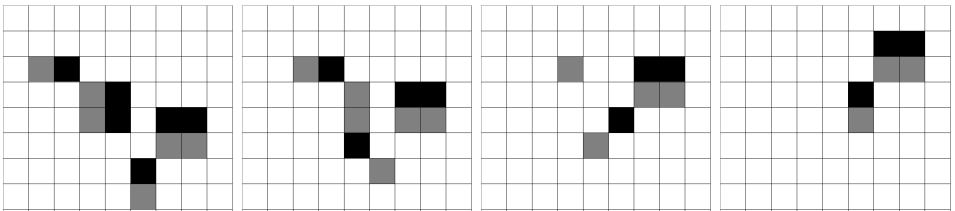


FIGURE 4.17: Example of a "Murdered Glider" that appears in rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". Two identical 1-period members of the glider family with one supplemental block collide with each other, and only the bottom one survived the clash. Code is provided in 6.12.29.

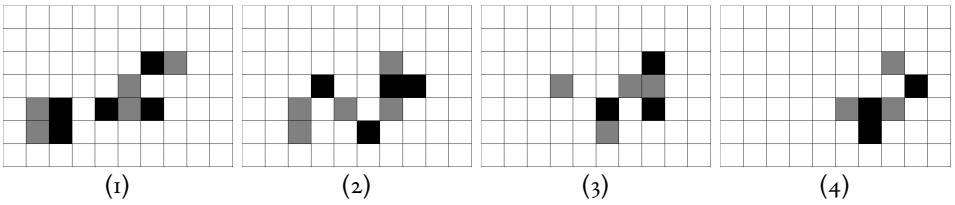


FIGURE 4.18: Example of a "Murdered Glider" that appears in rules "4,6/2/3", "4,6/2,4/3", "4,6/2,4/3", "3,6/2,6/3", "5,6/2,6/3". A fundamental block glider collides with a New Game of Life glider. The fundamental block is murdered in three steps. Code is provided in 6.12.30.

4.3 Other interesting discoveries

As the number of states increases, the complexity of the gliders increases correspondingly. In cellular with four possible states, we have found a more visually appealing rake like a spaceship emitting gas in FIGURE 4.19. The rake has a period of two and a speed of c. It generates a stream of one-period gliders, one every two steps.

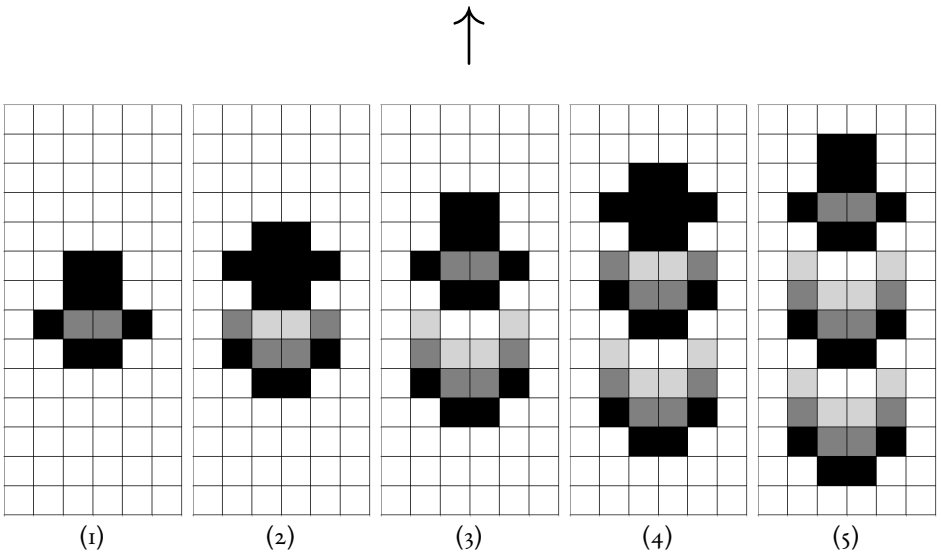


FIGURE 4.19: A rake that appears in rule "3,4,5/2/4". It has a period of two and a speed of c. It generates a stream of one-period gliders, one every two steps. Code is provided in 6.12.31.

We have also identified similar interactions between gliders when they collide. There are countless cases where the two gliders are both destroyed during the process. On the other hand, the other two cases are much rarer. We recorded two such cases in Figure 4.20 and Figure 4.21.

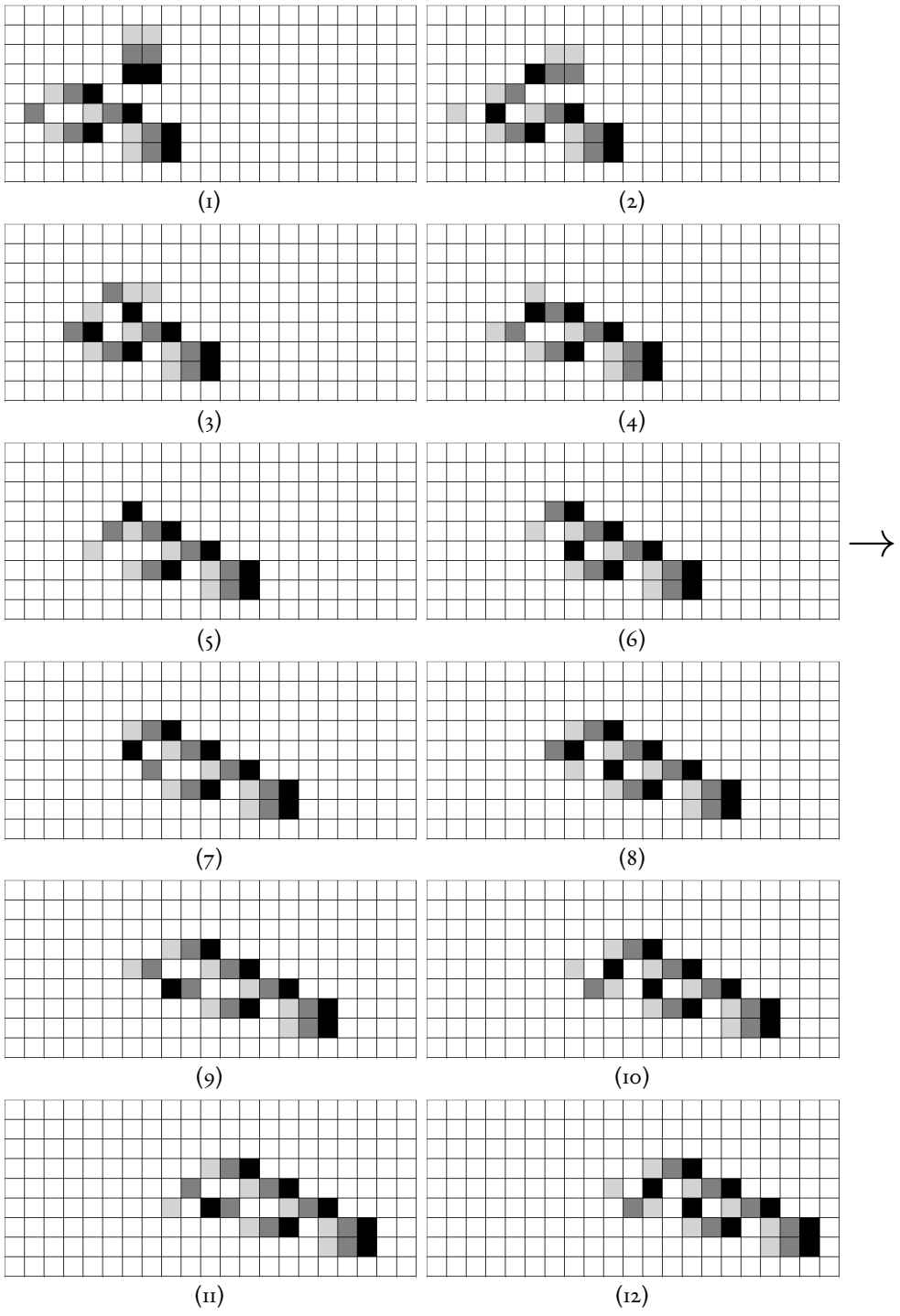


FIGURE 4.20: Example of an "Combined Glider" that appears in rule "3/2/4". Code is provided in 6.12.32.

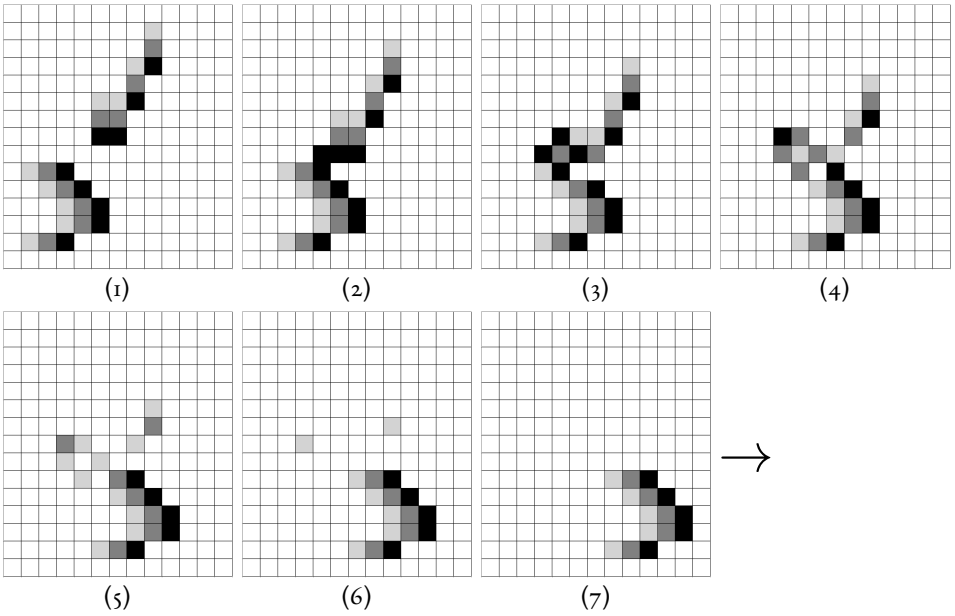


FIGURE 4.21: Example of a "Murdered Glider" that appears in rule "3/2/4". The bottom left glider murders the top right one in six steps. Code is provided in 6.12.33.

5 Conclusion

5.1 Experiment Summary

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 gliders is pertinent to the continued development of cellular automata theory. Gliders have piqued much interest since it literally represents life and proves to be beneficial to many biology and physics models. 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 relative fast but with a poor accuracy. This paper explores the possibility of using neural networks to find interesting rules in the "instant birth, gradual death, no recovery" model using 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 rerunning the rules with different initial configurations. Each of the outcomes is recorded using the python package CellPyLib as a sequence of grayscale frames. The first several generations are ignored because they depend highly on the initial configuration and therefore do not reflect the final stable pattern accurately. With many possible ways to play with the dataset, this paper introduces RNN, CNN, feature extraction, and entropy analysis.

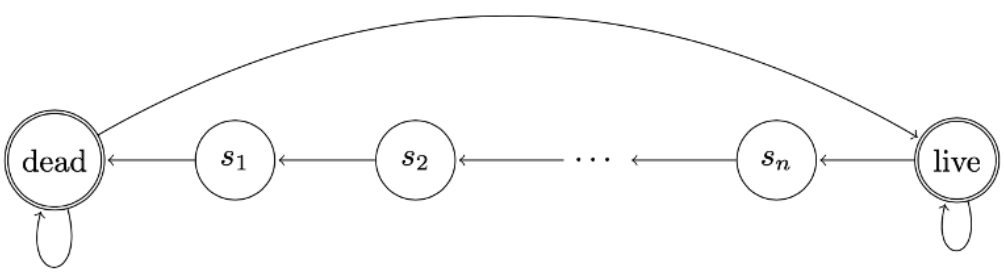
The RNN approach resembles a video classification task. Each pattern is represented by its corresponding sequence of frames and each sequence is treated as one single

instance. We performed hyperparameter tuning and eventually decided to use ConvLSTM2D, which leads to the best accuracy. CNN on the other hand is the best at classifying images. Therefore, we had to stitch some of the frames together into one congregated image in advance. Each of the stitched images is considered as an instance. Different selections of frames that are included in the image have been tested. However, since this is an uncononical approach, we wanted to find out the learnability of our dataset using Brainome.ai. To achieved 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 put them in an organized CSV file. However, the results shown by Brainome.ai was suboptimal. Therefore, we eventually used the image pixels directly as features and this time, Brainome.ai showed us much 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 survial and born rules, and find gliders in those that are classified as interesting. We discovered a handful of new interesting rules, and found an entire family of gliders, 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 was undoubtedly interesting and worth further exploration.

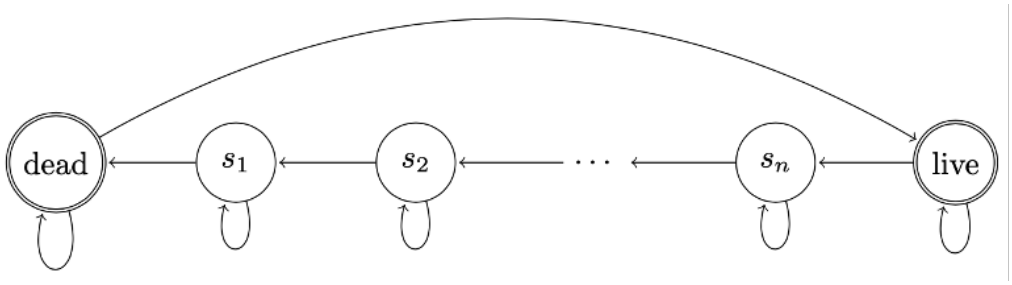
5.2 Future Work

Automatic detection of interesting cellular automata is a project that has not yet been successfully done in the past. This means this paper is breaking new grounds and has potential to explore many alternative possibilities. As mentioned in the introduction section, this paper 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 fell sick, can never recover and can only be one step closer to death at each generation. However, there are many other viable models in Figure 5.1

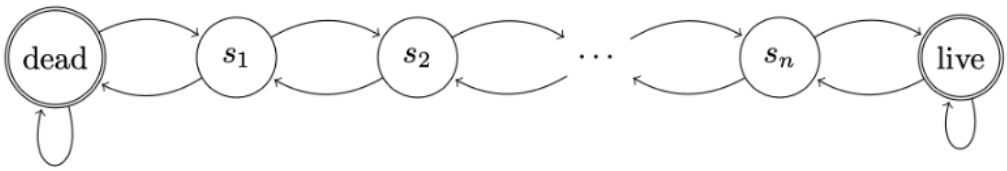
where the sick cells can have other outcomes and additional rules are required to define the achievable behavior of the sick cells. We can artificially engineer the models to simulate various real-life scenarios and harness for our purposes. One simple example is that if we want the cells to imitate 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 like Covid-19, which is able to murder an apparent 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 paper is exploring within only one model, but there are many other possibilities we have not even enumerated, all of which are worth exploring and it would be very interesting to find out what the gliders look like in these models.



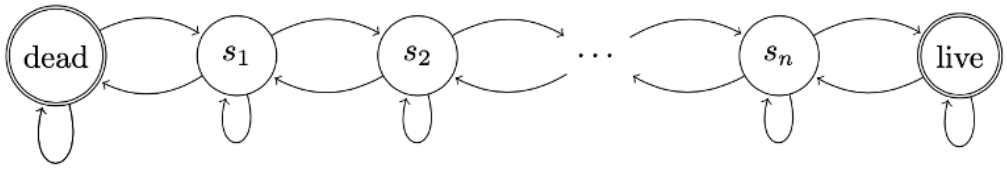
(1) "Instant birth, gradual death, no recovery" Model. Live (healthy) cells can get sick. Sick cells are not able to recover, and they will be one step closer to death at each step. Dead cells are always born healthy and cannot be born sick.



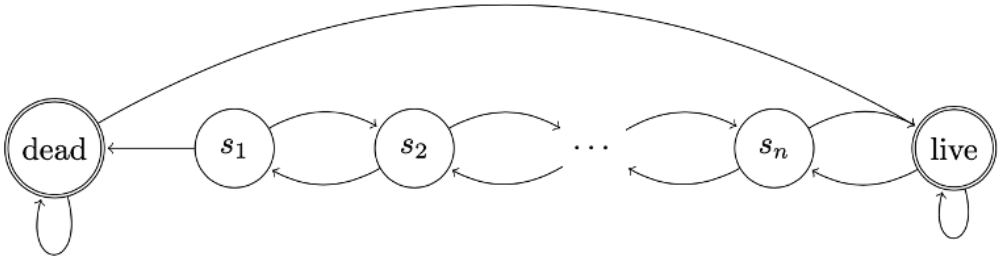
(2) "Instant birth, gradual death, stay sick, no recovery" Model. Live cells can get sick. Sick cells are not able to recover, and they will be either one step closer to death or stay the same at each step. Dead cells cannot be born sick.



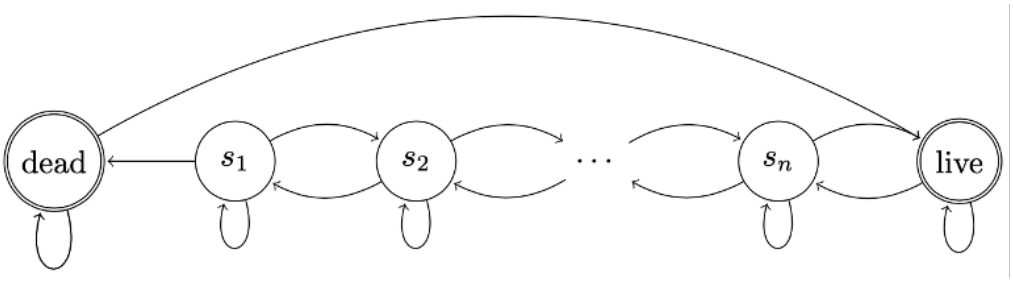
(3) "Gradual birth, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life at each step. Dead cells will be born sick.



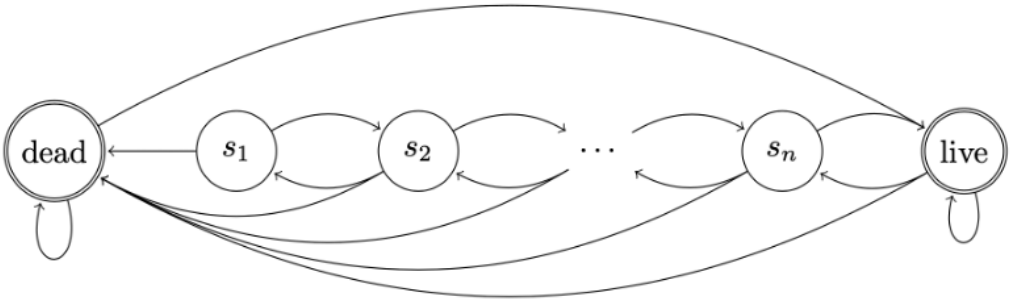
(4) "Gradual birth, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life or stay the same at each step. Dead cells will be born sick.



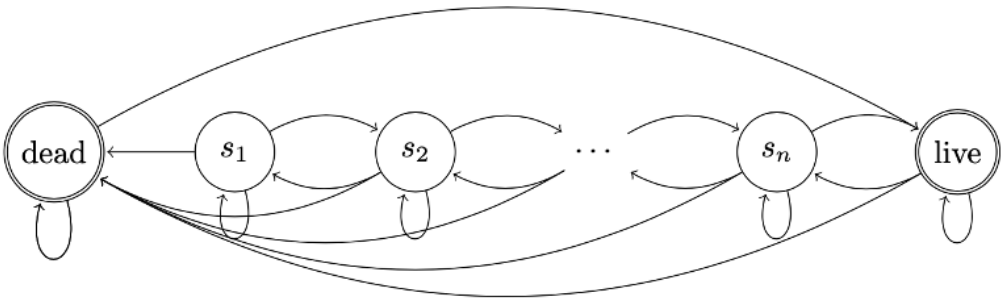
(5) "Instant birth, gradual death, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they will be one step closer to either death or life at each step. Dead cells cannot be born sick.



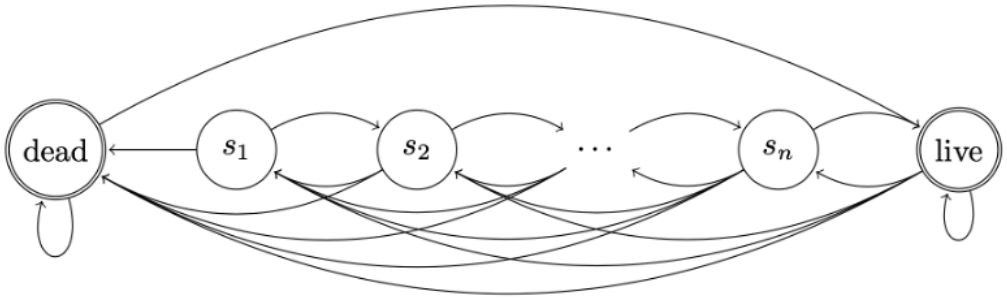
(6) "Instant birth, gradual death, stay sick, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they will be one step closer to death or life or stay the same at each step. Dead cells cannot be born sick.



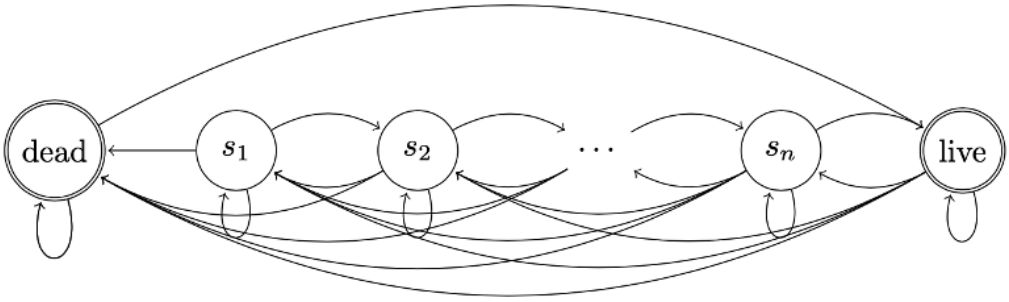
(7) "Instant birth, gradual and instant death, gradual recovery" Model. Live cells can get sick. Sick cells can recover, and they can be one step closer to either death or life at each step. But they can also die immediately in the next step, imitating sudden death in real life. Dead cells cannot be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

ploring, there is still a huge potential search space. This paper detailly explores cellular automata with three states: the live (healthy) state, the dead state and one sick state. We have identified a family of gliders and the new Life, but there could be more which 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 gliders would look like, or if there is any correlation between the rule and the structure of the gliders. It is expected that as the number of sick cells increases, so would the complexity of the patterns and gliders. By the beautiful rocket-like rake in Figure 4.19, we think it is highly likely that there might be more visually appealing gliders with more states. Furthermore, this paper focuses solely on gliders and only classifies those rules that generate gliders as interesting. However, there are many other interesting patterns in cellular automata that is 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 the real life is arguably particles and their behavior is controlled by temperature, which acts like the neighborhood rules in the space 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 the random distribution temperature.

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. 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.

6 Appendix

6.1 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
```

6.2 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)
```

6.3 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

6.4 RNN model structure

```
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"])
```

6.5 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 save_DIR
```

6.6 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)
```

6.7 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])
```

6.8 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"])
```

6.9 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)
```

6.10 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
```

6.11 Glider Image and GIF generation

Cellular automata generations with three or four possible states can be visualized using the code below. The initial configuration, the survival rule, the born rule, and the desired number of periods are the required parameters to be passed in.

```
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]
                    if data[i][j] == max_state and count not in
        survival_arr:
                        copy[i][j] = data[i][j] - 1
                    elif data[i][j] == 0 and count in born_arr:
                        copy[i][j] = max_state

filenames = []
length = len(data[0])
width = len(data)

for step in range(period):
        if number_states == 3:
            cmap = colors.ListedColormap(['white', 'gray', '
        black'])
            bounds = [-0.5,0.5,1.5,2.5] # White: 0, Gray: 1,
        Black: 2
        elif number_states == 4:
            cmap = colors.ListedColormap(['white', 'lightgray',
        'gray', 'black'])
```

```
        # White:0, lightgray:1, Gray:2, Black:3
         bounds = [-0.5,0.5,1.5,2.5,3.5]
    norm = colors.BoundaryNorm(bounds, cmap.N)
    fig, ax = plt.subplots()
    ax.imshow(data, cmap=cmap, norm=norm)

    # draw gridlines
    ax.grid(which='major', axis='both', linestyle='-', color
   ='k', linewidth=2)
    ax.set_xticks(np.arange(-.5, length, 1));
    ax.set_yticks(np.arange(-.5, width, 1));

    frame1 = plt.gca()
    frame1.axes.xaxis.set_ticklabels([])
    frame1.axes.yaxis.set_ticklabels([])
    plt.rcParams["figure.figsize"] = (20,20)
    plt.savefig(f'{step}.png', bbox_inches='tight')
    filenames.append(f'{step}.png')

    plt.tick_params(axis = "x", which = "both", bottom =
   False, top = False)
    data = evolve(data, survival_arr, born_arr)

with imageio.get_writer('mygif.gif', mode='I', duration =
   0.5) as writer:
    for filename in filenames:
        image = imageio.imread(filename)
        writer.append_data(image)
```

6.12 Initial configuration of the gliders

This section contains a list of initial configurations and the corresponding parameters of the gliders mentioned in this paper. One can pass these as parameters into the function in 6.11 to reproduce the images and gif.

6.12.1

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

6.12.2

```
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]
]
```

6.12.3

```
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]
]
```

6.12.4

```
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]
]
```

6.12.5

```
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, 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]
]
```

6.12.6

```
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]
]
```

6.12.7

```
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]
]
```

6.12.8

```
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]
]
```

6.12.9

```
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]
]
```

6.12.10

```
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]
]
```

6.12.11

```
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]
]
```

6.12.12

```
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]
]
```

6.12.13

```
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]
]
```

6.12.14

```
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]
]
```

6.12.15

```
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]
]
```

6.12.16

```
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]
]
```

6.12.17

```
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]
]
```

6.12.18

```
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]
]
```

6.12.19

```
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]
]
```

6.12.20

```
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]
]
```

6.12.21

```
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]
]
```

6.12.22

```
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]
]
```

6.12.23

```
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]
]
```

6.12.24

```
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]
]
```

6.12.25

```
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]
]
```

6.12.26

```
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]
]
```

6.12.27

```
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]
]
```

6.12.28

```
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]
]
```

6.12.29

```
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]
]
```

6.12.30

```
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]
]
```

6.12.31

```
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]
]
```

6.12.32

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

6.12.33

```
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, 0, 1, 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, 1, 3, 0, 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]
]
```

7 Bibliography

1. Spaceship (cellular automaton). (2021, April 23). In Wikipedia. https://en.wikipedia.org/wiki/Spaceship_(cellular_automaton)
2. Rake (cellular automaton). (2021, January 13). In Wikipedia. https://en.wikipedia.org/wiki/Rake_(cellular_automaton)
3. Speed of light (cellular automaton). (2020, April 12). In Wikipedia. https://en.wikipedia.org/wiki/Speed_of_light_(cellular_automaton)
4. Wolfram Stephan (2002) A New Kind of Science. Wolfram Media, Champaign Il
5. Bays C. (2009) Gliders in Cellular Automata. In: Meyers R. (eds) Encyclopedia of Complexity and Systems Science. Springer, New York, NY. https://doi.org/10.1007/978-0-387-30440-3_249
6. The Game of Life. (2021, April 21). In Wikipedia. https://en.wikipedia.org/wiki/The_Game_of_Life
7. Cellular automaton. (2021, April 26). In Wikipedia. https://en.wikipedia.org/wiki/Cellular_automaton
8. Rule 30. (2021, January 13). In Wikipedia. https://en.wikipedia.org/wiki/Rule_30
9. Glider (Conway's Life). (2021, April 20). In Wikipedia. https://en.wikipedia.org/wiki/Glider_(Conway's_Life)
10. Wuensche A. (2002) Finding Gliders in Cellular Automata. In: Adamatzky A. (eds) Collision-Based Computing. Springer, London. https://doi.org/10.1007/978-1447l-0129-1_13
11. Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, Wang-chun Woo. (2015) Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting. arXiv:1506.04214 [cs.CV]
12. Bays C. (2009) Gliders in Cellular Automata. In: Meyers R. (eds) Encyclopedia of Complexity and Systems Science. Springer, New York, NY. https://doi.org/10.1007/978 0 387 30440 3_249

13. Cellular Automata Lexicon http://psoup.math.wisc.edu/mcell/rullex_gene.html
14. David J.Eck. Department of Mathematics and Computer Science, Hobart and William Smith Colleges. Introduction to The Edge of Chaos http://math.hws.edu/xJava/CA/EdgeOfChaos.html
15. Christopher Olah. (5.17.2015) Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/