Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Reverse Engineering "Life"

Author:
Manuj Mishra

Supervisor:
Prof. Andrew Davison

Second Marker:
Dr. Edward Johns

June 22, 2022

Abstract

Since the 1970s, John Conway's Game of Life, often shortened to just "Life", has garnered great academic interest as an example of emergent complexity. Life is a cellular automaton (CA), a discrete model of computation that simulates localised interactions of state-storing "cells". CA can model real-world dynamical systems such as fluid flow[1], tumour growth[2], and urban land use[3]. Automatically learning CA models from observations of such systems could save time and yield more accurate predictions. Deep learning has been shown to effectively learn CA parameters from artifically generated observations (*full rule dynamics*)[4]. We investigate the use of evolutionary algorithms to this end.

We build a cellular automaton simulator and evolutionary algorithm toolkit to optimize life-like CA, a class of discrete state CA. As a precursory goal, we build a CA-based procedural maze generator which is optimized using the toolkit. We then learn the full rule dynamics of life-like CA. Ultimately, we extend the simulator and toolkit to Gray-Scott models[5], a class of continuous state CA.

We extend existing work around procedural generation of mazes using CA[6] through improved learning processes, fitness functions, and post-processing algorithms. We present, for the first time, evolutionary algorithms to learn the full rule dynamics of 2D cellular automata. For life-like CA, we show that genetic algorithms can learn global optima for 100% of targets in a test set of 100 random samples while traversing, on average, less than 0.05% of the search space. We develop a learning pipeline for continuous-state CA and find locally optimal regions to which genetic algorithms and self-adapting evolutionary strategies converge.

Acknowledgements

I would like to thank my supervisor, Prof. Andrew Davison, who has supported me in every direction I have taken this thesis. Your continued enthusiasm has given this project *life*.

I would like to thank my friends, who have made the last 3 years a joy and inspire me in ways they cannot imagine. Your presence gives *life* its colour.

Finally, I would like to thank my family, who have always been my anchor and my compass. Without you, *life* would be impossible.

Contents

List of Figures

List of Definitions

List of Algorithms

Chapter 1

Introduction

(a) t=0 (b) t=10 (c) t=20 (d) t=30

Figure 1.1: Gosper's Glider Gun, the first known pattern to exhibit unbounded growth in Conway's Game of Life.[7]

1.1 Motivation

Predicting effects is easier than predicting causes. This is the crux of the inverse problem in science. Estimating observations from a parameterised model of the world easier than recovering parameters from observations. For instance, a physical model of the universe may allow us to predict the sub-atomic particles ejected when two protons collide at high speed but building theories *based on* such collisions is difficult, especially when there are multiple equally valid explanations. Despite this, the pursuit of inverse problems is critical to advancing understanding around a system's behaviour.

In this thesis, we tackle an inverse problem for cellular automata (CA). We learn the parameters of CA based on observations of their state. CA are simple yet powerful models in which a discrete lattice of state-storing "cells" simultaneously perform localised computations at regular time steps. The state of each cell depends exclusively on the state of the cells in a neighbourhood around it in the previous time step. This localised interaction makes CA a useful representation of physical and biological systems at all scales including fluid flow[1], tumor growth[26, 2] and urban land use[3]. By automatically learning CA that accurately model such systems, we can improve our ability to predict their behaviour and develop a better understanding of their internal mechanisms. As well as simulatory models, CA are powerful computational engines owing to their inherently parallel structure. This makes learning CA a useful endeavour in the field of distributed computation as well[27].

1.2 Objectives

Top-down investigations, which seek to classify CA and prove general results about long-term behaviour from their parameters, are vast and varied[28, 10, 14]. In this thesis, we explore the less common bottom-up approach, where we deduce the underlying properties of a CA by observing its behaviour. Specifically, we learn the *transition function* of the CA which defines how it uses its current state to generate its state in the next time step.

We consider three learning goals. From easiest to most difficult, they are:

1. State Optimization: Learn a CA that behaves "similarly enough" to the target system according to some fitness function.

2. State Extrapolation: Learn a CA that, for a particular initial condition, behaves exactly like the target system.

3. Full rule dynamics: Learn a CA that, for all initial conditions, behaves exactly like the target system.

We focus on two classes of CA in particular. The first are life-like CA which are a discrete family of models in which Conway's infamous "Game of Life" CA belongs. These CA exhibit many of the properties we wish to model in real-life systems such as chaos, nonlinear dynamics and the emergence of complexity. The second are Gray-Scott models which are continuous state CA that simulate the reactions of two diffusive chemicals. They are a discrete-time simulations of Turing's original model of morphogenesis which underlies many theories in developmental biology. For example, learning the full rule dynamics of Gray-Scott models could allow us to understand the interactions of hormones that elicit the stripes on a zebra or the spots on a leopard.

We use evolutionary algorithms (EAs) as our learning framework. EAs have long been held as effective tools for black-box optimisation problems. Grounded in the principles of Darwinian evolution, EAs traverse over a search space by performing selection, mutation, and crossover on a population of candidate solutions. As selection pressure grows, increasingly strong solutions emerge. Using EAs, we tackle the following objectives. Listed from easiest to most difficult, they are:

1. State optimization for life-like CA: Learn life-like CA that induce stable states with desirable properties from random initial conditions.

2. Full rule dynamics for life-like CA: Learn the exact transition function of life-like CA based on a small sample of observations.

3. Full rule dynamics for Gray-Scott models: Learn the exact transition function of Gray-Scott models based on a small sample of observations.

1.3 Contributions

To achieve these objectives, this project makes the following key contributions:

- **Evolutionary algorithm toolkit**
  We build a versatile toolkit that implements evolutionary algorithms to train and optimise different classes of CA. It features support for different genotypes, loss functions, genetic operators and selection methods.

- **CA simulator**
  We build simulators for discrete and continuous CA. This allows various transition functions to be implemented in the EA toolkit. These also render snapshots of the CA directly during simulation which allows animations to be automatically generated afterwards.

- **Procedural maze generator**
  We write a CA-based maze generation program to test EA toolkit for state optimization of life-like CA. The maze generator is optimised by the EA toolkit to produce "challenging" mazes which exhibit properties such as a long solution path and many dead ends. Mazes generated are guaranteed to have a valid solution path from start to end.

- **Learning life-like CA**
  We implement a genetic algorithm that learns full rule dynamics of life-like CA. It is able to learn all 100 randomly generated test cases. On average, it searches 0.047% of the search space before converging on the global optimum. This is a significant improvement over a random exhaustive search which would search 50% of solutions on average.

- **Learning Gray-Scott CA**
  We implement a genetic algorithm and evolutionary strategy that converge to locally optimal regions for Gray-Scott models. These regions are visualised with respect to the target in the parameter space.

- **Systematic analysis of life-like CA**
  We provide an analysis of life-like CA based on repeated finite-time simulations over random initial conditions. This suggests characteristics of CA that make them easier or harder to learn automatically.

- **Media generation**
  A program to convert numerical data from CA simulations into high-fidelity media. In tandem with statistical summaries, this created opportunities for qualitative analysis. This also helps provide a visual intuition for the project when communicating results.

1.4 Technical Challenges

The primary technical challenges faced during this project were:

- **Building from the ground up**
  This project is built entirely from scratch with no external libraries except simple data and image processing packages (e.g. NumPy, Pandas, Python Image Library). This involved producing implementations of cellular automata, evolutionary procedures, genetic operators, and loss functions. These components went through many iterations to ensure they were extensible and modular.

- **Efficient simulation**
  Each evolutionary algorithm runs on large populations over multiple epochs. Fitness is calculated by repeated simulation of individual CA which means that thousands of simulations need to be run for each experiment. Therefore, the simulator must be efficient. This is especially important for the Gray-Scott simulator which uses numerical integration to approximate solutions for 2D partial differential equations. Producing a realistic yet efficient simulation in this case required much experimentation.

- **Avoiding local optima**
  Evolutionary algorithms are very susceptible to premature convergence to local optima. It is infeasible to test every combination of evolutionary techniques, genetic operators, loss functions, and hyperparameters. Therefore, the design of these algorithms had to be approached with forethought to maximize the chances of avoiding local optima.

- **Efficient experimentation**
  Hyperparameter tuning and evaluation of each technique required running large-scale experiments. With the computational resources available, it was important to consider which experiments would provide the most insight relative to the time taken to run them. As an example, simulation of all life-like CA rules for exploratory analysis required 25 hours on a 4-core, 8GB AMD-Compute cluster.

1.5 Ethical Considerations

Technologies falling in the broad field of self-organising systems, distributed systems, and automated systems can be misused. This presents some ethical issues that are worth discussing.

One area of legal concern is the infringement of copyright law when selecting training data. To train cellular automata for real world applications, we would use many datasets collected from physical, biological, or chemical experiments. As a collection of facts, such data is usually exempt from copyright law. However, training data can, in theory, be derived work which puts it in the remit of copyright law. Examples include certain terrain maps or urban land use reports. When choosing point data to train on, we will ensure that they do not fall under copyright restrictions and that they are legally suitable for academic use.

An area of societal concern is the explainability of CA systems. Suppose a CA model is built to forecast the spread of an epidemic and results are used to guide public policy. While this kind of problem is particularly pertinent to black-box systems like artifial neural networks, it is less of a concern for CA which tend to be simplistic and more transparent. This work does not delve into CA models with any public or business use.

An area of ethical concern is the general application of cellular automata in distributed technology like swarm robotics. Such systems can have military applications. It could be argued that this research could pave the way for highly distributed swarms of drones or ground robots that are capable of complex self-organising behaviour. However, this is unlikely to be a true concern since we only discuss theoretical concepts in this paper with little discussion about physical applications in robotics. Furthermore, this research is only in its very preliminary stages. As a relatively mathematical project, there are few major ethical factors to consider here.

Chapter 2

Preliminaries

2.1 Cellular Automata

A CA is a computational model that performs multiple parallel computations, each depending on local interactions, to produce complex global behaviour. We define a CA formally as follows.

Definition 2.1 (Cellular Automaton)**.** *A cellular automaton is an n-dimensional finite grid of computational units called cells. Each cell c_i is characterised by:*

- *A discrete state variable $\sigma_i(t) \in \Sigma$, where i indicates the index of the cell in the lattice, t indicates the current time step, and Σ denotes the finite set of all state variables.*
- *A local neighbourhood set $\mathcal{N}(c_i)$ with cardinality N.*
- *A transition function $\phi : \Sigma^N \to \Sigma$ which takes local neighbour states as input. This is also known as the CA "update rule".*

At each time step, the state of each cell is simultaneously updated according to the transition function. That is, $\sigma_i(t+1) = \phi(\ \{\sigma_j(t) \mid c_j \in \mathcal{N}(c_i)\}\ )$

Due to the breadth of systems studied in CA literature, the constraints of this definition are often altered to produce interesting arrangements. For example:

- The structure need not be a square grid. CA have been studied on hexagonal grids[29], aperiodic tessellations such as as the Penrose tiling[30], and even randomly generated structures like the Voronoi partition[31].
- The system need not be deterministic. Probabilistic cellular automata (PCA) have stochastic transition functions which describe a probability distribution of possible outcomes for any given input. PCA are able to model random dynamical systems in the real world from stock markets[32] to infectious diseases[33].
- The state space Σ need not be finite. In this thesis we will explore multiple possible state variable representations including bit arrays and continuous vectors.

For the purpose of this thesis, we will assume the original definition of CA unless otherwise stated.

We consider a "neighbourhood function" for each cell $c_i \mapsto \mathcal{N}(c_i)$. This makes it easier to discuss neighbourhood sets of cells in the CA, each of which are typically homogenous. There are many possible neighbourhood functions for any given CA geometry. When defining the neighbourhood function, we select a distance metric $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ to measure the proximity of two cells and we set a threshold T under which we consider two cells to be within each other's neighbourhood.

$$c_i \in \mathcal{N}(c_j) \iff d(c_i, c_j) \leq T \tag{2.1}$$

There are two neighbourhoods that are frequently used on Euclidean lattices. These are depicted in Figure 2.1. The *von Neumann neighbourhood* contains all cells within a Manhattan distance of 1. For a 2D square lattice, this contains the cell itself and the 4 cells in the cardinal

directions. For a 3D cubic lattice, it contains the central cell and a 6-cell octahedron around it. The *Moore neighbourhood* contains all cells at a Chebyshev distance of 1. For a 2D square lattice, this is the central cell with the 8 neighbouring cells in a square around it. In the 3D case, it is a cube.

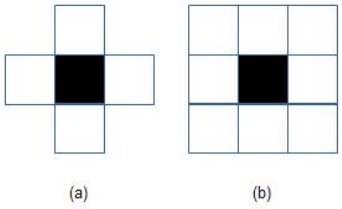


Figure 2.1: (a) von Neumann Neighbourhood and (b) Moore neighbourhood on a 2D square lattice [8]

In a finite grid CA, border cells must be given special consideration since they do not have the same number of neighbours as interior cells and therefore cannot share the same neighbourhood function. One option is to define a case-wise neighbourhood function with different behaviour for border cells. Another option is to freeze the state of border cells. In the field of partial differential equations, this is known as setting "fixed boundary conditions". The problem can also be circumvented entirely by relaxing the finite grid assumption and allowing cells to "wrap around" the grid. This is known as setting "periodic boundary conditions" and can be imagined visually as running the CA on an infinite periodic tiling or, alternatively, on a torus.

2.1.1 Life-Like CA

The most popular example of a CA is the Game of Life (henceforth "Life") formulated by John Conway in 1970 [34]. It consists of a 2D grid of cells, each with a boolean state variable signifying that the cell is either "alive" or "dead". The transition rule takes as input the cell's own state $\sigma_i(t)$ and the number of living individuals in the cell's Moore neighbourhood (excluding itself), denoted $n_i(t)$. This is as follows:

$$\phi(\sigma_i(t), n_i(t)) = \begin{cases} 0 & \sigma_i(t) = 1 \text{ and } n_i(t) < 2 \text{ (Death by "exposure")} \\ 0 & \sigma_i(t) = 1 \text{ and } n_i(t) > 3 \text{ (Death by "overcrowding")} \\ 1 & \sigma_i(t) = 1 \text{ and } n_i(t) \in \{2, 3\} \text{ (Survival)} \\ 1 & \sigma_i(t) = 0 \text{ and } n_i(t) = 3 \text{ (Resurrection)} \\ 0 & \sigma_i(t) = 0 \text{ and } n_i(t) \neq 3 \end{cases} \tag{2.2}$$

Despite its simple setup and update rule, Life can exhibit the emergence of complex patterns. It is possible to simulate a fully universal Turing machine within Life[35] and, as a corollary of the Halting Problem, this means that Life is undecidable. Given two arbitrary configurations, it is impossible to algorithmically determine whether one will follow the other.

Patterns found within Life include fixed-point solutions to the transition function like the *block* as well as periodic oscillators like the *beacon* which has period 2. There are also periodic patterns that move across the lattice such as the *glider* pattern. The density of a state is the number of living cells as a ratio of total grid size. It is possible to discover new stable patterns by repeatedly running specific rules on random initial states of a pre-determined density (called soups) and classifying the objects remaining after transient patterns have dissipated. Large-scale experiments of this nature are called "soup searches"[36].

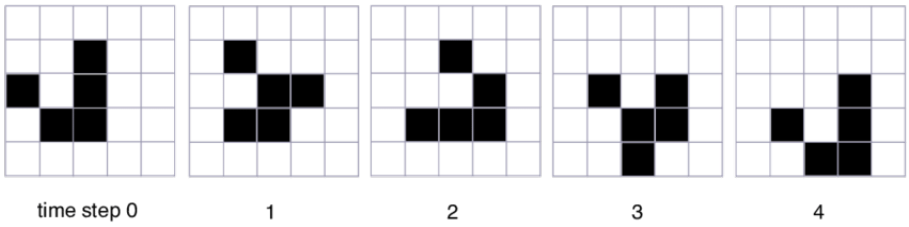


Figure 2.2: The glider pattern in the Game of Life [9]

A CA is considered "Life-like" if it exists on a 2D lattice, has binary state, and uses the Moore neighbourhood function. Life-like CA exist in two varieties: inner-totalistic and outer-totalistic.

Definition 2.2 (Inner-Totalistic)**.** *A Life-like CA is inner-totalistic if the output of the transition function depends only on the number of living cells in a cell's neighbourhood (including the cell itself).*

$$\sigma_i(t+1) = \sigma_j(t+1) \iff \sum_{c_p \in \mathcal{N}(c_i)} \sigma_p(t) = \sum_{c_q \in \mathcal{N}(c_j)} \sigma_q(t) \tag{2.3}$$

Definition 2.3 (Outer-Totalistic)**.** *A Life-like CA is outer-totalistic if the output of the transition function depends on both the number of living cells in a cell's neighbourhood and the state of the cell itself.*

$$\sigma_i(t+1) = \sigma_j(t+1) \iff \sum_{c_p \in \mathcal{N}(c_i)} \sigma_p(t) = \sum_{c_q \in \mathcal{N}(c_j)} \sigma_q(t) \quad \text{and} \quad \sigma_i(t) = \sigma_j(t) \tag{2.4}$$

As an example of the subtle difference here, consider the configurations shown in Figure 2.3. An inner-totalistic CA would yield identical configurations in the next time step since both input configurations have 3 active cells in the neighbourhood set. However, an outer-totalistic CA would treat both configurations differently as one has an live centre cell and the other has a dead centre cell. This discrepancy corresponds to a great difference in the size of search spaces. There are $2^{10} = 1024$ inner-totalistic CA but $2^{18} = 262144$ outer-totalistic CA. We represent the transition function of an outer-totalistic CA in a form called birth-survival (B/S) notation. Using this notation, we can represent the Game of Life as B3/S23. In this thesis, when we refer to Life-like CA, we implicitly assume the outer-totalistic variety.

Figure 2.3: Two possible configurations of a Life-like CA

Definition 2.4 (Birth-Survival notation)**.** *Let N_b and N_s be sets of integers. We say an outer-totalistic CA has rulestring* BN_b/SN_s *if it has transition function:*

$$\phi(\sigma_i(t), n_i(t)) = \begin{cases} 1 & \sigma_i(t) = 0 \text{ and } n_i(t) \in N_b \text{ (Birth)} \\ 1 & \sigma_i(t) = 1 \text{ and } n_i(t) \in N_s \text{ (Survival)} \\ 0 & \text{otherwise} \end{cases} \tag{2.5}$$

2.1.2 Elementary CA

Elementary CA are defined on the simplest nontrivial lattice, a finite one-dimensional chain. The neighbourhood of each cell contains the cell itself and the two cells adjacent to it on either side.

The state variable is a boolean which means there are $2^3 = 8$ possible neighbourhood state configurations. A transition rule maps each of these neighbourhood states to a resultant state and can therefore be represented as an 8-digit binary rule table $(t_7t_6t_5t_4t_3t_2t_1t_0)$ where configuration (000) maps to t_0, (001) maps to t_1, ..., and (111) maps to (t_7). Consequently, there are $2^8 = 256$ possible transition functions for elementary CA.

The Wolfram code, a number between 0 and 255 obtained by converting the binary rule table to decimal, is the standard naming convention for these rules. Rule 110 is particularly notable as it can exhibit and is Turing complete [37]. Figure 2.4 shows an example progression of a Rule 110 system. Each row of pixels represents the state of the automaton at one snapshot in time with the topmost row representing the randomized initial state. It shows the emergence, interaction, and subsequent dissipation of multiple long-lived impermanent patterns. This is called class 4 behaviour[10] (see Def 3.1).

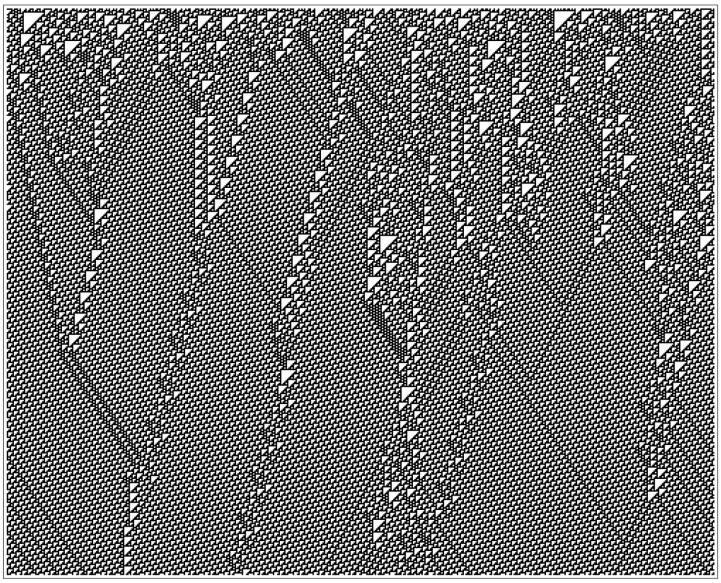


Figure 2.4: Rule 110 progression with random initialisation [10]

2.2 Turing Patterns

Morphogenesis is the process by which a system develops into a particular shape or pattern. Biologically, this is seen in most multicellular organisms which can robustly develop specialised organs and intricate skin patterns without any centralised decision-making. Through simple rules encoded in the genome and homeostatic feedback loops enforced through chemical signalling, a tissue knows exactly how to grow and when to stop.

Figure 2.5: Complex patterns on sea shells (left) can be replicated using Turing patterns (right) [11]

In *The Chemical Basis for Morphogenesis (Turing, 1952)*[38], Alan Turing proposes that spatially periodic phenomena in the natural world like the stripes on a zebra or the skin patterns on pufferfish can arise autonomously from random or uniform initial conditions through the interaction for two diffusible substances. These patterns are known as Turing patterns and Turing's model forms the basis of major theories in developmental biology. Such patterns are visible at all scales from the fold patterns of mammalian brains[39] to the distribution of matter in the Milky Way[40]. Figure 2.5 is an example of Turing patterns on sea shells.

2.2.1 Gray-Scott Model

Turing patterns arise out of two component reaction-diffusion systems. One specific example is the Gray-Scott model[5] in which one component, U, is consumed while the other, V is produced in a chemical reaction. In this thesis, we consider a simple reaction scheme called cubic autocatalysis where the following reaction occurs with a certain probability

$$U + 2V \rightarrow V \tag{2.6}$$

To compensate for U being consumed, the system replenishes U and removes V by rates controlled with feed and kill factors f and k respectively. The substances diffuse over the grid at rates r_u and r_v. The system is characterised by two equations.

Definition 2.5 (Reaction-Diffusion System)**.**

$$\frac{\partial u}{\partial t} = -uv^2 + f(1-u) + r_u \nabla^2 u \tag{2.7}$$

$$\frac{\partial v}{\partial t} = uv^2 - (f+k)v + r_v \nabla^2 v \tag{2.8}$$

where u and v represent the densities of each component in a given cell. $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$ represent the change in these densities with respect to time. Each density has three sources of change described by the three terms on the right hand side of each equation. The first term describes the reaction 2.6 where one U molecule reacts with two V molecules which is why the term is a product of u and v^2. The second term represents external inputs and outputs. In 2.7, the feed rate is multiplied by $1-u$ so that replenishment depends on current density. In 2.8, v is multiplied by $-(f+k)$ so that V is removed faster than U is added. Finally, the third term describes the change in each density due to diffusion using the 2D Laplacian to get the difference between a cell's current state and the average of the neighbourhood cell states.

For most values of feed and kill rate, the Gray-Scott model attains one of two quiescent states, either completely dominated by U or completely dominated V. However, there are certain feed and kill rates which elicit complex stable patterns. Many of these bear a strong resemblance to Turing patterns observed in nature as shown by Figure 2.6.

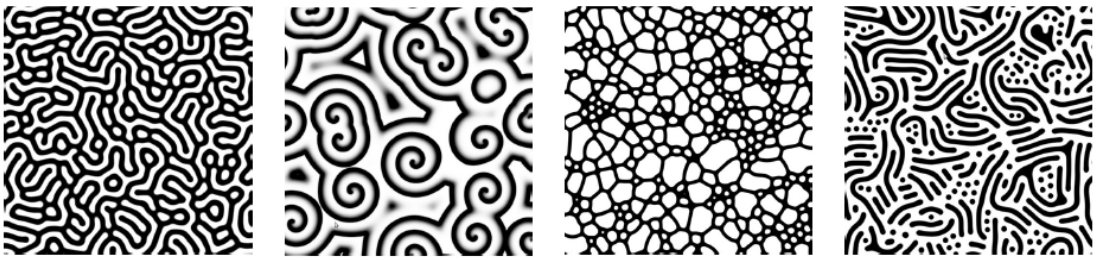


Figure 2.6: Turing patterns arising out of Grey-Scott model simulations[12]

2.3 Evolutionary Algorithms

Evolutionary algorithms (EAs) are a family of heuristic-based search algorithms for black-box optimisation problems. They are inspired by biological evolution. A chromosome is an indirect

encoding of a candidate solution. The structure of the chromosome is called the "genotype" and the behaviour of the corresponding candidate solution is called the "phenotype". The fitness of a candidate is calculated from its phenotype. An evolutionary algorithm initializes population of chromosomes and modified them through repeated recombination, mutation, and selection. The purpose of recombination is to produce new solutions with "fit" attributes by combining aspects of existing candidates. Mutation, on the other hand, is applied to explore new areas of the search space by randomly perturbing existing candidates. Mutations may lead to weaker candidates in the short term but is useful as a method of discovering new attributes that, over many generations, can produce fitter candidates. Finally, selection applies competitive pressure to the population, ensuring that the overall fitness of the population is improving and increasingly fit solutions are being discovered.

EAs are valued for their broad applicability as they require no information about the constraints or derivative of the objective function. In fact, an explicit representation of the objective function is not even neccessary to run an EA as long as candidates can be compared to each other. Selection pressure can then be introduced in the form of tournament-based elimination.

2.3.1 Genetic Algorithms

A genetic algorithm (GA) is a particular type of evolutionary algorithm with binary string genotype. For a genetic algorithm, we perform recombination by randomly picking two parents with replacement from the elite subset and mixing their chromosomes to produce a child. This is called crossover. We formalise the structure of a genetic algorithm in Algorithm 1.

Algorithm 1 Schematic Genetic Algorithm

Require: S - the set of possible chromosome values
Ensure: $s^* \in S$
  $t \leftarrow 0$
  $M_0 \leftarrow \mu$ random individuals from S
  while stopping condition is false **do**
    EVALUATE(M_t)
    $P_t \leftarrow$ SELECTPARENTS(M_t) ▷ Parents
    $\Lambda_t \leftarrow$ RECOMBINE(P_t) ▷ Children
    $Pmod_t \leftarrow$ MUTATE(P_t)
    $\Lambda mod_t \leftarrow$ MUTATE(Λ_t)
    $M_{t+1} \leftarrow$ SELECTPOPULATION($Pmod_t$, Λmod_t)
    $t \leftarrow t + 1$
  end while
  $s^* \leftarrow$ FINDBESTCANDIDATE(P_t)

The initial selection phase (SELECTPARENTS()) uses a fitness function to compare and select the top candidates. The latter selection phase (SELECTPOPULATION()) produces a new population from the modified parents and children. Population-wide selection criteria can be enforced in this phase. For example, weak parents can be eliminated if they have survived for too many generations or, symmetrically, children can be granted immunity for a particular number of generations.

Genetic operators are specific implementations of actions such as crossover and mutation. The most common mutation operator is pointwise mutation where each bit in the chromosome is flipped with some fixed probability p. The three common crossover operators used in genetic algorithms are:

1. Uniform Crossover: For each bit (or "gene") in the chromosome, copy the corresponding gene from one parent.
2. Single Point Crossover: Randomly pick a split point. Take the all bits to the left of the split point from one parent and all bits to the right from the other.
3. Multiple Point Crossover: Randomly pick n split points. Alternate picking contiguous sections of the chromosome between split points from each parent.

Chapter 3

Related Works

This chapter summarises recent work on the analysis and learning of CA. We focus on the two classes of CA presented in Chapter 2, life-like CA and Gray-Scott models.

3.1 Life-Like CA: Exploration

The choices of lattice geometry, neighbourhood function, state variable, and transition rule define the behaviour of a CA. Fixing the former three factors, Wolfram[41] classified CA based on transition rules as follows:

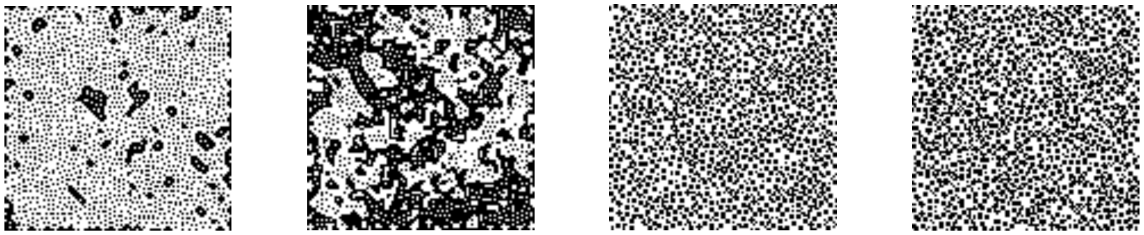
Definition 3.1 (Wolfram Classes)**.** *The four Wolfram classes are:*

- *Class 1 (Null) : Rules that lead to a trivial, homogenous state*
- *Class 2 (Fixed-point / Periodic) : Rules that lead to localized stable or periodic patterns*
- *Class 3 (Chaotic) : Rules that lead to continued, unending chaos*
- *Class 4 (Complex) : Rules that can lead to complex, long-lived impermanent patterns*

Early attempts to categorise 2D CA by Packard and Wolfram[28] extend Wolfram's original four categories. They classify rules based on information content and rate of information transmission measured using Shannon entropy and Lyapunov exponents respectively. These works provide the basis for exploration and classification of life-like CA. However, Wolfram's classes do not have clear decision boundaries and these metrics do not help quantitatively define them either. This is because Wolfram's classes are based on the characteristics that a CA is capable of possessing under *some* not *all* initial conditions. Equivalently, they are based on what a CA *cannot* do. For example, a class 2 CA is one which cannot exhibit chaotic patterns. This makes it impossible to classify CA before observing the result of their simulation. As proven by Yaku[42], many questions about global properties of 2D CA are formally undecidable which makes the construction of definitions based on long-term outcomes difficult and limits the usefulness of Wolfram's taxonomy.

Adamatzky et al.[13] produces a systematic analysis of life-like CA in which the birth and survival sets are contiguous intervals. These are dubbed "binary-state reaction-diffusion cellular automata (RDCA)" as they provide a discretized model of simple two-chemical reactions with substrate '0' and reagent '1'. The birth set is analogous to diffusion rate and the survival set is analogous to reaction rate. The analysis includes categorisations based on qualitative factors like the features and density of resulting configurations. It also includes quantitative factors such as the outcome of glider collisions within each universe. For example, the **P**-class contains rules with high diffusion rate (i.e. wide birth interval) and low reaction rates (i.e. narrow survival interval) which produce large regions of 0-state and 1-state each containing scatterings of the other within them. These patterns are qualitatively distinct from, for example, **O**-class rules which have low diffusion rate and high reaction rate producing irregular spotted patterns. These are shown in Figure 3.1. Despite the depth of this investigation, the 1296 CA rules analysed cover less that 0.05% of all life-like CA. The broader issue in both Wolfram's and Adamatzky's classifications is the lack of objective distinction between class boundaries which makes it difficult to predict the

behaviour of rules *a priori*. Indeed, some CA have been proven to span multiple classes[43].



(a) B12345678/S23 (b) B12345678/S234 (c) B1/S345 (d) B1/S345678

Figure 3.1: Configurations generated from **P**-class (a,b) and **O**-class (c,d) rules [13]

This dilemma is alleviated to some degree by Eppstein's four-way classification[14] which is based on strict definitions of *fertility* and *mortality*. A rule is fertile if there exists a finite pattern that eventually escapes any bounding box B. Note this is symmetrically opposite to the definition of periodicity since any infertile rule can only iterate through $2^{|B|}$ steps before repeating a previous state. A rule is mortal if it supports a pattern which transitions to the *quiescent* state (i.e no live cells) in the next time step. Eppstein conjectures that "interesting" behaviour arises out of rules that are both fertile and mortal. Figure 3.2 depicts a schematic map of his analysis.

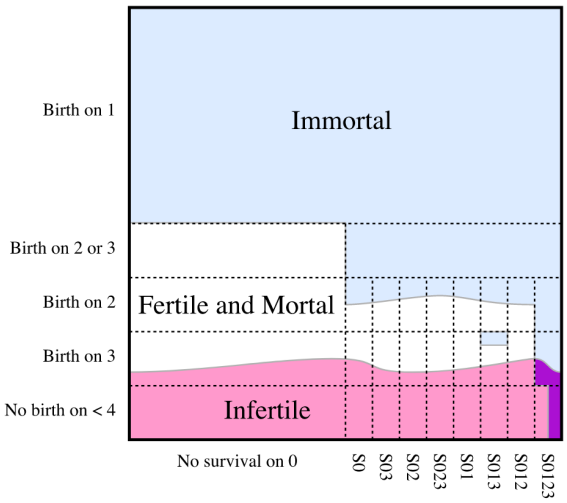


Figure 3.2: Map of fertile, infertile, mortal, and immortal regions in binary-state RDCA rulespace [14]

This work provides a strong theoretical foundation to guide our search of life-like CA and to verify that our techniques are effective on different varieties. However, they are not grounded in a systematic statistical search which makes it difficult to ascertain the proportion of each category that exist in contested regions. For example, we may be interested in the ratio of fertile to infertile configurations for rule B3/S01. Although a closed-form solution for this ratio is infeasible, it is possible to come to an approximation through simulation. As mentioned in the preliminaries, large scale simulations of random initial conditions on particular rules have proven to be an effective way of identifying new patterns[36]. This is called soup searching. In a similar vein, we will use soup searches to approximate the periodicity of all rules in the life-like CA rulespace.

3.2 Life-Like CA: Learning

A seminal work by Meyer et al.[15] looks at learning 2D CA neighbourhood functions using genetic algorithms. Later works by Mitchell et al.[16] explore the effectiveness of genetic algorithms in learning entire transition functions but only in the domain of elementary cellular automata. Around

the same time, Koza et al. make leaps by applying genetic programming to a broad variety of tasks including the CA majority classification problem[17]. Incremental improvements have been made since then with Breukelaar and Bäck[44] delivering experimental evidence that the CA inverse design problem using evolutionary computation is more tractable in higher dimensions. We delve briefly into some of these papers to compare their aims, methods, and outcomes.

3.2.1 Learning Neighbourhood Functions

In *Learning Algorithm for Modelling Complex Spatial Dynamics (Meyer et al., 1989)*[15], the neighbourhood function of a binary probabilistic cellular automaton (PCA) was evolved to model artificially generated datasets. The motivation was to establish a CA architecture capable of codifying patterns in physical interactions directly from experimental data. It was successful to this end as Richards et al.[45] used results from this work to predict the dendritic solidification structure of NH_4BR.

Meyer's genetic algorithm seeks solutions within a 20-cell vicinity where each cell can be included or excluded from the neighbourhood set. It is the intersection of the Moore neighbourhood in time step $t-1$ and the von Neumann neighbourhood of range two in time step $t-2$ as visualised in Figure 3.3. The full 20-cell neighbourhood is called the master template and each chromosome encodes some subtemplate $s_1, ..., s_m$.

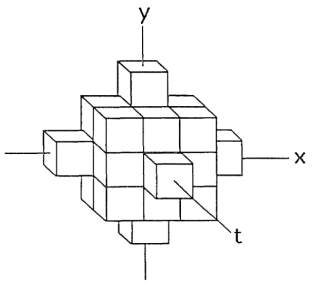


Figure 3.3: 20-cell, two step neighbourhood in space and time[15]

The fitness function used is

$$F = I - \frac{2^m}{N} \tag{3.1}$$

$$\text{where } I = \sum P(s, s_1, ..., s_m) \log_2 \frac{P(s, s_1, ..., s_m)}{P(s)P(s_1, ..., s_m)} \tag{3.2}$$

Here, I is the mutual information of the subtemplate and represents the amount of information, measured in Shannon bits, that can be obtained about the value of the central cell from subtemplate states. It is calculated by summing across all 2^m configurations of the subtemplate in the data and across both values of $s \in \{0, 1\}$. The second term in the fitness function ensures that subtemplates of varying sizes are treated appropriately by proportionately penalising large subtemplates that, by nature, will contain more information. $N = 20$ is the size of the master template.

The genetic algorithm initialises the population at a randomly chosen subset of possible subtemplates. Selection is performed using a truncated linear ranking. Crossover is applied using an arbitrary cut in space-time on the master template as the crossover point. Point mutation is applied by either adding or removing a single cell from each candidate. This process is iterated to converge towards an optimum.

As the first notable exploration of learning CA properties with genetic algorithms, this paper demonstrates the ability of genetic algorithms to efficiently traverse an opaque search space. The algorithm precisely learns neighbourhoods interior to the master template such as the one time step Moore neighbourhood and even when the objective neighbourhood lies partially outside the master template, the algorithm successfully finds a close approximation. For example, when given data

produced by a one time step von Neumann neighbourhood, the algorithm learns a neighbourhood set that produces correct behaviour 96% of the time on tested targets.

This work also raises many questions for future research. The most pertinent is whether it is possible to link learned rules to existing and future theoretical models. Moreover, this work only explores binary state CA but application of similar techniques on continous-state CA could closer approximate the partial differential equations that underlie the physical processes being modelled.

Finally, this paper focuses on optimising the neighbourhood set of the CA model only. It aims to establish *which* parameters in a local vicinity of a current cell are most relevant to predicting the future state, not *how* those parameters are combined and transformed to produce the result. In this thesis, we are interested in going beyond this and approximating the full transition function. In some cases we will fix the neighbourhood function used to reduce our search space under the assumption that techniques from this paper can be used to find optimal sub-neighbourhoods if they exist.

3.2.2 Learning 1D Transition Functions

There are a number of problems in elementary CA computation that have piqued academic interest from both analytical and computation angles. One example is the firing gun synchronisation problem[46] which seeks a rule that minimizes the time to get a CA from a quiescent state (all 0) to a firing state (all 1). Another is the density classification problem, or majority problem, which aims to find a binary elementary CA rule that accurately performs majority voting. That is, all cells converging to the state that dominates the initial condition. Despite their simple formulation, both of these problems require the transfer of information through compressed, latent representations and a global consensus based on localised computations. This makes them useful benchmarks when measuring the capability of CA and the algorithms used to design them. For the sake of brevity, we focus only on the majority problem in this section. We formalise it in Def 3.2.

Definition 3.2 (Majority Problem)**.** *An elementary CA of size N solves the majority problem for some initial conditions* $\{\sigma_i\}_{i=1}^{N}$ *if* $\exists T$ *s.t.* $\forall t > T$*:*

$$\sigma_i(t) = \begin{cases} 0, & \sum_{i=1}^{N} \sigma_i(0) < \frac{N}{2} \\ 1, & \sum_{i=1}^{N} \sigma_i(0) > \frac{N}{2} \end{cases} \tag{3.3}$$

The desired result is undefined if the initial state contains an equal number of 0 and 1 cells.

The Gacs-Kurdyumov-Levin (GKL) rule is a human-designed solution to solve this problem. The function, as defined in Def 3.3, allows consensus to be reached in $O(N)$ time and, for $N = 149$, achieves success on 81.6% of inputs[47]. Modifications throughout the 1990s incrementally improved this classifier[48]. Although these were very promising, the human designed aspect of these algorithms meant there was little to support their optimality compared to others in the rulespace.

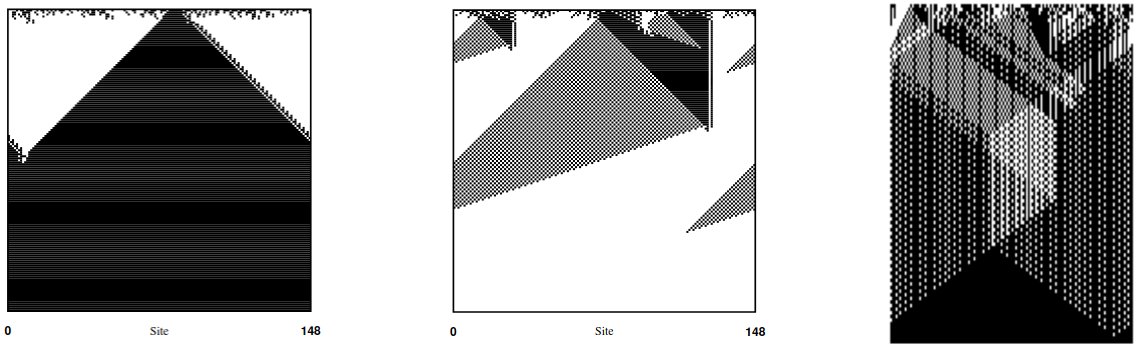
Definition 3.3 (GKL Classifier)**.** *A GKL density classifier is an elementary CA on periodic boundary conditions with transition function*

$$\sigma_i(t+1) = \begin{cases} Mo(\sigma_{i-3}(t) + \sigma_{i-1}(t) + \sigma_i(t)), & \sigma_i(t) = 0 \\ Mo(\sigma_i(t) + \sigma_{i+1}(t) + \sigma_{i+3}(t)), & \sigma_i(t) = 1 \end{cases} \tag{3.4}$$

where $Mo(.)$ *returns the mode of its arguments.*

A seminal series of work by Mitchell, Crutchfield, and Das[16] tackled this issue by automating the process of CA transition function design through evolutionary computation. These works made effective use of genetic algorithms operating on fixed length bitstrings. Rules with radius $r = 3$ were considered, leading to chromosomes of length $2^{2r+1} = 128$. The size of the rulespace was therefore 2^{128} which eliminates the possibility of any exhaustive search. The size of the CA itself was $N = 149$, chosen to be odd so that the solution to the majority problem is well defined. Upon initialisation, 100 chromosomes are chosen from a distribution that is uniform over chromosome

density. This can be viewed as picking the binary representations of 100 samples from a binomial distribution. This is markedly distinct from the usual unbiased distribution which assigns each bit in the chromosome to 0 or 1 with probability 0.5, equivalent to picking 100 samples from the $Uniform(0, 128)$ distribution. The choice of binomial initialisation has been shown to considerably improve performance[49]. At each generation, 100 new initial conditions (ICs) were created and fitness was defined as the percentage of correctly classified ICs. This stochastic fitness function was effective at reducing overfitting. A $(\mu + \lambda)$ selection method was employed with $\mu = 20$ and $\lambda = 80$ and mutation was performed with a two-point crossover. Although not as accurate as GKL, the best discovered solution still achieves a 76.9% accuracy. However, the evolved solutions were not very sophisticated. Most fell into the category of "block-expanding algorithms" or simple "particle-based algorithms" shown in Figure 3.4 as (a) and (b) respectively. In the former case, white and black boundaries meet to form a vertical line whereas, in the latter, they form a checkerboard region to propagate signals about ambiguous regions across the CA.



(a) Block-Based (b) Particle-Based (c) Koza Algorithm

Figure 3.4: Space-time behaviour of discovered transition functions on randomly initialised elementary CA of size N=149. (a) and (b) by Mitchell et al.[16] and (c) by Andre et al.[17]

A later work by Andre, Bennett, and Koza[17] uses genetic programming to achieve superior results qualitatively and quantitatively. The obtained solution uses various internal representations of density to transfer and collate information across the CA as showin in Figure 3.4 (c). It attains an accuracy of ∼82.3%. However, with its numerous latent representations of density, it is more difficult to intuitively understand how the Koza algorithm works and what patterns it is capable of producing than it is to understand how the particle-based algorithm works. Recently, it was proven that a perfect density classification rule for an infinite CA in any dimension, stochastic or deterministic, is impossible[50]. However, evolutionary computation still surprises in its ability to find approximations with ever-increasing performance.

3.2.3 Learning 2D Transition Functions

Few attempts have been made to extend evolutionary algorithms to multidimensional CA. A key series of work by Breukelaar and Bäch[18] shows that genetic algorithms can effectively solve information transfer problems on 2D CA such as the majority problem, AND problem, and XOR problem. The AND and XOR problems, which we collectively call the logical problems, aim to set the state of every cell to the result of the respective logical operation on the initial state of the top left and bottom right cells. These logical operators are picked because the result cannot be calculated from a single operand alone. In other cases, like the OR operator, we can deduce that the result will be true from either of the operands being true. For the majority problem, tests showed that results with a fitness of up to 70% have been acheived evolved using a von Neumann neighbourhood. For the logical problems, a von Neumann neighbourhood can elicit results with over 90% accuracy and a Moore neighbourhood with perfect accuracy. Notably, it was shown that crossover did little to aid the evolution process in the logical problems and frequently even hindered progress.

They also show promise at small scale morphogenesis. Morphogenesis for CA is the problem of evolving a rule that, given an initial condition, produces a particular goal state within a certain

number of steps. In this work, the goal bitmaps were 5x5 square patterns shown in Figure 3.5.

Figure 3.5: Goal bitmaps for morphogenesis experiment[18]

Using only mutation, not crossover, the algorithm was able to find a rule to produce every goal bitmap from a seed state of a single live cell in the centre of the CA within 5000 generations as shown in Figure 3.6. This is very promising and indicates that the value of genetic algorithms in producing CA rules not only encode information transfer mechanisms, but latent representations of data itself.

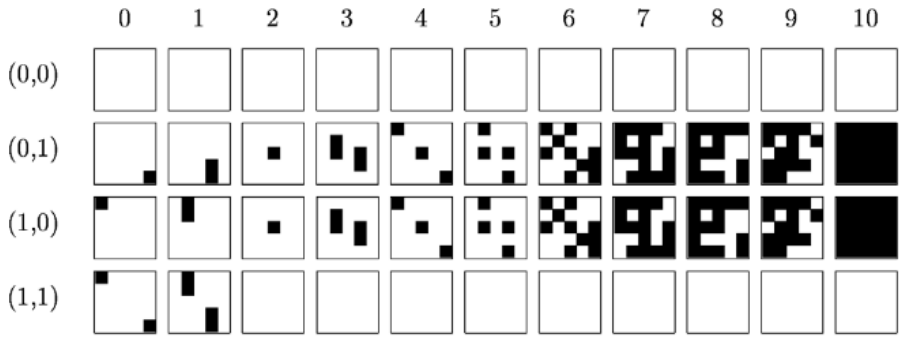


Figure 3.6: Iterations of XOR CA for 4 possible input states[18]

Early work by Wulff and Hertz in 1992[4] uses a lattice-shaped neural network style structure to learn CA dynamics. Each node in this lattice is a Probabilistic Logic Node, also known as a $\Sigma - \Pi$ unit [51]. These units are capable of representing any boolean function. That is to say $\forall f : \mathbb{R}^N \to \{-1, 1\}, \exists\ w_1, w_2, ..., w_N \in \mathbb{R}$ such that,

$$f(S_1, S_2, ..., S_N) = sgn\left[\sum_{j=1}^{N} w_j \prod_{i \in I_j} S_i(t)\right] \tag{3.5}$$

where index set I_j is randomly drawn from the integers $\{1, 2, ..., N\}$ without replacement. Notably, these units do not require input from every neighbour to learn successfully. In fact, this paper found any index set of size $|I_j| \geq \frac{N}{2}$ to be sufficient. This insight significantly reduced training time. Three learning goals were established for the network. In order of increasing difficulty they were:

1. Extrapolation: Learn to simulate a CA for a *particular* initial condition at any time
2. Dynamics : Learn to simulate a CA for *any* initial condition after short-lived patterns have been exhausted
3. Full Rule : Learn to simulate a CA for any initial condition at any time

This work was largely concerned with class 3 (chaotic) and class 4 (complex) behaviour. All 9 known examples of class 3 1D CA were tested on. However, at the time, it was believed that 1D CA could not exhibit class 4 behaviour. Therefore, testing was also conducted on Conway's Game of Life. With a shared network across all cells, this approach was very promising, with extrapolation and dynamics being very easy in the 1D and 2D cases. Learning the full rule was much harder. The network only was able to do so for 4 out of the 9 candidates in the 1D case. This work also divided class 3 elementary CA into two categories according to how easy it is to learn their underlying rule. However, this work was limited to only exploring class 3 and class 4 CA.

More recent work has used modern feed-forward neural networks to learn transition functions for morphogenesis. Specifically, in *Growing Neural Cellular Automata, (Mordvintsev et al., 2020)*[19], a CA is designed with a transition function that is itself a forward pass of a trainable convolutional neural network. In this form, the CA is trained to converge to complex yet stable image patterns from a single seed as shown in Figure 3.7. Later works build upon these techniques to learn full dynamical systems such as the Boids algorithm[52]. However, even for small neural networks, the potential rulespace is enormous with thousands of possible parameters. Ultimately, this makes a deep learning derived solution powerful yet intractable. Although CA with neural network transition functions are much more expressive and powerful due to the granularity with which behaviour can be tuned, they arguably defeat the purpose of encoding rich behaviour in a compressed chromosome such as a bitstring.

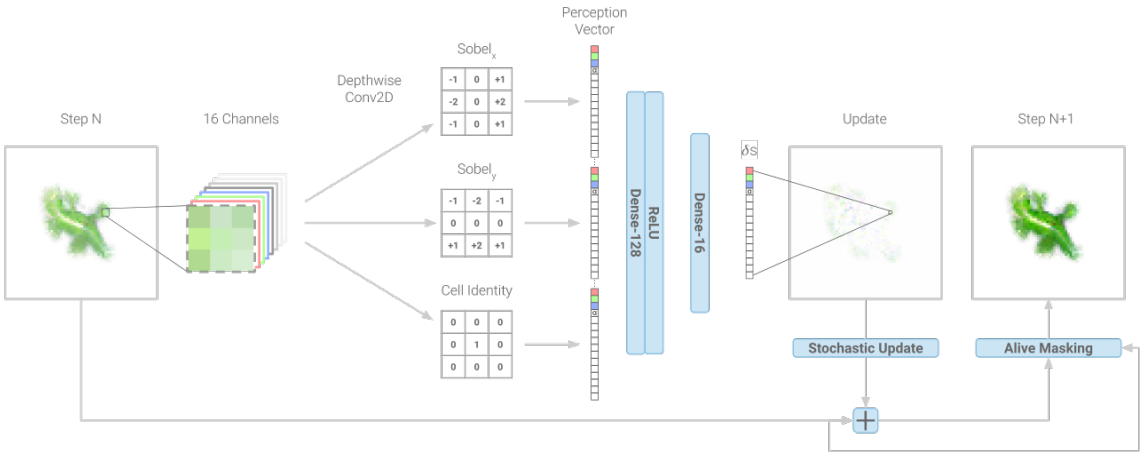


Figure 3.7: Single training step for a neural cellular automaton[19]

At the time of writing, no attempts have been made to use evolutionary computation to learn full rule dynamics of life-like cellular automata. This thesis is a first attempt.

3.3 Gray-Scott Systems: Exploration

Pearson[20] classifies patterns arising from Gray-Scott models into 12 categories based on temporal factors such as stability and decay rate as well as spatial factors such as regularity and emergence of subpatterns like spots and stripes. These are shown in Figure 3.8. Each simulation operates on a 256×256 grid with periodic boundary conditions. The initial conditions are uniform ($u = 1$, $v = 0$) with the exception of a 20×20 square patch in the centre with ($u = \frac{1}{2}$, $v = \frac{1}{4}$) perturbed with $\pm 1\%$ Gaussian noise.

Pearson discovered a threshold near which interesting behaviour can be observed. As the parameters f and k move across the threshold, the system transitions from one uniform stable state to another. These uniform states are dubbed R for red and B for blue. The red state corresponds to the trivial fixed point ($u = 1$, $v = 0$) and the blue state depends on the exact parameters, but tends to exist around ($u = 0.3$, $v = 0.25$). During the transition between these states, the system has two equilibria and at least one of these is a saddle point. The change in stability and number of equilibria elicits Turing instability and causes the aforementioned patterns to emerge. This unstable region is depicted in Figure 3.9 between the dotted and solid lines. Considering a fixed kill rate, the system transitions from blue to red as f increases at the upper solid line through saddle-node bifurcation wherein the two equilibria collide and annihilate. This explains why the transition is so sudden in this region. As f decreases through the dotted line, Hopf bifurcation occurs and a branch of periodic solutions is formed. These create a larger region of interesting behaviour.

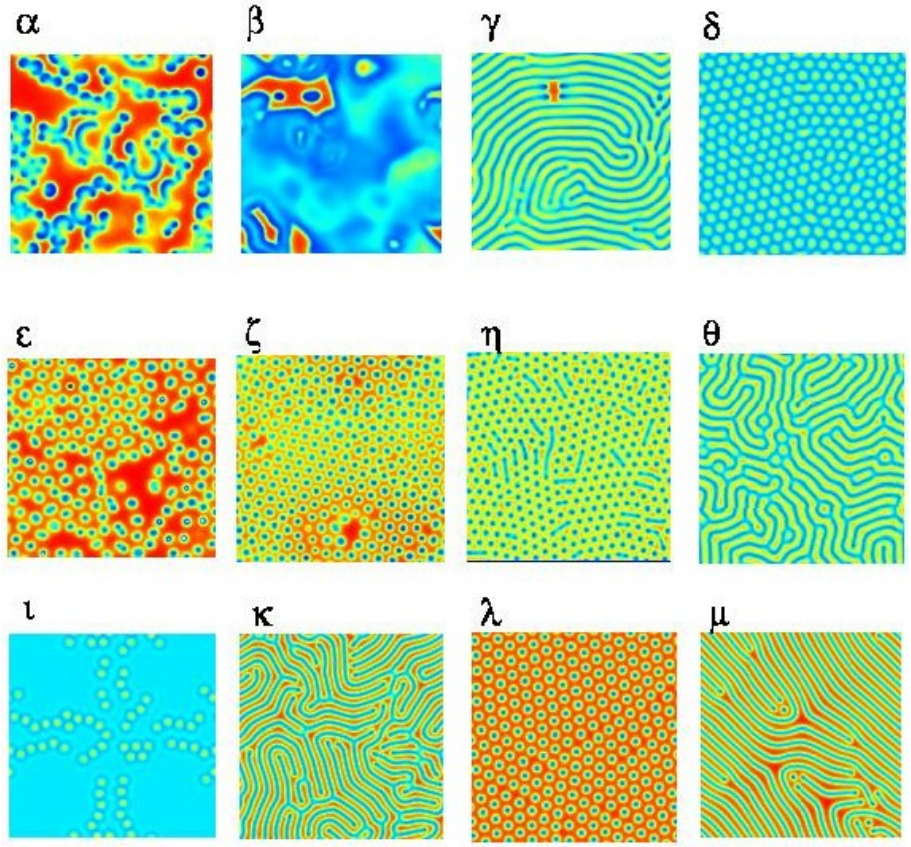


Figure 3.8: Pearson's 12 categories of Gray-Scott systems[20]

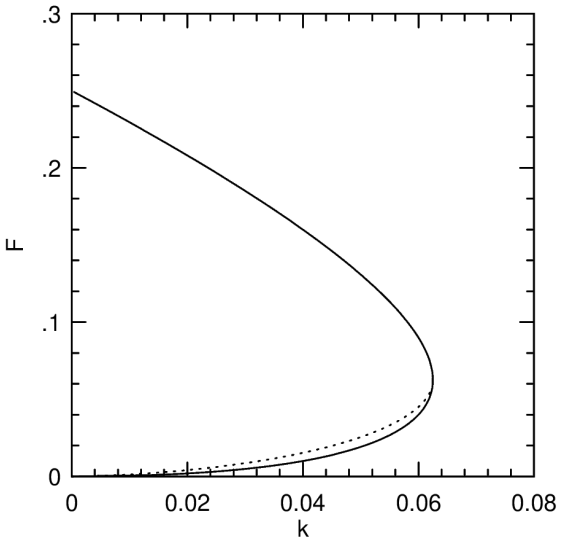
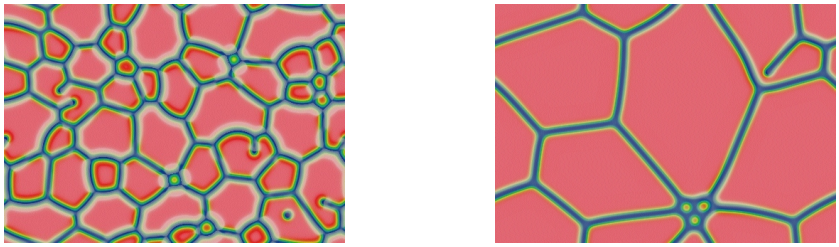


Figure 3.9: Phase diagram of Gray-Scott systems[20]

However, Pearson's analysis is limited by the initial condition he uses. Contrarily, Munafo[53] is able to reveal 5 new types of pattern differentiated from others by oscillation and spot shape by using a single central patch of ($u = 0$, $v = 1$) on an otherwise red background (i.e. an inversion of Pearson's initial condition) or several spots of diverse (f, k) values. One example is the ρ pattern shown in Figure 3.10. In imprecise terms, the feed rate controls oscillation, stability, and chaos while the kill rate controls the shape and quality of objects formed.

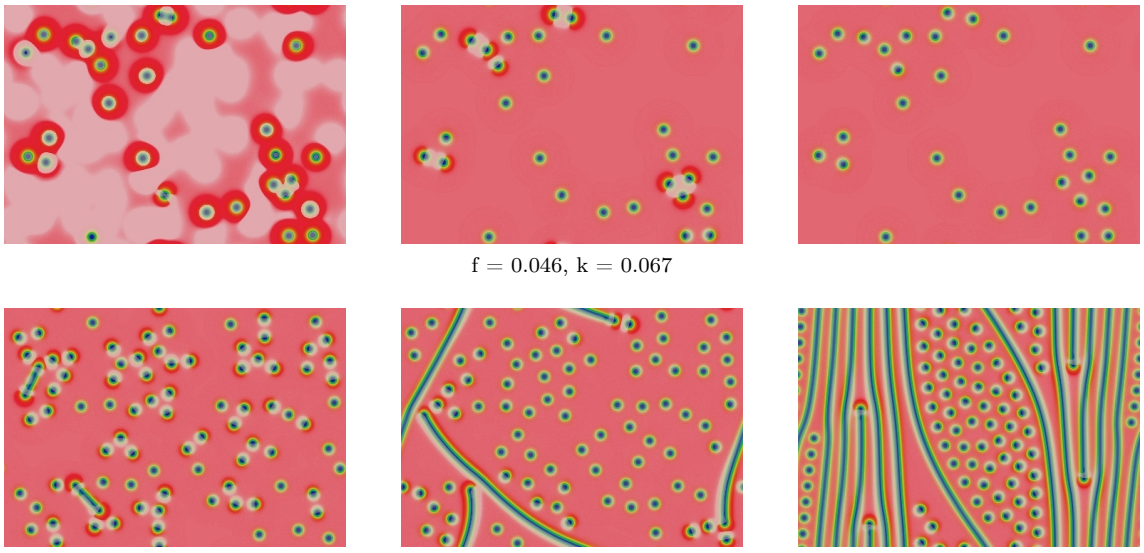


(a) f = 0.090, k = 0.059 (b) f = 0.102, k= 0.055

Figure 3.10: The ρ class of Gray-Scott pattern resembles a set of soap bubbles under surface tension. These clearly do not resemble any of the 12 Pearson categories. [21]

Munafo also presents a mapping from the Pearson / Munafo classes to the universal complexity classes established by Wolfram. However, due to the clear variety of behaviour in continuous state systems, three additional subclasses are presented.

- Class 2-a: These combine features of class 1 and 2. Certain starting conditions look like class 2 but eventually induce cascades which lead to asymptotic stasis as shown in Figure 3.11.
- Class 3-a: A subset of class 3. Although the patterns formed are relatively homogenous after a certain period like all class 3 systems, these patterns do not exhibit unending chaos. Instead, there are areas of long-lived localized stability through which chaos propagates as shown in Figure 3.12. However, these are not class 2 or 4 as the rate of change approaches a non-zero constant.
- Class 4-a: These are hypothetical class 4 systems that are subject to the halting problem. No example has yet been proven.



f = 0.046, k = 0.067

f = 0.066, k = 0.065

Figure 3.11: Class 2 behaviour (top) against class 2-a behaviour (bottom). Time moves left to right. [21]

One parameterisation of particular interest is (f = 0.0620, k = 0.0609). Under this setting, certain patterns can be observed that move indefinitely until intercepted. These U-shaped patterns called *skates* closely resemble the gliders in Conway's Game of Life and open the door to questions of universal computation.

Finally, Figure 3.13 shows a schematic map of the (f, k) parameter space with the complex, crescent-shaped region showing the variety of patterns that can exist.

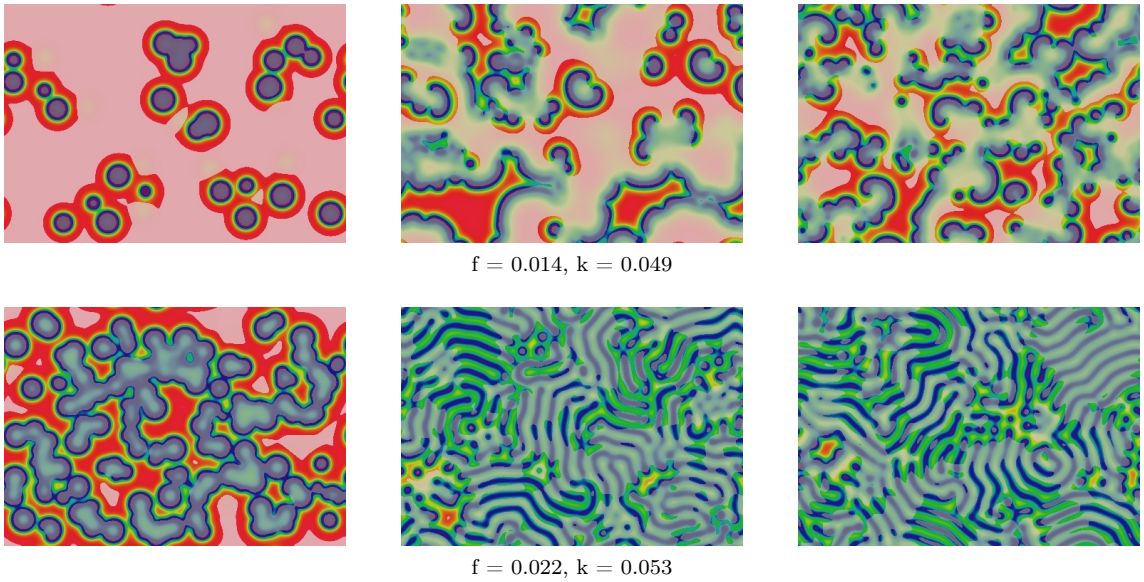


Figure 3.12: Class 3 behaviour (top) against class 3-a behaviour (bottom). Time moves left to right. [21]

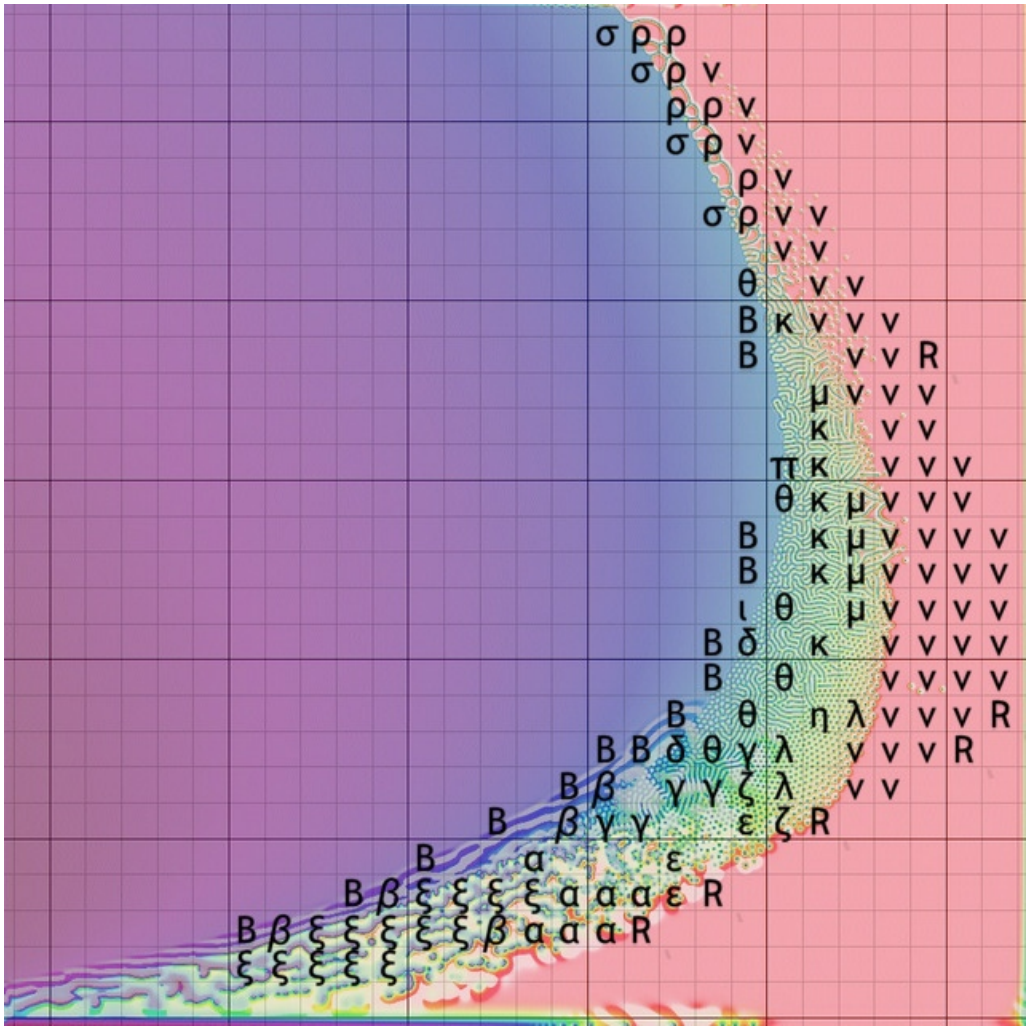


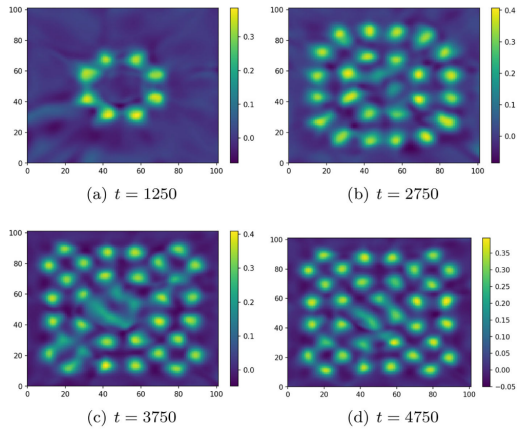
Figure 3.13: Map of Gray-Scott parameter space depicting all 19 Pearson-Munafo classes [22]

3.4 Gray-Scott Systems: Learning

As a system of partial differential equations, Gray-Scott systems are typically solved using numerical analysis such as finite difference methods[54]. There are not many attempts learn solutions using machine learning, possibly due to the success of state-of-the-art numerical methods at solving such systems.

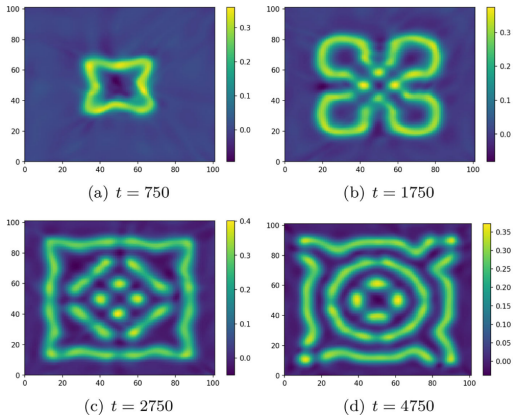
One key work that does use machine learning is *Physics-Informed Neural Networks Approach for 1D and 2D Gray-Scott Systems (Giampaolo et al., 2022)*[23] as shown in Figure 3.14. Finite difference methods are used to produce a constrained search space of physically feasible solutions for a neural network to learn on. In particular, physics-informed neural networks (PINNs) use a residual loss that encodes the divergence of a solution from the physical laws underlying the problem. The network aims to become a surrogate model of the true system and can be trained in a forward or inverse setting. In this work, at each of n time points, forward difference methods are used to obtain the correct "known" state which is fed into the surrogate model. This is to avoid the system converging to local minima under extended simulation times. Without corrections based on known data, the final Gray-Scott system exhibits trivial state in which the reactant either disperses entirely or overwhelms the domain. The loss function depends on the differential equation, the initial conditions and the known data.

The ANN architecture consists of 4 hidden layers of 20 neurons each with a *tanh* activation on input and *sigmoid* activation on output. The initial and boundary conditions are of the same form as Pearson's analysis and 10 known data collections are performed at 500 time step intervals. When testing on 4 common parameterisations, the system can mimic the patterns produced reasonably effectively with RMSE values between 0.1 and 0.3. As with many deep learning approaches, this method is computationally intensive with execution times of about 15 minutes on a GPU NVIDIA GeForce RTX 3080 with CPU intelcore i9-9900k and 128 GB of RAM. However, PINNs possess better ability to generalise compared to "blind" neural networks, due to the physical constraints imposed in the loss function.



(a) $t = 1250$ (b) $t = 2750$ (c) $t = 3750$ (d) $t = 4750$

(a) Mitosis pattern (f = 0.028, k = 0.062)



(a) $t = 750$ (b) $t = 1750$ (c) $t = 2750$ (d) $t = 4750$

(b) Soliton pattern (f = 0.030, k = 0.060)

Figure 3.14: Patterns predicted using PINN [23]

Chapter 4

Procedural Maze Generation

Figure 4.1: Example maze generated by our program. The start cell is yellow and goal state is red. Base CA: B1/S2345, Bias $\lambda = 0.6$

We build an evolutionary algorithm toolkit to evolve life-like CA for procedural generation. Maze generation is picked as an appropriately challenging domain since the quality of a maze is dictated by global properties. Figure 4.1 is an example of a maze generated by our program which we use throughout this chapter. These properties cannot be directly encoded in the chromosome of life-like CA. To generate a good maze, a CA rule must induce the birth and survival of cells at times that increase the likelihood of corridor and junction-like structures forming. Mazes possess many structural similarities to life-like CA including a lattice shape and two mutually-exclusive states, *wall* and *path*. Furthermore, two mazes with similar global properties tend to be phenotypically alike which simplifies evolutionary learning. However, CA are stochastic systems and provide no guarantees that any maze generated will be solvable. To mitigate this, we implement a repair mechanism that amends near-valid solutions and discards solutions that do not meet a minimum criterion. This allows genes that contribute to high quality solutions to survive while maintaining a level of selection pressure on solution validity alongside solution quality.

The task of CA-based maze generation has some similarities to a work by C. Adams[55]. However, this application differs notably from Adams' in both the evolution algorithm and fitness function design. Key features in our maze generator include the notion of failed rules, the stochastic region merging algorithm, and automated loss calculation (i.e. no external human input required to rank mazes).

The EA toolkit is written entirely in Python and features the following key classes:

- Experiment Suite: Includes methods and configurations for running evolutionary experiments.

- Population: Handles population-wide actions like crossover, elitism, and selection. It also tracks metrics for hyperparameter tuning.
- Chromosome: Handles individual actions like mutation, and fitness calculation.
- Simulator: Runs binary-state CA, performs repair, and caches state data when needed.

Each time we learn a new class of CA or optimise for a new objective, we write a new instance of the Population, Chromosome, and Simulator classes.

4.1 Simulator

Due to the undecidability of various CA rules, the state of a CA after some time cannot, in general, be calculated without simulating each transition in turn. For this, a CA simulator was built. Figure 4.2 shows snapshots from an example simulation.

(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$ (e) $t = 4$

Figure 4.2: 5 consecutive snapshots of *Stains*, a stable rule with rulestring B3678/S235678, created in the simulator

The simulator stores the state of a 2D square CA of side length N in an $N \times N$ matrix. The CA is initialised with birth set B and survival set S which are given as direct arguments or calculated from the chromosome. When simulating n time steps, the simulator begins by caching the current state $X^{(t)}$. Then a neighbourhood matrix M is calculated by convolving $X^{(t)}$ with kernel κ where

$$\kappa = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{4.1}$$

The value $M_{i,j}$ is the number of live neighbours of $X^{(t)}_{i,j}$. The convolution is calculated with wrapped boundaries to simulate periodic boundary conditions. The next state is calculated as follows

$$X^{(t+1)}_{i,j} = (\neg X^{(t)}_{i,j} \wedge n \in B) \vee (X^{(t)}_{i,j} \wedge n \in S) \tag{4.2}$$

where the left conjunction corresponds to the case of a dead cell becoming alive and the right conjunction corresponds to a living cell surviving. If $X^{(t+1)} = X^{(t)}$, where $X^{(t)}$ is the cached state, then no further steps are calculated since $X^{(t+n)} = X^{(t+n-1)} = ... = X^{(t)}$ by induction. Otherwise, the current state is cached and the simulator continues until n steps have elapsed or a later fixed point is reached. The simulator does not automatically detect periods of length greater than one step as this would involve maintaining and comparing against a large history of previous states which would be too computationally intensive. The simulator allows the initial state $X^{(0)}$ to be set randomly with a particular density or to be set explicitly.

4.2 Procedural Generation

A maze is generated from a chromosome in three stages: growth, region search, and region merging. During the growth stage, a CA is run in the simulator for a fixed number of iterations, typically 50, using the birth and survival sets encoded in the chromosome.

4.2.1 Region Search

The region search stage, defined by Algorithm 2, uses randomly initialised breadth-first searches to find all disconnected regions within the maze. In order to perform the merge stage later on, two data structures are populated during the search stage. The first is a hashmap from each cell to the index of the region it occupies. The second is the reverse, a hashmap from each index to the set of cells in that region.

Algorithm 2 Region Search Algorithm

Require: X - the state of the CA after the growth stage
Ensure: cells$[(c_x, c_y)] = r_c \iff (c_x, c_y) \in$ regions$[r_c]$

▷ Initialisation

```
cells ← empty dictionary of type {(int, int): int}
regions ← empty dictionary of type {int: set{int}}
spaces ← set of cells in X with state 0
```

▷ Find first region

```
r_1 ← BFS(start-cell, X)
UpdateDicts(r_1, 1)
```

▷ Find remaining regions

```
counter ← 1
while spaces not empty do
    counter ← counter + 1
    startCell ← randomly chosen 0-state cell
    r ← BFS(startCell, X)
    UpdateDicts(r, counter)
end while
```

▷ Update Function

```
procedure UpdateDicts(region, index)
    for c in region do
        cells[c] = index
        regions[index].add(c)
    end for
    spaces ← spaces - r
end procedure
```

Figure 4.3 shows the region search algorithm.

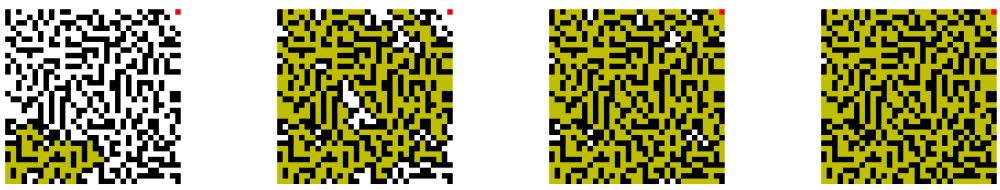


Figure 4.3: Snapshots from region search algorithm. Yellow represents visited path cells and red represent the goal cell. Time moves from left to right.

4.2.2 Region Merge

The region merge stage, defined by Algorithm 3, connects regions randomly until a connected path exists from the start cell to the goal cell or the maze is deemed invalid. It is crucial for the region merging algorithm to be stochastic. If it is deterministic and merges regions according to a pre-designed pattern, the genetic algorithm is incentivised to learn rules that lend themselves well to this pattern. For example, if mazes with longer solution paths are considered fitter and the merging algorithm connects regions in horizontal bands sweeping left-to-right (as in [55]) then the evolutionary process is incentivised to produce rules with shorter horizontal corridors over longer vertical corridors. This is in direct conflict with the fitness function. To avoid this, we design a

stochastic region merging algorithm. It begins with the region containing the start cell. Each wall cell bordering this region is examined to determine whether removing the cell would connect to a distinct region. One of these wall cells is randomly chosen and removed. This process repeats on the union of the two joined regions. If no such wall cells exist, the simulation is deemed invalid. This can be thought of as only permitting walls of width 1 to be broken when connecting regions. Allowing walls of arbitrary thickness to be joined is undesirable as it diminishes the contribution of the CA over the final solution. Consider the extreme case of a maze with only wall cells except the start and end goal. When the merging algorithm connects the start and end, the solution's form is a consequence of the merging algorithm alone and not the CA which generated a maze of walls in the first place. This would clearly be undesirable. If a chromosome does not yield valid simulations at least 80% of the time, it is assigned a fitness of 0 and usually removed from the population in the following iteration.

Algorithm 3 Region Merge Algorithm

```
Require: cells, regions, X
  visited ← regions[1]
  while True do
    fringe ← OneNeighbours(visited)
    if goalCell in fringe then
      return True                                   ▷ Success
    end if
    candidates ← []
    for f in fringe do
      zeros ← ZeroNeighbours(f)
      if length(zeros - visited) > 0 then
        candidates.append(f)
      end if
    end for
    if length(candidates) > 0 then
      c ← PopRandom(candidates)
      visited.add(c)
      X[c] = 0
      newRegions ← {cells[d] for d ∈ ZeroNeighbours(c)}
      visited ← visited ∪ {regions[r] for r ∈ newRegions}
    else
      return False                                  ▷ Failure
    end if
  end while
```

where OneNeighbours and ZeroNeighbours return the 1-state and 0-state neighbours of a cell respectively.

Figure 4.4 shows the region merge algorithm.

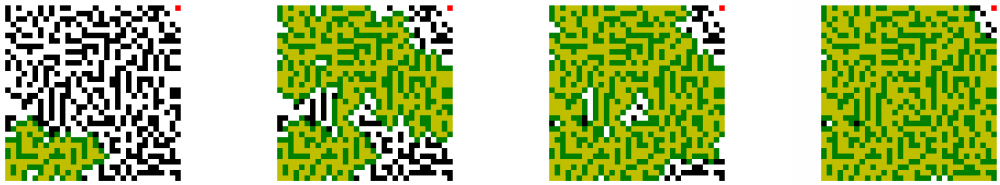


Figure 4.4: Snapshots from region merge algorithm. Yellow represents visited path cells, red represents goal cell and green represent fringe wall cells that can be considered for destruction if they border the current region and an unvisited region. Time moves from left to right.

4.3 Genetic Algorithm I

A genetic algorithm is used to evolve a population of chromosomes that represent the transition rule of the life-like CA. We explore the chromosome representation, genetic operators, and fitness function in this section.

4.3.1 Chromosome

A transition rule can be represented in a number of ways. Most intuitively, consider a birth and survival set indicating the number of neighbours that elicit a dead cell to become alive or a living cell to remain alive respectively. This can be encoded as a binary string or in integer form as shown in Figure 4.5.

Number of neighbours:	0 1 2 3 4 5 6 7 8 | 0 1 2 3 4 5 6 7 8
Binary representation:	0 0 0 1 0 0 0 0 0 | 0 0 1 1 0 0 0 0 0
	↑ ↑ ↑
Set representation:	B:{3} S:{2, 3}
Integer representation:	`0b000100000001100000` = 16480

Figure 4.5: Different representations of a life-like CA chromosome

Each binary string chromosome has length 18, so the discrete search space is of size $2^{18} = 262144$. An initial population of μ chromosomes is chosen randomly from a distribution that is uniform over the density of the binary representations. This is preferable to a distribution that is uniform over the integer representations[49]. When initialising a random chromosome, a density ρ is picked uniformly from $[0, 1]$. Then, each bit is a sample from the $Bernoulli(\rho)$ distribution.

4.3.2 Genetic Operators

At each iteration, the algorithm explores the search space through a collection of genetic operators. Crossover and mutation produce novel candidates, fitness functions evaluate each individual, and selection concentrates learned information in a set of elite parents. We produce λ new children in each expansion stage and reduce down to μ elite candidates in each contraction stage with $\lambda \approx 4\mu$ as advised by Hansen et al.[56]. Single-point crossover is used to produce new children. Pointwise mutation is applied with probability $\frac{1}{18}$ such that the expected number of bit flips per chromosome is 1. For efficiency, this is implemented by generating a mutation mask with expected density $\frac{1}{18}$ and applying XOR between the chromosome and mutation mask.

Common forms of selection include roulette and truncation. In truncation selection, all candidates are linearly ranked by fitness and the top μ candidates progress to the next generation. In roulette selection, the probability of each individual progressing to the next generation is proportional to its objective fitness. Both are implemented and compared in Chapter 7.

4.3.3 Fitness Function

The aim of the fitness function is to assess the maze using quantitative metrics that can be calculated efficiently. For example, the number of vacant cells reachable from the start cell is an important metric because, if this is too low, a large portion of the maze is wasted space. This metric is computed in linear time with respect to the number of cells in the CA by performing a breadth-first search in the region containing the start cell and recording the number of vacant cells in the maze that exist outside this region. We opt for two metrics: the number of dead ends and the solution path length. A cell is considered to be a dead end if all its neighbours are wall cells or vacant cells that have already been visited. These two factors work well as they are easy to calculate, well-correlated with perceived maze difficulty[55] and, crucially, oppose each other. A maze with a long solution tends to have long corridors, whereas a maze with many dead ends

tends to have shorter corridors and more decisions to make at each junction. By optimizing on conflicting objectives, we automatically regulate learning speed and reduce the probability of premature convergence to local optima. Assuming a maze with at least one solution, both metrics can be calculated simultaneously in a single breadth-first search traversal. Figure 4.6 shows this process.

Figure 4.6: Snapshots from BFS algorithm calculating number of dead ends. The green cells represent dead ends discovered. Time moves left to right.

Initially, the fitness function $f(c_i) = s + \lambda d$, where s is the solution path length and d is the number of dead ends, was considered. However, this is not normalized as the solution length and number of dead ends are not on the same scale. Moreover, we cannot normalize each metric individually based on the range of values present in a given generation since the ranges vary from experiment to experiment and generation to generation. Instead, a truncated linear selection is performed with fitness function $f(c_i) = r_s + \lambda r_d$ where r_s and r_d are the rank of the cell in the population according to solution length and number of dead ends respectively.

Chapter 5

Modelling Life-Like CA

Modelling life-like CA involves making observations on a surrogate CA produced from a candidate chromosome and comparing these with observations on a true CA produced from the goal chromosome on the same initial conditions. In this chapter we explore the genetic algorithm used to learn the full rule dynamics of life-like CA as visualized in Figure 5.1. We also elaborate on key decisions made regarding software design.

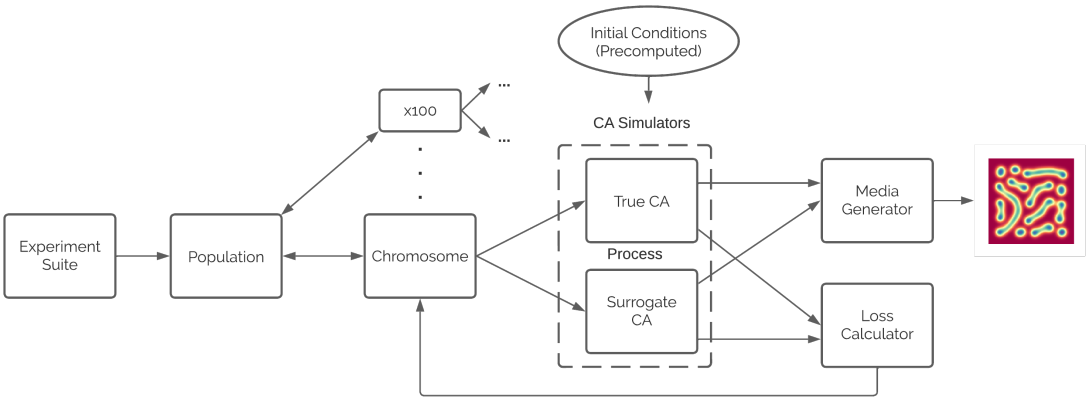


Figure 5.1: Process for learning Life-Like and Gray-Scott CA

5.1 Genetic Algorithm II

In this case, the optimization goal is to find the transition rule that generated the observations made. We maintain the same genetic operators as Genetic Algorithm I using single point crossover and point mutation.

Selection comes in two flavours, *plus* and *comma*. Under $(\mu + \lambda)$ plus-selection, we produce a population of λ children from μ parents and the next generation is selected from the collective. Under (μ, λ) comma-selection, the next generation is selected exclusively from the λ children. We can interpret this as an age restriction of 1 generation on the μ parent candidates. While this promotes exploration outside of local optima, it is easy for promising solutions to die out if they do not immediately pass on their advantageous characteristics to the next generation. This hinders convergence.

Fitness is calculated by running multiple simulations with the given transition function. For an $N \times N$ CA, it is infeasible to test on all 2^{N^2} initial conditions. Instead, a sample is picked. To ensure fairness, all CA are tested on the same set of initial conditions sampled uniformly on densities in $[0, 1]$. In order to learn full rule dynamics, we design a fitness function that quantifies the ability of a candidate to convert the state observed in the goal CA at time t to the state observed at

time $t + \delta$ over δ time steps. Suppose K observations of the goal CA are made producing states $X_{\delta_1}, X_{\delta_2}..., X_{\delta_K}$ where the number of time steps between X_{δ_k} and $X_{\delta_{k+1}}$ is δ_{k+1}. We define the loss of a candidate between observations k and $k+1$ as the mean number of differing states cell-wise between $X_{\delta_{k+1}}$ and $\phi^{\delta_{k+1}}(X_{\delta_k})$ which is the state of the candidate CA initialised at X_{δ_k} when observed δ_{k+1} time steps after initialisation. The loss between each observation is in $[0, 1]$. The fitness is defined by taking the mean of the losses and subtracting from 1.

Definition 5.1 (Life-Like Fitness Function 1)**.** *We define the fitness F as*

$$F = 1 - \bar{L} \tag{5.1}$$

$$\text{where } \bar{L} = \frac{1}{N^2(K-1)} \sum_{k=1}^{K-1} \left| X_{\delta_{k+1}} \oplus \phi^{\delta_{k+1}}(X_{\delta_k}) \right| \tag{5.2}$$

where $\oplus$ is the XOR operator and we use $|X|$ to represent the number of living cells in state X.

The number of observations and the values of the inter-observation times (or "step sizes") are hyperparameters. If K is too high, we perform needless computations observing increasingly similar states as the CA stabilises. If K is too low, we only observe early transient patterns instead of the long-lived patterns that characterise the objective rule. With regard to step sizes, we consider 3 possibilities.

1. Constant: $\delta_1 = \delta_2 = ... = \delta_K = C$.
2. Random Uniform: $\delta_k \sim \mathit{Uniform}(D_{min}, D_{max})$
3. Random Increasing Uniform $\delta_k \sim \mathit{Uniform}(f_{min}(k), f_{max}(k))$

where f_{min} and f_{max} are monotonically increasing functions of k. While a constant stepsize is simpler to implement, a random uniform stepsize is less likely to conflate periodic patterns in the CA with convergence. For example, consider *Fumarole*, a 5-period oscillator in the Game of Life shown in Figure 5.2. If $C = 5$, the loss at each observation would be calculated using only one of its states. A rule that supports a still-life of the same configuration would be considered as optimal as the true rule. On the contrary, a random uniform stepsize with $D_{min} < 5 < D_{max}$ is extremely unlikely to land on the same state each time. The chances of this are only

$$\left(\frac{1}{D_{max} - D_{min}} \right)^K \tag{5.3}$$

Therefore, an algorithm with random uniform step size is much more likely to rank the true rule as fitter than the imposter. The random increasing uniform distribution goes a step further, increasing the expected value of δ_k as k increases to allow time for late-stage patterns to appreciably change before making further observations.

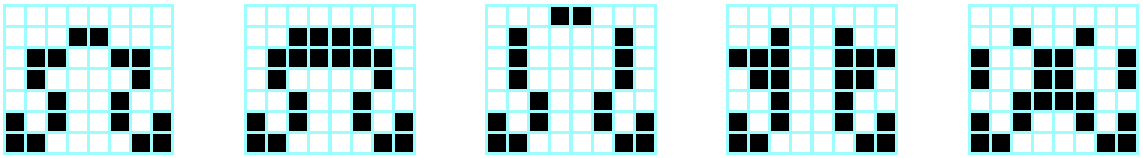
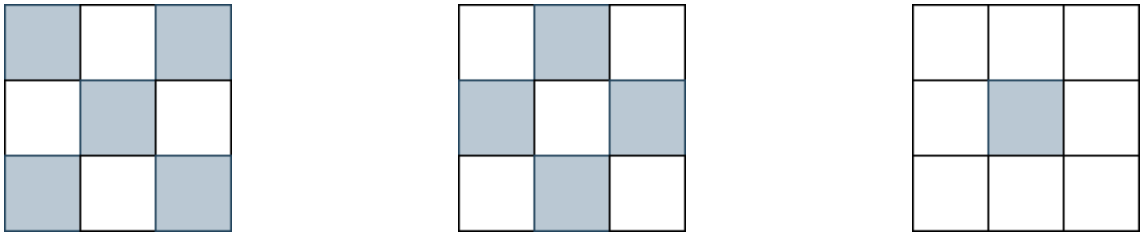


Figure 5.2: *Fumarole*, a 5-period oscillator in the Game of Life. [24]

However, a fitness function that compares states cell-wise can be too fine-grained. It fails to capture macroscopic properties such as the density of live cells across different regions in the lattice. As a simple example, Figure 5.3 shows two predictions for a goal state. Prediction 1 clearly has a similar density and pattern to the goal but its live cells do not align with the live cells of the goal. Prediction 2 only has a single live cell making it quite different from the goal but due to the position of that cell, it achieves a much higher fitness than prediction 1. To mitigate this affect, a multi-resolution fitness function is proposed which uses convolutions to capture density across broad regions.



(a) Goal State

(b) Prediction 1, fitness = 0

(c) Prediction 2, fitness = 0.55

Figure 5.3: Example of fine-grain loss failing to capture macroscopic properties

Definition 5.2 (Life-Like Fitness Function 2)**.** *We define the multi-resolution fitness F as*

$$F = 1 - \bar{L} \tag{5.4}$$

$$\text{where } \bar{L} = \frac{1}{N^2 M(K-1)} \sum_{k=1}^{K-1} \sum_{m=1}^{M} \left| \lfloor \omega_m * X_{\delta_{k+1}} \rceil \oplus \lfloor \omega_m * \phi^{\delta_{k+1}}(X_{\delta_k}) \rceil \right| \tag{5.5}$$

$$\text{where } \omega_m = \frac{1}{m^2} \begin{pmatrix} 1 & \dots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \dots & 1 \end{pmatrix} \in \mathbb{R}^{m \times m} \tag{5.6}$$

where $\omega * f$ is a 2D convolution over image f with filter kernel ω and $\lfloor . \rceil$ is the integer rounding operator.

The filter kernel used, ω_m, is the all-ones matrix divided by the size of the kernel. This ensures that $\omega_m * X$ has entries between 0 and 1 and that, after rounding, the convolution is a binary matrix with each cell representing whether there are more live or dead cells in an $m \times m$ region of the lattice. After XORing and summation, the loss $\bar{L}$ is between 0 and 1 and so is the fitness.

5.2 Software Engineering Design

Design decisions regarding algorithms and process have been discussed throughout Chapters 4 and 5. In this section, we summarise key software engineering design decisions when writing the evolutionary algorithm toolkit.

Figure 5.4 depicts the sequence of events that is triggered in a typical experiment learning life-like CA. There are three major loops that run during an experiment. The outer loop performs generational learning where the population is updated on each iteration. The inner loop calculates loss where each of the two CA is stepped forward by a certain number of time steps on each iteration. SurrogateCA is a subclass of CA with a `step_from` method that can set an explicit initial condition. This allows it to be reseeded with true data on each iteration of the inner loop. The final loop reruns top solutions to generate statistics and media.

One notable decision is about changes to fields in the Chromosome class during operations like mutation and crossover. To avoid repeated calculations converting between birth/survival sets and binary rule strings, we write the Chromosome class to keep track of both formats simultaneously. However, this requirement means both sets must be updated every time the binary rule string is updated and vice versa. Although Python allows direct access to private class fields, it is verbose and error-prone to update all fields every time a single field needs to be changed. We automatically enforce synchronisation between these fields through using `@property` decorators. We define a setter method that updates the birth and survival sets every time the binary rule string is updated. By decorating this with `@property`, the setter is implicitly called every time a modification is made to the rule string field. We implement the same for the birth and survival sets too.

Another key decision was about initialisation. Two common initialisation methods for chromosomes are based on sampling uniformly across density, and sampling uniformly across value.

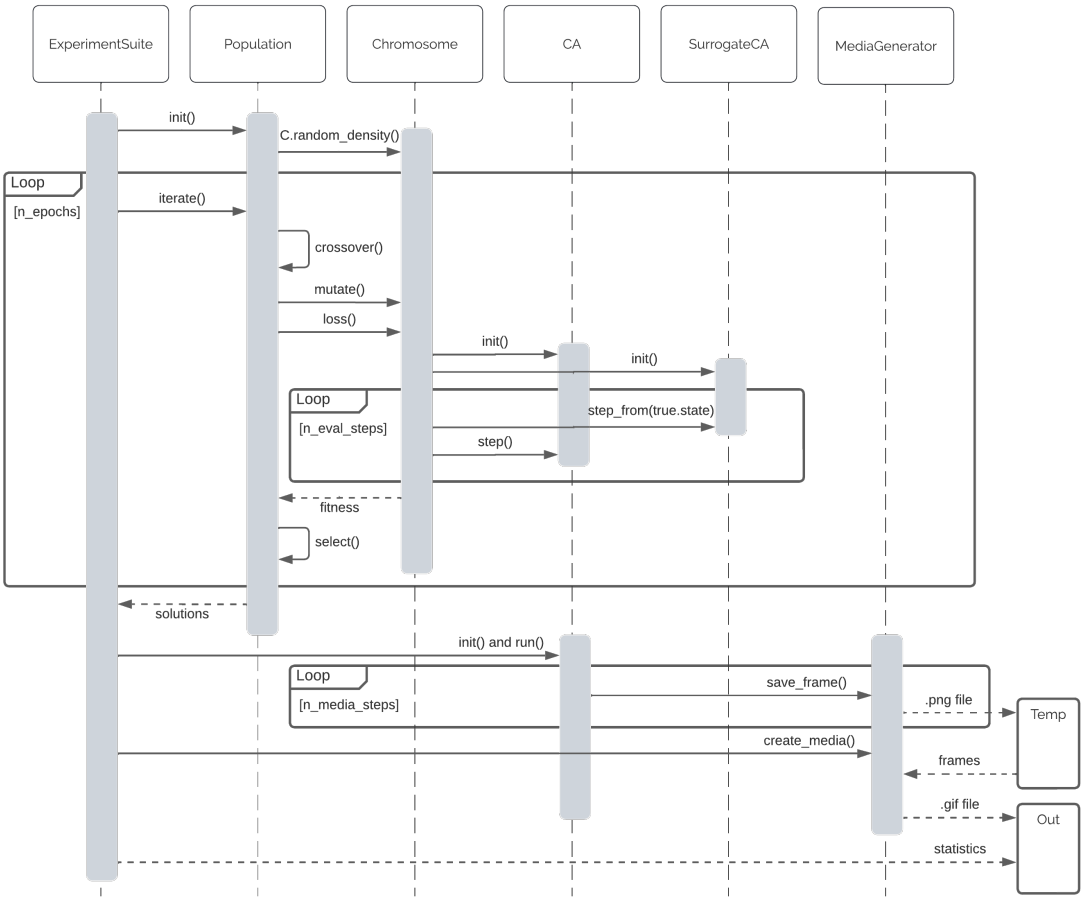


Figure 5.4: UML sequence diagram of life-like CA learning

By implementing both of these as factory methods, we create a extensible class where new initialisation methods can be easily implemented and tested. `Chromosome.random_density()` and `Chromosome.random_value()` are public factory methods that generate a binary rule string and call another internal factory method `Chromosome.from_rstring()` which calculates the birth and survival sets then passes all three parameters to the main constructor which creates a new `Chromosome` object.

Metrics and media are generated to extract insights from the toolkit. The experiment suite class configures populations according to user-defined parameters and records metrics from the population at each epoch in Pandas dataframes. These metrics are saved to a .csv file at regular intervals so that data is partially logged if the program fails midway through an experiment. Media is generated automatically once an experiment has finished. The top 3 solutions are simulated again. During simulation, a media save is triggered at regular intervals whereby the state of the running CA is converted into a 2D regular raster graphic using Matplotlib and saved as a png file in a temporary folder. If there are multiple stages of simulation (e.g. growth, region find, and region merge), one temporary folder is created per stage. This simplifies the generation of animated gifs and promotes extensibility in case more stages are to be added in the future. After simulation, these images are stitched together into animations using the Python Imaging Library (PIL). The final state of each CA is also saved as a png image and in its original array form as a .npy file. The .npy file is saved in case a later analysis requires the CA to continue running from where it stopped. The temporary frame files are deleted to save memory. Analysis and visualisation of population-wide properties is done manually after the experiments.

Chapter 6

Modelling Gray-Scott Systems

To learn Gray-Scott models, the evolutionary algorithm toolkit is extended with a new simulator, a modified genetic algorithm, and support for evolutionary strategies (ES). We explore each of these extensions in this chapter.

6.1 Simulator

In this setting, we run a discretized simulation of a continuous system so there are more subtleties to consider. We use two matrices of floats to store the density of u and v across the CA. To simulate the change in these densities efficiently, we use a pair of finite-difference equations that approximate the continuous partial differential equations defined in Def 2.5. We first define discrete approximations for the derivative of each density in Def 6.1.

Definition 6.1 (Gray-Scott Model)**.**

$$\dot{u} = -uv^2 + f(1-u) + r_u\mathcal{D}(u) \tag{6.1}$$

$$\dot{v} = uv^2 - (f+k)v + r_v\mathcal{D}(v) \tag{6.2}$$

Each equation has 3 terms. From left to right, these are the reaction term, the external term and the diffusion term. This is exactly the same as the continuous definition, Def 2.5, with the exception of the discrete Laplace operator $\mathcal{D}$ which replaces the continuous Laplacian Δ. Calculating the first two terms is simple enough as we are given the feed and kill rates. However, the third term must approximate the Laplacian which is usually computationally intensive to calculate. To do this, we use a kernel that, when convolved over the density matrices of u and v, produces matrices whose entries are approximations of $\Delta u(x,y)$ and $\Delta v(x,y)$ respectively. Different kernels have been used for this convolution in the literature. One example used in Compeau's Biological Modelling book[57] is

$$K = \begin{bmatrix} 0.05 & 0.2 & 0.05 \\ 0.2 & -1 & 0.2 \\ 0.05 & 0.2 & 0.05 \end{bmatrix} \tag{6.3}$$

Another popular choice is the nine-point stencil[58].

$$\Delta_h^{(9)} = \begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & -3 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix} \tag{6.4}$$

Testing on a few examples revealed that both kernels capture complex diffusion behaviour with similar effectiveness. However, the choice of kernel is not arbitrary and these two choices are well principled. We briefly outline the reasoning behind the nine-point stencil. If we consider u as a function of only x (i.e fix y and t constant) and rearrange the Taylor series expansions for $u(x+h)$ and $u(x-h)$, we get

$$\frac{\partial u}{\partial x} = \frac{u(x+h)-u(x)}{h} - \frac{u^{(2)}(x)}{2!}h - \frac{u^{(3)}(x)}{3!}h^2 - \frac{u^{(4)}(x)}{4!}h^3 - \ldots \tag{6.5}$$

$$\frac{\partial u}{\partial x} = -\frac{u(x-h)-u(x)}{h} + \frac{u^{(2)}(x)}{2!}h - \frac{u^{(3)}(x)}{3!}h^2 + \frac{u^{(4)}(x)}{4!}h^3 - \ldots \tag{6.6}$$

Subtracting one from the other yields the 3-point stencil approximation for the Laplacian in one dimension denoted $\Delta^{(3)}u(x)$.

$$0 = \frac{u(x+h) + u(x-h) - 2u(x)}{h} - u^{(2)}(x)h - 2\frac{u^{(4)}(x)}{4!}h^3 + ... \tag{6.7}$$

$$\implies \frac{\partial^2 u}{\partial x^2} = \underbrace{\frac{u(x-h) - 2u(x) + u(x+h)}{h^2}}_{=: \Delta^{(3)}u(x)} + O(h^2) \tag{6.8}$$

By combining the 3-point stencils for the x and y directions, we get a two dimensional 5-point stencil.

$$\Delta u(x,y) = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \tag{6.9}$$

$$\approx \frac{u(x-h,y) - 2u(x,y) + u(x+h,y)}{h^2} + \frac{u(x,y-h) - 2u(x,y) + u(x,y+h)}{h^2} \tag{6.10}$$

$$= \frac{u(x-h,y) + u(x+h,y) - 4u(x,y) + u(x,y-h) + u(x,y+h)}{h^2} \tag{6.11}$$

$$= \frac{1}{h^2}\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} u(x,y) = \Delta_h^{(5)} u(x,y) \tag{6.12}$$

Finally, we obtain a softer discretization by combining two 5-point stencils, shown in 6.12, into a 9-point stencil. Consider $\Delta_h^{(5)}u$ with $h = 1$, the side length of a cell in our lattice. Then, the stencil aligns perfectly with our CA. Now consider another stencil rotated by 45° with $h = \sqrt{2}$. By combining these two stencils, we get

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} + \frac{1}{(\sqrt{2})^2}\begin{bmatrix} 1 & 0 & 1 \\ 0 & -4 & 0 \\ 1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & 0.5 \\ 1 & -6 & 1 \\ 0.5 & 1 & 0.5 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0.25 & 0.5 & 0.25 \\ 0.5 & -3 & 0.5 \\ 0.25 & 0.5 & 0.25 \end{bmatrix} = \frac{1}{2}\Delta_h^{(9)} \tag{6.13}$$

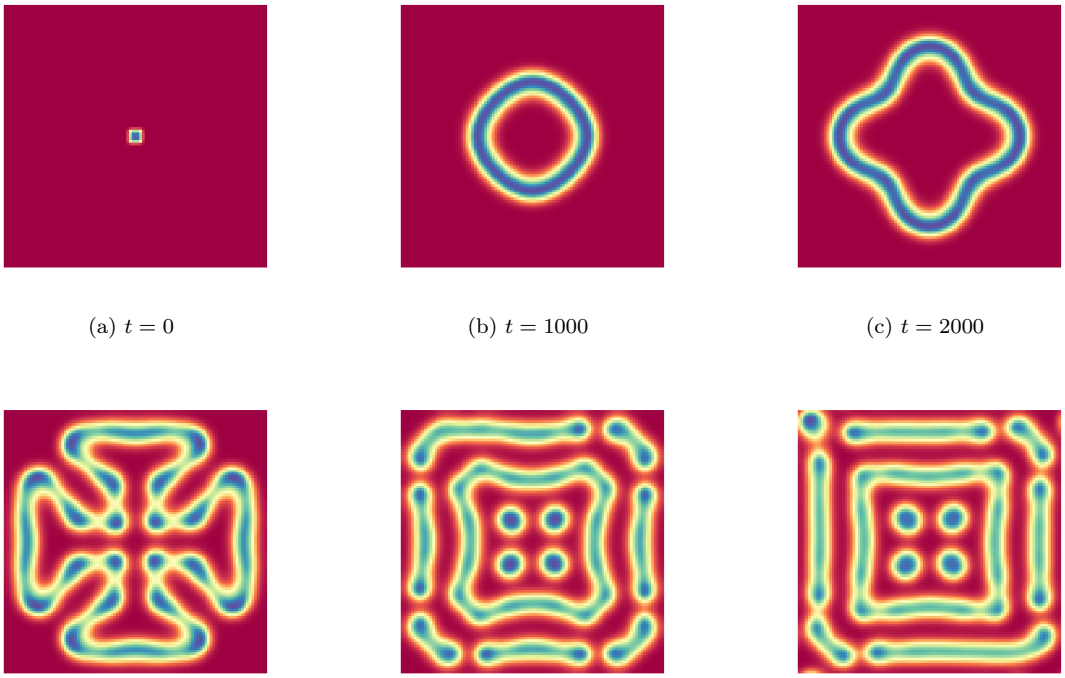
In this sense, the 9-point stencil approximates the second partial derivative across 8 directions. The positive and negative x and y directions as well as the 4 diagonal directions between them. The constant factor of $\frac{1}{2}$ is consumed by the diffusion constant. We can now convolve $\Delta_h^{(9)}$ over the density matrices of u and v to approximate Δu and Δv and therefore calculate $\dot{u_n}$ and $\dot{v_n}$ from Def 6.1. We iterate using forward Euler integration as follows

$$u_{n+1} = u_n + \dot{u}(u_n, v_n)\ \delta t \tag{6.14}$$

$$v_{n+1} = v_n + \dot{v}(u_n, v_n)\ \delta t \tag{6.15}$$

There are a number of constants that affect the behaviour of simulation. These include the diffusion constants, time delta, initial densities, feed rate, and kill rate. We pick a diffusion constant ratio of $D_u = 2D_v$ as this ratio has been shown, in the literature, to elicit complex behaviour[57]. The temporal resolution can be adjusted using the time delta δt. Through spot checks, we find values of $\delta t \in [0.5, 1.0]$ produce interesting behaviour in a reasonable number of epochs (i.e. under 10000). The remaining constants are parameters of the simulation passed in by the calling Chromosome class. As before, we convolve with wrap boundaries to ensure periodic boundary conditions and we cache states during simulation to reduce computation time. When visualising state, we set the colour of a cell based on the ratio of densities of u and v inside it.

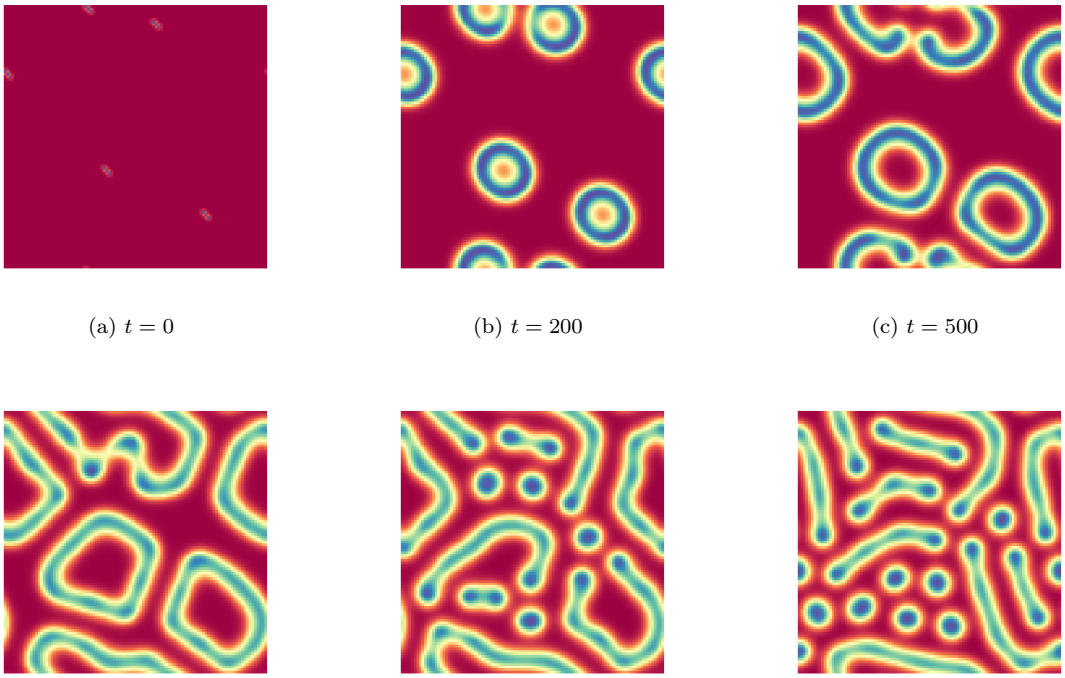
The simulator allows for two different initialisation settings. Both have a background state populated entirely of reactant ($u = 1$, $v = 0$) and apply perturbations of ($u = \frac{1}{2}$, $v = \frac{1}{4}$). The *patch* setting, in the style of Pearson[20], creates a square perturbation with ±1% Gaussian noise as shown in Figure 6.1. The *splatter* setting produces n perturbation "seeds" of size 3×3 as shown in Figure 6.2. The patch setting tends to unfold in a reproducible manner as the only source of randomness in the small Gaussian noise. The splatter setting is unpredictable as the final outcome is dependent on the initial seed locations and way in which they collide.



(a) $t = 0$ (b) $t = 1000$ (c) $t = 2000$

(d) $t = 3000$ (e) $t = 4000$ (f) $t = 5000$

Figure 6.1: Gray-Scott simulation under *patch* initialisation ($f = 0.03$, $k = 0.06$)



(a) $t = 0$ (b) $t = 200$ (c) $t = 500$

(d) $t = 1000$ (e) $t = 2000$ (f) $t = 5000$

Figure 6.2: Gray-Scott simulation under *splatter* initialisation ($f = 0.03$, $k = 0.06$)

6.2 Genetic Algorithm III

6.2.1 Chromosome

The chromosome is made up of two vectors, *state* and *control*. The state vector contains the parameters of the model, feed and kill rate, as two real numbers, f and k. The control vector contains the volatility of these parameters as real numbers df and dk. This allows us to implement a variable mutation rate which is important given the search space is made up of a small band of interesting behaviour amidst a background of trivial solutions. The toolkit allows the user to initialise chromosomes in one of two ways. The first choice is to sample a random uniform distribution with $f \in [0.0, 0.30]$ and $k \in [0.0, 0.08]$. The boundaries are based on Munafo's phase diagram in Figure 3.13. The second choice is to sample pairs close to the saddle-node bifurcation threshold $f = f(4+k)^2$ depicted as the solid line in Figure 3.9. Values of f are picked uniformly between the roots of the bifurcation parabola, 0.0 and 0.25. k is the positive solution of this parabola $k = \frac{1}{2}\sqrt{f} - f$ perturbed by $\pm 10\%$ Gaussian noise and truncated at $k = 0$.

6.2.2 Genetic Operators

The crossover method used is a combination of uniform crossover and blended crossover (BLX-α). In uniform crossover, the state of the child c is defined by a value sampled uniformly from the interval between the corresponding state in the parents, x and y, as shown below.

$$c_f \sim \mathit{Uniform}(\min(x_f, y_f),\ \max(x_f, y_f)) \tag{6.16}$$
$$c_k \sim \mathit{Uniform}(\min(x_k, y_k),\ \max(x_k, y_k)) \tag{6.17}$$

However, the control property is different due to the BLX-α algorithm which combines exploration and exploitation. A traditional crossover algorithm seeks to exploit generational knowledge to create novel candidate solutions by deriving a child state interior to the boundaries defined by the parents' states. For example, consider a single-point crossover on a binary string. If both parents have a 1 in the i^{th} position bit, it is impossible for the child to have a 0 in the i^{th} bit. In this sense, BLX-α is not pure crossover as it explores areas of the search outside the parent-defined borders. BLX-α is defined in Def 6.2 and visualised in Figure 6.3.

Definition 6.2 (Blended Crossover (BLX-α))**.** *We define the blended crossover of real numbers $x_i,\ y_i \in [a_i, b_i]$ with $x_i < y_i$ and parameter $\alpha \in \mathbb{R}^+$ as a sample*

$$BLX(x_i, y_i, \alpha) \sim \mathit{Uniform}(x_i - \alpha(y_i - x_i),\ y_i + \alpha(y_i - x_i)) \tag{6.18}$$

The hyperparameter α defines how much exploration the operator undertakes as a fraction of the difference between the parent states. In the special case $\alpha = 0$ this is uniform crossover. When α is maximised, this approaches a uniform mutation operator where the value of a child gene is a uniform sample across the predefined gene boundaries a_i and b_i.

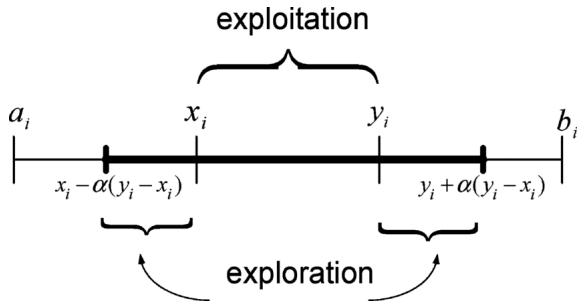


Figure 6.3: Blended Crossover [25]

To ensure exploration is occurring for state properties too, a Gaussian mutation is added to f and k. The variance of this mutation is dictated by the control property.

6.2.3 Fitness Function

As before, we simulate a true CA based on the goal parameters and maintain a surrogate CA which is regularly synchronised with the true CA. The loss for a single cell is calculated by finding the difference in density between the true and surrogate cell as a fraction of the true density. The loss of the whole CA is the mean of these values and the fitness is defined as $1 - \text{loss}$. This is a continuous extension of the XOR loss used for life-like CA.

6.3 Evolutionary Strategy

Evolutionary strategies (ES) are black-box optimization algorithms suited to continuous domains. Some ES use fitness-based recombination where parents with a higher fitness produce more offspring[56]. In this case, selection pressure and exploitation of generational knowledge are pursued simultaneously, so traditional environmental selection can be omitted. For our use case, we stick with a fitness-independent recombination method and a separate environmental selection scheme to allow flexibility in tuning both processes individually.

An ES individual has the form $a_i : (\boldsymbol{x}, \boldsymbol{\sigma}, F(\boldsymbol{x}))$ which are the state parameters, control parameters, and fitness respectively. The control parameters control the average size of mutation and are specific to each individual. ES also often use more than 2 parents for recombination unlike GA. Typically, all children are produced from the top ρ parents with $\rho \leq \mu$. These are called the mixing parents. Common recombination operators include:

- Discrete: Each of the offspring's state variables is inherited from one of the ρ parents.
- Intermediate: The offspring's state is the mean of the states of the parents.
- Weighted Intermediate: The offspring's state is a weighted combination of the parent states with a weighting function that increases monotonically with fitness.

A fundamental idea in ES is parameter control by self-adaptation. For a self-adapting ES, control parameters are evolved just like state parameters. Each of the λ offspring are generated as follows

$$b_i \leftarrow \begin{cases} \boldsymbol{x}_i = \bar{\boldsymbol{x}} + \boldsymbol{\sigma}_i \boldsymbol{N_i(0, I)}, \\ \boldsymbol{\sigma_i} = \bar{\boldsymbol{\sigma}} e^{\tau \boldsymbol{N_i(0,I)}}, \\ F_i = F(\boldsymbol{x}_i) \end{cases} \tag{6.19}$$

Where we define the intermediate recombination operator as the arithmetic mean

$$\bar{\boldsymbol{y}} = \frac{1}{\rho} \sum_{m=1}^{\rho} \boldsymbol{y}_m \tag{6.20}$$

where $\boldsymbol{y}_m$ is the $\boldsymbol{y}$ property of the m^{th} best mixing parent by fitness. The state vectors receive Gaussian mutation with variance defined by the control parameters and the control parameters receive log-normal mutation with variance dependent on a constant learning rate τ. This is usually set proportional to $\frac{1}{\sqrt{n}}$ but can be tuned as a hyperparameter. The terms $\boldsymbol{N_i(0, I)}$ are normally distributed vectors with the same dimensions as the vector being mutated.

We implement this strategy but, in the style of Hansen et al.[56], we apply mutation in two parts. There is a common mutation applied to all control parameter components of $e^{\tau_1 N_i(0,1)}$ and a specific mutation applied to each component individually of $e^{\tau_2 \boldsymbol{N_i(0,I)}}$ with $\tau_1 = n^{-\frac{1}{2}}$ and $\tau_1 = n^{-\frac{1}{4}}$.

Chapter 7

Evaluation

In this chapter, we begin by outlining metrics used to evaluate individual CA, and EAs as a whole. We then systematically analyse the life-like CA rule space using these metrics. Finally, we evaluate the evolutionary algorithms developed in this project against each goal. We establish which types of rules are easier or harder to learn based on our systematic analysis.

7.1 Metrics

Alongside properties discussed in Sections 3.1 and 3.3, we design metrics to categorise CA and assess the effectiveness of evolutionary algorithms.

7.1.1 Cellular Automata

First, we consider metrics for individual CA. These are: periodicity, volatility, and density.

By estimating the portion of initial conditions that enter a periodic cycle, and the speed with which they do so, we can infer a CA's Wolfram class. The key distinction we intend to make is between CA in class 1 or 2 and CA in class 3 or 4. The former two classes should converge relatively quickly for almost all ICs whereas the latter two will not converge for a significant proportion of ICs.

Definition 7.1 (CA convergence to a periodic state)**.** *If a state previously visited at time step t is produced again at step $t + \delta$, the rule is said to have converged to a periodic solution at $t + \delta$ time steps with period δ. If $\delta = 1$, this is called a fixed point solution.*

For the trivial initial conditions with all zero-cells or all one-cells, all life-like CA will converge to a periodic state with $\delta \leq 2$. The proof of this is detailed in 7.1.1.

Definition 7.2 (Quiescence)**.** *A CA is quiescent if all cells are in the same state. A CA with each cell c_i in state $\sigma_i(t) = 0$ is denoted $\underline{\mathbf{0}}$ and the opposite quiescent CA with $\sigma_i(t) = 1$ is denoted $\underline{\mathbf{1}}$.*

Lemma 7.1. *A quiescent life-like CA is at a fixed point or oscillates with period 2.*

Proof. Consider an arbitrary cell c_i in $\underline{\mathbf{0}}$. $\sigma_i(t)$ is the state of c_i at time t and $n_i(t)$ is the number of live cells in the neighbourhood of c_i not including itself. Initial conditions are $\sigma_i(0) = 0$ and $n_i(0) = 0$.

If $0 \notin B$:
  $\sigma_i(1) = 0$ and $n_i(1) = 0$
  $\implies$ convergence to $\underline{\mathbf{0}}$ with period 1.

If $0 \in B$:
  $\sigma_i(1) = 1$ and $n_i(1) = 8$
  If $8 \in S$:
    $\sigma_i(2) = 1$ and $n_i(1) = 8$
    $\implies$ convergence to $\underline{\mathbf{1}}$ with period 1.
  If $8 \notin S$:
    $\sigma_i(2) = 0$ and $n_i(1) = 0$
    $\implies$ oscillation between $\underline{\mathbf{0}}$ and $\underline{\mathbf{1}}$.

The case for a CA at quiescent state $\underline{\mathbf{1}}$ is exactly symmetrical. □

Another useful measure is volatility, which measures the average number of cells that change per time step. We expect class 1 CA to have a volatility that approaches zero. Class 2 and 3 CA will have a volatility that approaches a non-zero constant. Class 4 CA can have a volatility that does not converge at all.

Definition 7.3 (Volatility)**.** *The volatility v of a CA between time steps t and $t+\delta$ is*

$$v = \frac{1}{\delta N^2} \sum_{i=0}^{\delta-1} |X_{t+i} \oplus X_{t+i+1}| \tag{7.1}$$

A good way to separate class 1 and class 2 CA is to observe their density at t time steps for some relatively large t. Density is the average number of live cells in the CA state. If the CA have converged to their respective cycles, we will find that class 1 CA have density of 0 or 1 for almost all initial conditions whereas class 2 CA will regularly have densities that are in between 0 and 1.

Definition 7.4 (Density)**.** *The density d of a CA at time step t is*

$$d = \frac{1}{N^2} |X_t| \tag{7.2}$$

7.1.2 Evolutionary Algorithms

It is useful to evaluate the manner in which a population converges towards solutions during evolution. Although we reuse the word convergence to describe this, note the difference between a CA converging to a periodic solution and a population converging towards an optimal transition function. An EA converges when its elite individuals are relatively homogenous. To evaluate the convergence of a population, we would like to have a notion of similarity between individuals. This allows us to quantify the speed of convergence and group individuals into families if they are converging towards different optima. For a binary string chromosome, the simple matching coefficient (SMC) between two individuals is an appropriate metric to quantify their relative similarity.

Definition 7.5 (Simple Matching Coefficient)**.** *Consider two binary strings A and B. The frequency table enumerating the number of instances of each possible combination of bit settings is*

	$A_i = 0$	$A_i = 1$
$B_i = 0$	$n_{0,0}$	$n_{1,0}$
$B_i = 1$	$n_{0,1}$	$n_{1,1}$

The simple matching coefficient between two binary strings A and B of length n is

$$SMC = \frac{n_{0,0} + n_{1,1}}{n_{0,0} + n_{1,1} + n_{0,1} + n_{1,0}} \tag{7.3}$$

Symmetrically, we consider the Simple Matching Distance, SMD $= 1 -$ SMC, as a measure of diversity between individuals. This is appropriate as SMD fulfills all the formal criteria for a distance metric: non-negativity, symmetry, identity of indiscernibles, and the triangle inequality. If each bit is to be imagined as a gene, the SMD is the mean number of differing genes in the chromosome. It can be calculated in $O(n)$ time where n is the chromosome length. Aside from being simple to understand and efficient to calculate, the SMD is preferable to other metrics as it accounts for mutual presence and mutual absence. Jaccard Distance d_J, on the other hand, only registers mutual presence.

Definition 7.6 (Jaccard Distance)**.** *For two binary strings A and B, the Jaccard distance between them is:*

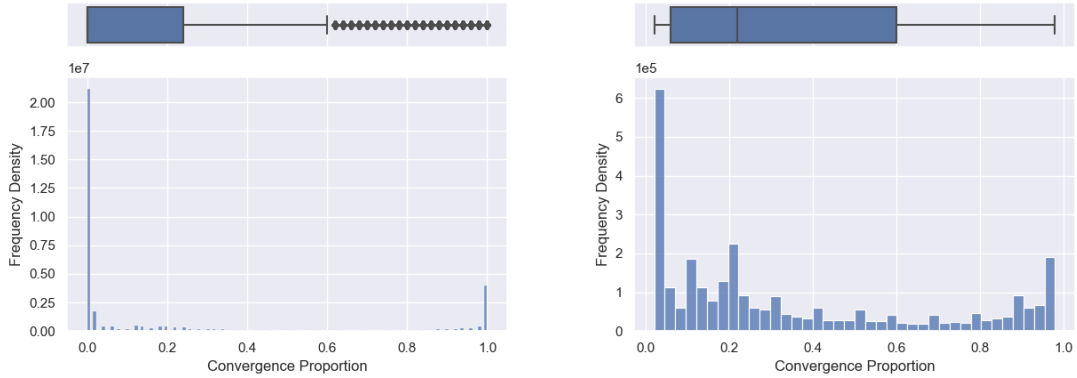
$$d_J = \frac{n_{0,1} + n_{1,0}}{n_{0,1} + n_{1,0} + n_{1,1}} \tag{7.4}$$

This is useful in settings where true negatives ought to be ignored. For example, when testing if two water samples came from the same source, we would not be interested in enumerating all the compounds that are *not* present in both samples. However, our use case benefits from knowing when a gene is *not present* in both chromosomes as much as knowing if it *is present* since both of these properties can equally affect CA dynamics. For this reason we use the Simple Matching Distance.

7.2 Exploration

When evaluating the effectiveness of evolutionary techniques at learning life-like CA rules, it is important to contextualise quantitative properties like convergence rate, fitness, and diversity. We perform an exploratory statistical analysis to gather data on the life-like CA rule space. In conjunction with the analyses of Wolfram[41] and Eppstein[14], this will shed light into the characteristics that make a rule easier or harder to predict.

There are $2^{18} = 262144$ possible outer-totalistic cellular automata rules which makes a systematic analysis of their properties feasible through random simulation. 100 initial conditions are sampled from a distribution uniform across densities 0 to 1. Each rule is simulated on each initial condition for 100 time steps and the state at each step is recorded in a hashmap. We use this data to find the percentage of initial conditions that converge within 100 steps for each rule. This is shown in Figure 7.1.



(a) Full dataset (262144 rules)

(b) Reduced dataset (74425 rules)

Figure 7.1: Distributions of convergence of full and reduced set of life-like CA

First we consider the two extremities. 11.4% of rules converge for all initial conditions and 60.3% converge for no initial conditions. This leaves 28.3% or 74425 of the original rules remaining. Although most rules in this reduced dataset have a relatively low convergence proportion, there is no clear decision boundary visible to separate trivial and periodic CA (class 1 and 2) from chaotic and complex CA (class 3 and 4).

For the 39.7% of rules that converge for at least one initial condition, we consider 4 key properties. The mean convergence time (i.e. how many time steps did it take before a periodic cycle was entered), the mean period, the mean density of the state after 100 time steps, and the mean volatility throughout 100 time steps. Figure 7.2 shows the distributions and relationships between these properties. We separate rules based on whether they converged for *all* initial conditions or only *some* initial conditions.

We see that the mean convergence time is positively skewed and mean period is very positively skewed. This is especially true of rules that converge for all initial conditions. Such rules also tend to converge to very high density or very low density states. These can be thought of as "explosive" or "implosive" rules respectively. Explosive rules are likely to have many elements in their birth set and few elements in their death set making them conducive to rapid growth into a high density state. The opposite is true for implosive rules. Rules at these extremities are likely to be class 1 whereas the rules in between are likely to be class 2. For rules that converge for some initial conditions, there are still indications of explosive/implosive behaviour in the form of two spikes on either side of the central KDE plot but we see that there is clear bias towards rules with a density around 0.5. This is what we would expect if we were randomly generating states. Rules that converge for all states tend to have very low volatility indicating they converge out of slowly developing patterns rather than chaos.

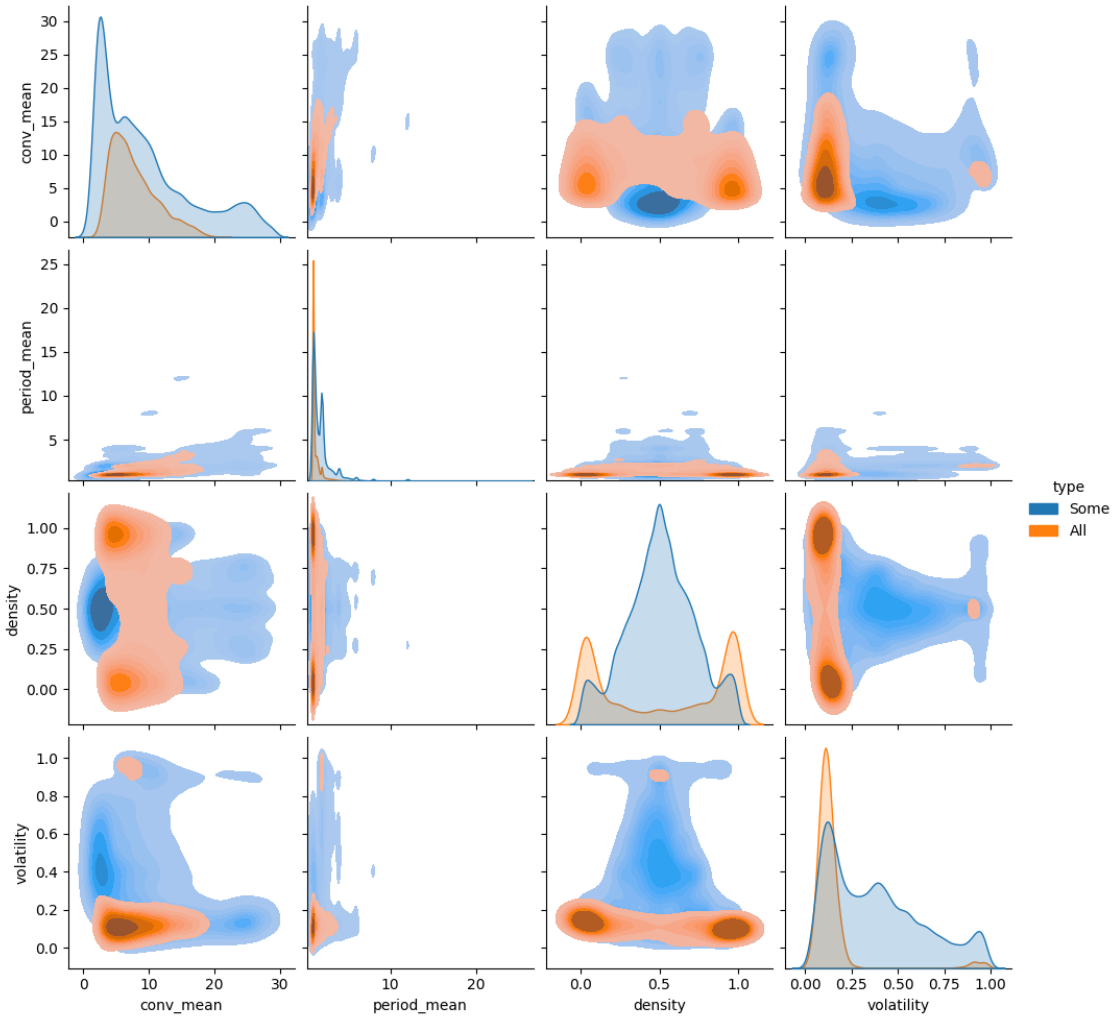


Figure 7.2: KDE plots showing distributions between mean convergence time, mean period, mean density of final state, and mean volatility for all CA that converged to a periodic cycle for at least one initial condition

7.3 Maze Generation

We evaluate the effectiveness of the procedural maze generator at maximizing the two fitness metrics. These are the length of the solution path (p) and the number of dead ends (d). From spot checks on a few examples in the simulator, we find that these two metrics happen to have similar ranges (around 70-100). For this reason we begin by treating them with equal weighting during hyperparameter tuning.

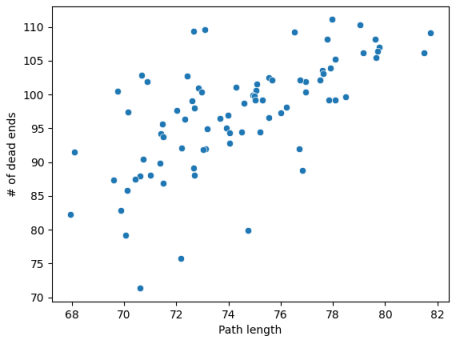
7.3.1 Hyperparameter Tuning

We perform a grid search on training hyperparameters across the following ranges

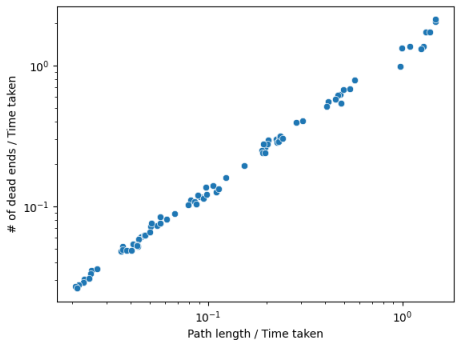
Hyperparameter	**Values Tried**
Number of Epochs	10, 50, 100
Population Size	20, 50, 100
Elitism Rate	0.1, 0.2, 0.5
Mutation Rate	0.01, 0.05, 0.1

The results of this are shown in Figure 7.3. The optimum metrics achieved are ($d = 109.1$, $p = 81.8$) with configurations (number of epochs = 100, population size = 20, elitism rate = 0.5, mutation rate = 0.01). Under a time adjusted measure, we try to optimize for the value of the

metric achieved relative to the execution time. Here, the optimum value is ($d = 100.5$, $p = 69.8$) for configurations (number of epochs = 100, population size = 20, elitism rate = 0.1, mutation rate = 0.05). This is not significantly worse than the optimum value achieved when disregarding execution time.



(a) Fitness across both metrics, d against p.

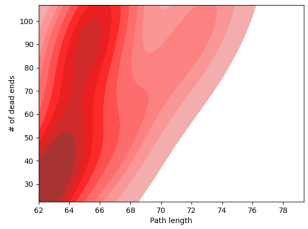


(b) Runtime adjusted fitness. $\frac{d}{t}$ against $\frac{p}{t}$ with log axes.

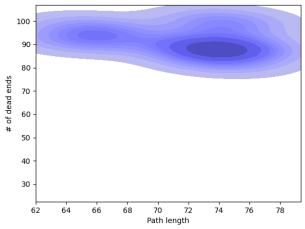
Figure 7.3: Fitness of different hyperparameter configurations across both metrics for maze generation

7.3.2 Ablation Analysis

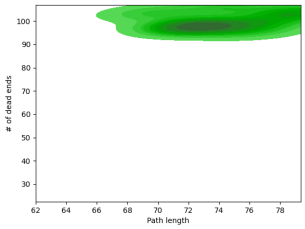
Hyperparameter tuning revealed that the lowest mutation rate produced the best results. This suggests that mutation may not be necessary for maze evolution. To test this hypothesis, we conduct an ablation analysis on the genetic operators used. We run 10 maze generations, each for 100 epochs, using the optimal runtime-adjusted hyperparameters for efficiency. We repeat this procedure, each time systematically adding and removing genetic operators from the learning procedure. Figure 7.4 shows KDE plots which visualise the area occupied by the optimal results from each of these learning procedures.



(a) Mutation and no crossover



(b) Crossover and no mutation



(c) Both mutation and crossover

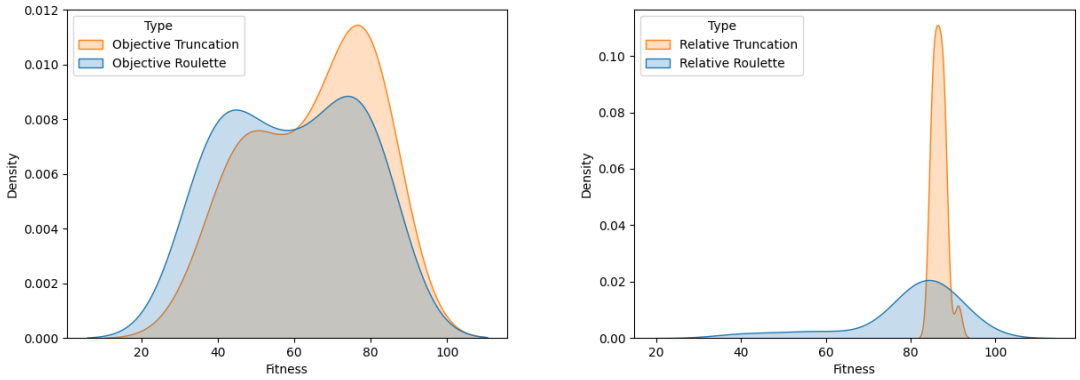
Figure 7.4: Optimum results produced when removing mutation and crossover components for mazes

It is clear that crossover is important for convergence to a strong solution as the experiment without crossover produces highly variable results. Mutation appears to have a much smaller effect. Without mutation, we obtain a slightly higher variance in results produced and solutions tend to have a lower number of dead ends. If both mutation and crossover are omitted, no learning occurs and, in the majority of cases, no legitimate mazes are found.

7.3.3 Objective vs Relative Fitness

Since we are optimising for two metrics, we can take a linear combination of the objective values of p and d or we can rank each candidate separately by p and d then take a linear combination

of the ranks. We call this objective and relative fitness respectively. For each fitness type, we run 20 experiments with a bias of 0.5 (i.e. equal weighting to the two goals) In Figure 7.5, KDE plots show the performance of roulette and truncation selection using objective and relative fitness for selection. Note that the x-axis for both plots is still objective fitness as that is what the algorithm is attempting to maximise overall.



(a) Objective Fitness

(b) Relative Fitness

Figure 7.5: Objective and relative fitness for roulette and truncation selection

Under both objective and relative fitness, truncation selection outperforms roulette selection on average. There is also a lower variance in the fitness of solutions produced using truncation selection. This is especially clear in the case with relative fitness. One reason for this is the decreasing variability in fitness of elite candidates as the population evolves. This makes a roulette selection increasingly likely to pick suboptimal parents. Truncation selection, on the other hand, asserts that a selected candidate will never have a lower fitness than a candidate that has not been selected. However, this can also increase the likelihood of premature convergence since pairs or groups of genes that have the potential to produce global optima when together are evicted in favour of individual genes that produce suboptimal solutions independently. Based on the results shown, we opt for a truncation selection using relative fitness. It is interesting to see that an alternative selection criterion, relative fitness, can optimize the objective function better than using the objective function itself for selection.

7.3.4 Bias Tuning

The bias parameter λ decides the weighting given to solution path length over number of dead ends when calculating fitness. Although this can be set by the user based on personal preference, we can tune the bias to find the value that maximizes overall fitness $f(\lambda) = \lambda\hat{p}_\lambda + (1-\lambda)\hat{d}_\lambda$ where $\hat{p}_\lambda$ and $\hat{d}_\lambda$ are the optimal values discovered by the algorithm under a bias of λ.

Figure 7.6 shows that the algorithm performs best for higher bias values except $\lambda = 0.0$ which has a very high performance. For a maze with a mix of dead ends and long solution path, the user may find better results from a bias of around 0.5 than around 0.3. Figure 7.7 shows the top mazes produced for different bias values. Qualitatively these appear to optimize the metrics they are being tested against.

7.4 Life-Like CA

7.4.1 Single Case Evaluation

We begin by learning a few specific life-like CA rules to test the efficacy of the learning process. Consider Conway's Life rule B3/S23. In binary this is '000100000001100000' and as an integer it is 16480. Based on spot checks, we use the following hyperparameters

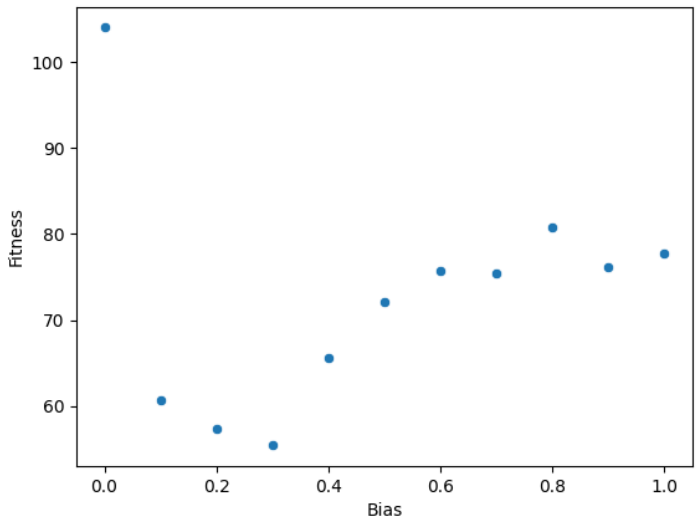
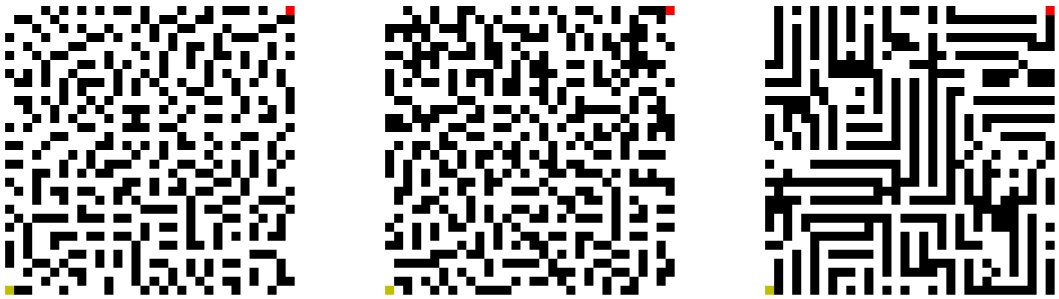


Figure 7.6: Best fitness rule strings discovered under different bias values λ



(a) B16/S1235 ($\lambda = 0.0$) (b) B2678/S12348 ($\lambda = 0.4$) (c) B238/S12356 ($\lambda = 1.0$)

Figure 7.7: Best mazes produced from rules discovered using different bias values

Number of Epochs	30
Population Size	20
Number of Initial Conditions	20
Elitism Rate	0.2
Mutation Rate	0.05
Evaluation Steps	10
Minimum Step Size	1
Maximum Step Size	5

These have been obtained by looking at common parameters used in the literature, using parameters from the maze generation experiments, and running spot checks on a few examples. A more rigorous hyperparameter tuning of the final 3 parameters is covered in Subsection 7.4.3. We test on initial conditions sampled uniformly on density.

Figure 7.8 shows the algorithm to be very successful at learning the Life rule. In this experiment, the algorithm converges to the correct solution in 5 epochs. At many epochs, and especially after the true rule has been discovered, we see less than 4 points because a few of the elite rules are the same. Note that the graph has a log y-axis, so convergence is extremely fast.

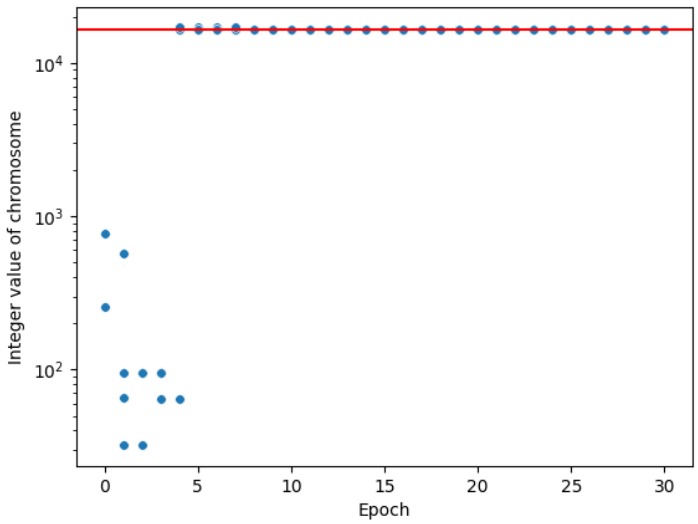


Figure 7.8: Top 4 elite rules from each epoch in an experiment learning Life. The red line represents the goal.

Figure 7.9 shows the portion of the search space visited by the algorithm. The top areas far from the goal quickly fade away as selection pressure removes such solutions and offspring in this area are no longer produced. We can also see some dark bands emerging far from the goal at epoch 30. However, the evolution history shows that by epoch 8, all the elite candidates are identical copies of the goal. This means that the dark bands seen at the end of the graph are not local optima but rather instances of the limited types of offspring that can be produced by applying crossover between two identical rule strings representing the goal solution.

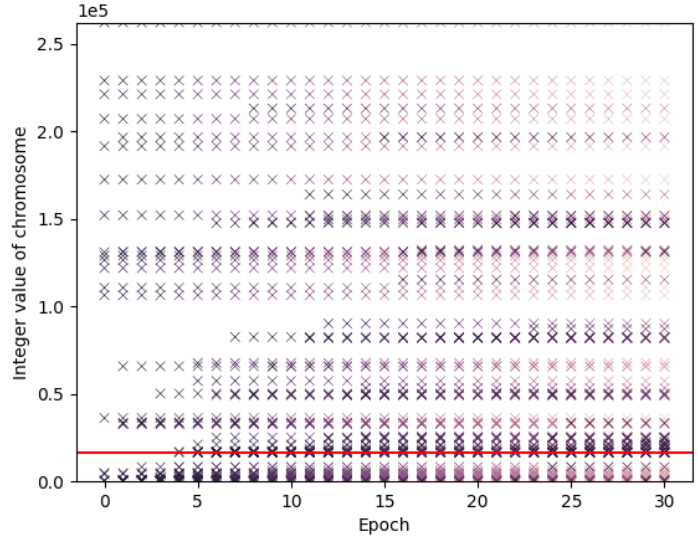
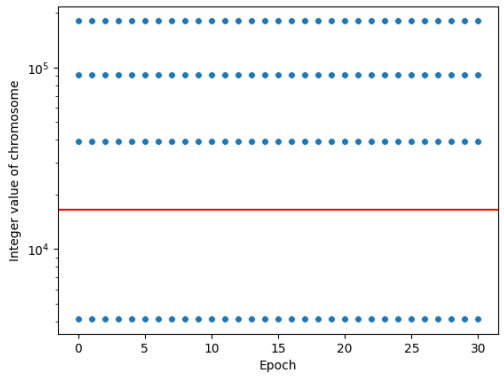


Figure 7.9: All visited rules during evolution. Each rule is represented as a dark cross when first discovered and gradually fades in colour from left-to-right. The red line represents the goal.

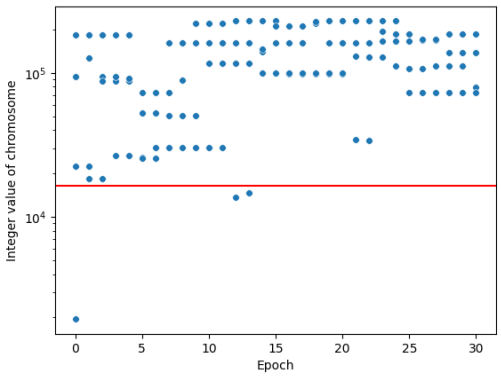
7.4.2 Ablation Analysis

Ablation analysis reveals that both mutation and crossover are critical to achieving a performance like this. Consider Figure 7.10. 'Visited' indicates the number of unique candidates considered by the algorithm in 30 epochs. Under no mutation and no crossover, we see that the population

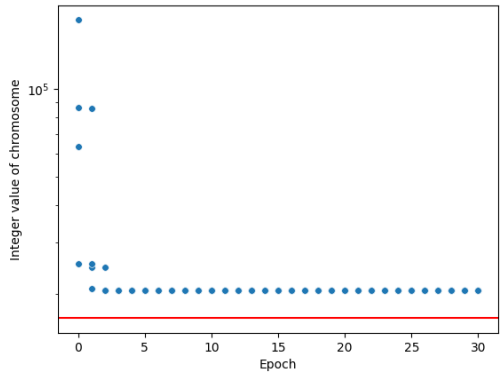
does not evolve at all. The number of visited candidates is equal to the initial population size, 20. With mutation but no crossover, the algorithm is incapable of convergence as it is unable to retain generational knowledge. With crossover but no mutation, the algorithm quickly converges to a local minimum from which it is powerless to escape. This also stagnates the number of candidates visited to 37. When both mutation and crossover are maintained, the algorithm successfully converges to the global minimum.



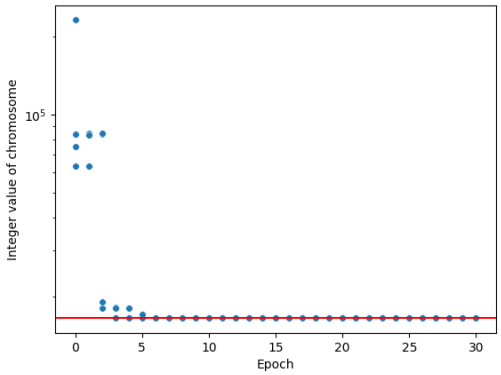
(a) No mutation and no crossover (visited = 20)



(b) Mutation and no crossover (visited = 90)



(c) Crossover and no mutation (visited = 37)



(d) Both mutation and crossover (visited = 197)

Figure 7.10: Optimum results produced when removing mutation and crossover components for modelling life-like CA

7.4.3 Stepsize Tuning

Since the rule space is of order 10^6, performing hyperparameter tuning across all parameters on a representative sample of the search space would be infeasible in the given time frame with the resources available. We find the parameters established in 7.4.1 to be appropriate based on spot checks on a set of rules covering all 4 Wolfram classes. These bear some similarity to the hyperparameters established in the procedural generation case. One point of interest is optimising the number of evaluations and the stepsize since these factors have considerable effect on the training time.

Considering a random uniform stepsize, $\delta_k \sim Uniform(D_{max}, D_{min})$ we set $D_{min} = 1$ since we would like to give the algorithm a chance of learning on very small steps. To determine D_{max}, the algorithm is run on 100 goal rules sampled uniformly across the integer rule values and the loss of each rule is calculated by simulating on 20 random initial conditions. Note that by initialising the population uniformly on rule densities but sampling the goals uniformly on integer values, we ensure that we are not gaining an unfair advantage by testing on a subset of the search space where

our algorithm is more likely to place initial guesses. The results are shown in Table 7.1.

D_{max}	Number of Steps		
	1	5	10
1	100%	76%	52%
5	100%	100%	100%
10	100%	100%	100%

(a) Average convergence rate

D_{max}	Number of Steps		
	1	5	10
1	9	12	11
5	9	9	10
10	9	9	9

(b) Average convergence time (epochs)

Table 7.1: Results of learning life-like CA.

Almost all configurations of step number and step size were able to converge to 100% of the target rules within 30 epochs. The two configurations that failed to do this involved a maximum step size of 1 and many evaluation steps. In the extreme case of the configuration with 10 consecutive steps, only 52% of targets were learnt successfully. Here, the CA is in the early stages of simulation with many transient patterns, so additional observations appears to be counter-productive. However, we also see that the genetic algorithm can perform surprisingly well calculating fitness on a single immediate observation of the goal rule after 1 time step. Computationally, this is very efficient compared to taking multiple stochastically chosen observations over longer periods of time.

There is some variability in the average convergence times obtained when repeating this experiment so we only list average convergence time to the nearest epoch. Almost all configurations seem equally good in this regard with the exception of configurations with very small step sizes or very high number of observations. In general, many late-stage observations can dilute information gathered in the early-stages of observation. With the top configurations tested, these algorithms are able to find the correct solution after considering less than 130 candidates. This is impressive given that this "visited region" makes up less than 0.05% of the entire search space.

However, it is also important to note the size of the test set. The set of 100 targets, despite being representative of the all possible rule strings, also makes up less than 0.05% of the entire search space. It is plausible to imagine that there is a section of the search space in which our algorithm performs badly but has not been represented in the test set. With additional time and computational resources, it would be sensible to run this experiment again with a larger test set to see if any rules emerge that cannot be learnt by the algorithm in 30 epochs.

7.4.4 Fitness Functions

We now perform the same hyperparameter grid search on the second fitness function designed: multi-resolution loss. This function takes 3 differently sized smoothing convolutions of the true and surrogate CA, XORs them with each other to get the difference in density in a region around each cell, and takes the average of these differences. The reasoning behind this method is that we expect candidates with similar genotypes to have similar phenotypic behaviour. Therefore, we would expect similar rules to create patterns of similar density in different regions of the lattice. A convolution leverages this effect since the loss between two states with a slight phase shift would be much higher using single-resolution loss than using multi-resolution loss. The results are shown in Table 7.2.

D_{max}	Number of Steps		
	1	5	10
1	96%	63%	52%
5	96%	97%	98%
10	96%	98%	98%

(a) Average convergence rate

D_{max}	Number of Steps		
	1	5	10
1	11	12	15
5	12	12	12
10	10	11	12

(b) Average convergence time (epochs)

Table 7.2: Results of learning life-like CA with multi-resolution loss.

We find that multi-resolution loss is patently worse than single resolution loss. When learning with multi-resolution loss, all configurations are able to learn fewer targets and convergence time is higher. This suggests that the algorithm is not using high level, stable features to distinguish between different CA. Instead, it is using high-fidelity behaviour in the early chaotic stages of each CA to uniquely identify them. This also explains why performance decreases as the number of observations increase.

7.4.5 Class Evaluation

We would expect our algorithm to be far better at predicting class 1 and class 2 rules than class 3 and class 4 rules due to the difference in complexity. However, as shown in Figure 7.11, there is no clear correlation between the convergence percentage of a rule in the exploration experiment and the number of candidates visited by the GA when learning that rule.

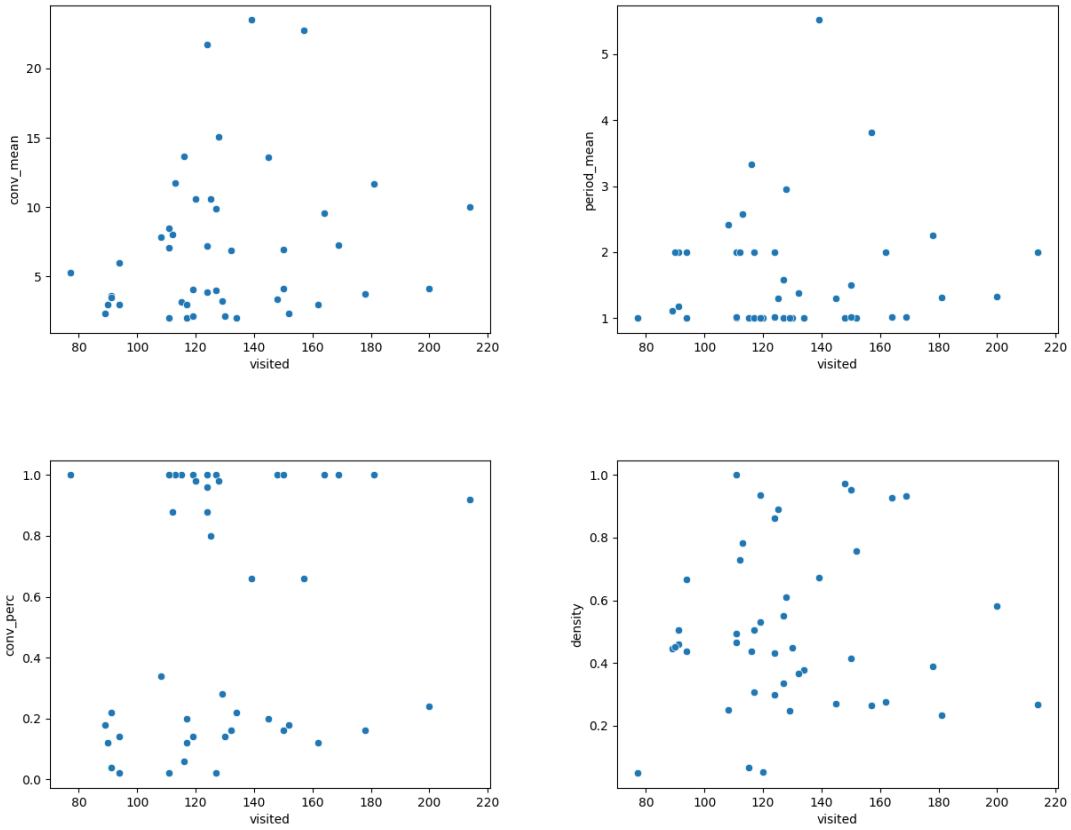


Figure 7.11: Exploration metrics of CA on random initial conditions against number of rules visited by GA before discovering goal

In fact, the difficulty of learning, measured as the number of candidates visited before the correct solution was found, seems to be uncorrelated with any metric collected during the exploration phase. This suggests that the GA is capable of learning equally well on a wide variety of CA behaviour.

We can visualise populations converging to optimal solutions very easily using the simple matching coefficient (SMC). On every epoch, we measure the fitness of the population and the average SMC between individuals in the population. As Figure 7.12 shows, fitnesses increases and average SMC decreases in the population over multiple epochs. These metrics change in a symmetrically opposing way indicating that the population becomes fitter and more genotypically homogenous as time progresses.

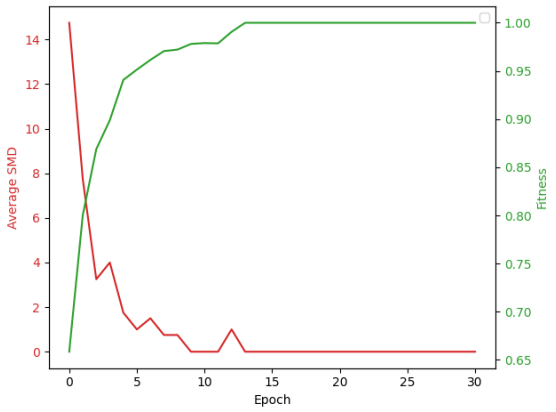


Figure 7.12: Change in fitness and diversity of the population over time. Learning B3/S23 with the standard hyperparameters outlined in Table 7.4.1.

7.5 Gray-Scott Models

The search space for Gray-Scott models is infinite so in the style of Giampaolo et al.[23], we consider four example targets shown in Table 7.3.

Target Name	Feed Rate (f)	Kill Rate (k)
Flower	0.055	0.062
Zebrafish	0.035	0.060
Soliton	0.030	0.060
Mitosis	0.028	0.062

Table 7.3: Example targets for Gray-Scott systems.

We choose to train a population of 25 candidates for 30 epochs since similar values worked effectively for life-like CA. Limited computational resources also mean that it is infeasible to run the range of experiments required on larger populations for longer periods. We use a linear truncation loss, splatter initialisation for each CA, and random uniform initialisation of the parameters across $f \in [0, 0.3]$ and $k \in [0, 0.08]$.

7.5.1 Evolutionary Strategy and Genetic Algorithm III

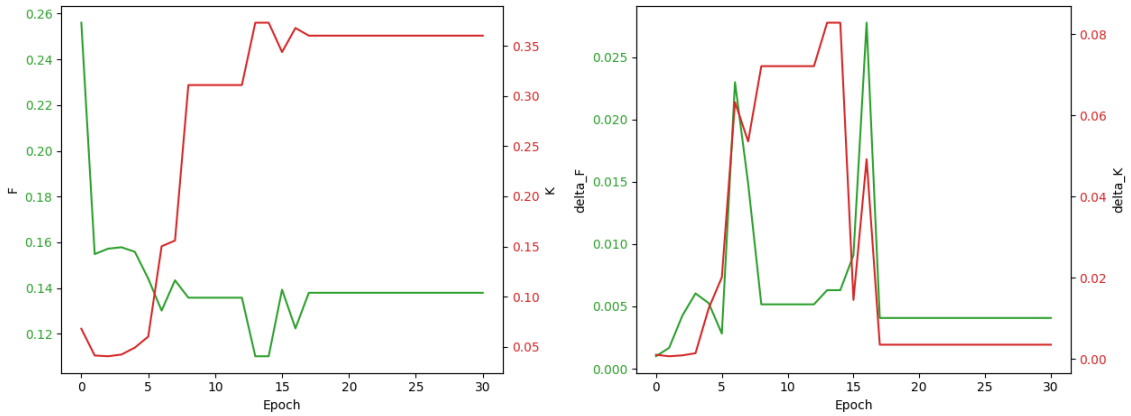
Using a self-adapting evolutionary strategy, we find that the patterns converge to local optima. Consider the flower pattern. Figure 7.13 shows both the state and control parameters converging at around 17 epochs. However, the converged solution of ($f = 0.138, k = 0.036$) is not the true solution.

Increasing the population size does not appear to resolve matters. Changing the discrete Laplacian operator to the alternative presented by Compeau[57],

$$K = \begin{bmatrix} 0.05 & 0.2 & 0.05 \\ 0.2 & -1 & 0.2 \\ 0.05 & 0.2 & 0.05 \end{bmatrix}$$

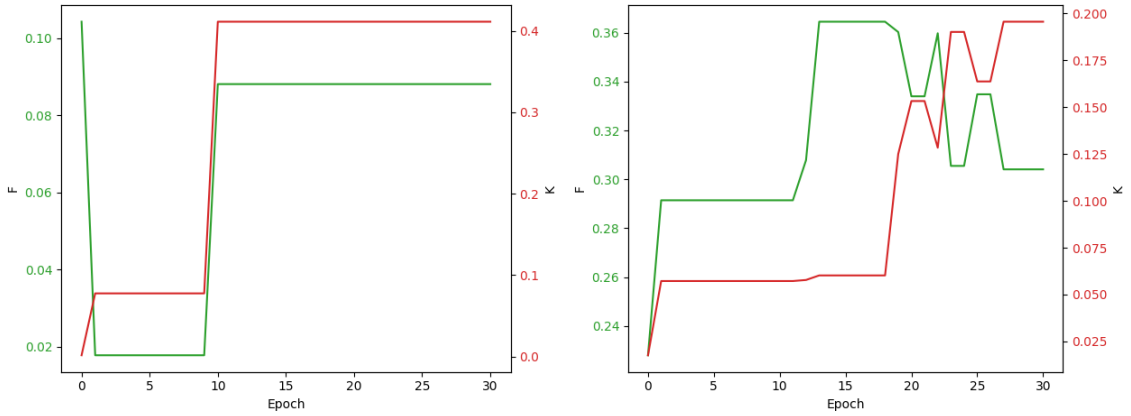
also appears to have no effect. These are shown in Figure 7.14.

The case for genetic algorithms is very similar. We graph the local optima found for each algorithm, chromosome initialisation method, initial condition and recombination operator in Figure 7.15. It is clear that there are around many distinct regions of local optimality and, by varying the hyperparameters, we end up in different regions.



(a) State parameters (b) Control parameters

Figure 7.13: Evolution of *flower* pattern using evolutionary strategy (population size = 20)

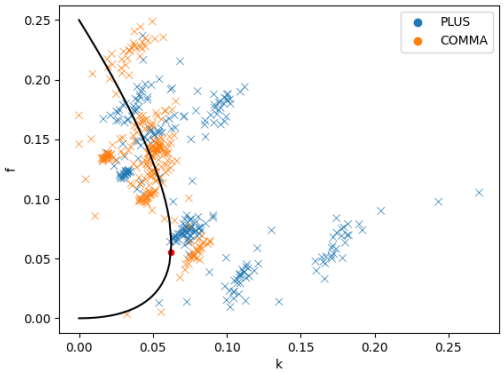


(a) State parameters for *zebrafish* pattern (pop size = 50, Laplacian = nine-point stencil)

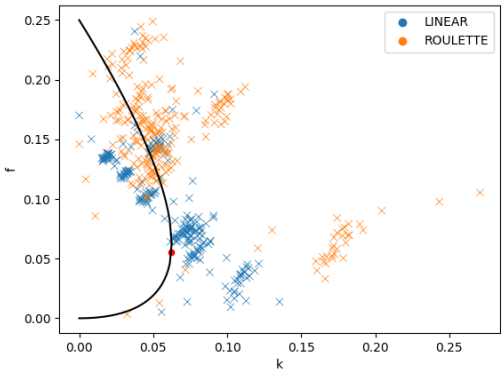
(b) State parameters for *soliton* pattern (population size = 25, Laplacian = Compeau operator)

Figure 7.14: Evolution of Gray-Scott models with different population sizes and Laplacian operators

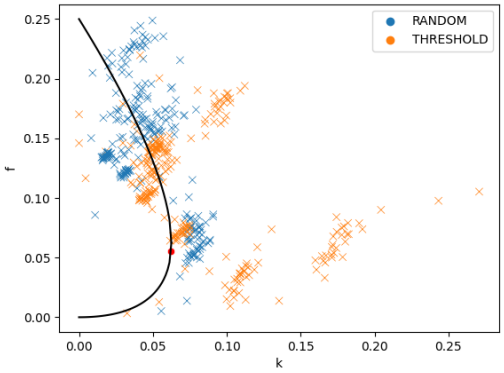
Figure 7.16 shows a simulation of the target pattern (flower) and Figure 7.17 shows a simulation of a local optimum that was discovered.



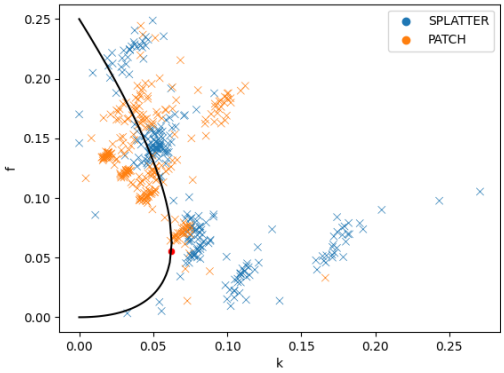
(a) Recombination



(b) Selection



(c) Initialisation



(d) Seed

Figure 7.15: Genetic algorithm on Gray Scott model with different genetic operators. The red dot is the target and the black line is the bifurcation line.

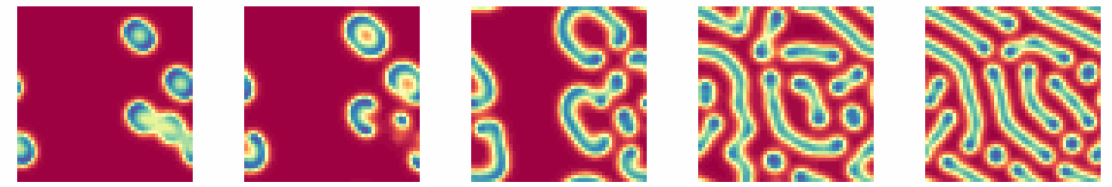


Figure 7.16: Simulation of Gray-Scott target

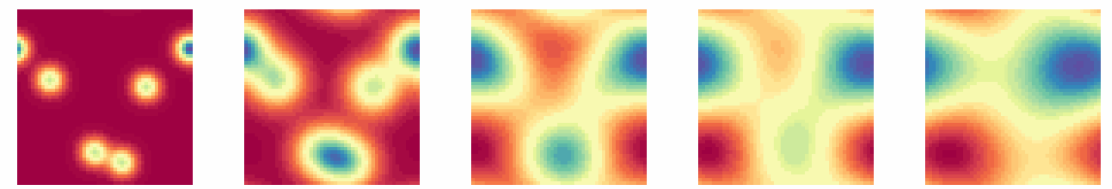


Figure 7.17: Simulation of Gray-Scott local optimum

Chapter 8

Conclusions

This project has shown that evolutionary algorithms can effectively tackle inverse problems for cellular automata. At the outset, we introduced three objectives. We address these in turn.

1. **State optimization for life-like CA**
   We built genetic algorithms to learn life-like CA transition functions that, with minor post-processing, randomly generate mazes with desirable qualities: long solution paths and many dead ends. This pushes the boundaries on existing work using CA for maze generation[6, 55] through a stochastic region merging algorithm that is unbiased with respect to orientation and guarantees playability.

2. **Full rule dynamics for life-like CA**
   We built genetic algorithms that successfully learned the full rule dynamics of all 100 life-like CA, sampled uniformly across the search space, based on a very small number of observations (in some cases, a single observation). This was a very quick learning process requiring an average of 9 epochs of training on a population of 20 individuals. This represents a traversal of only 0.047% of the total search space. As the first work to learn full rule dynamics for a class of 2D CA, we establish an important result in this regard.

3. **Full rule dynamics for Gray-Scott models**
   We further extended these genetic algorithms and implemented an evolutionary strategy method for continuous state CA. When testing on a few examples, these algorithms converge to locally optimal regions which we visualise in the parameter space. This presents an area with great potential for future work.

During this project, an efficient simulation system was built for both life-like and Gray-Scott CA which produces rich media in the form of images and animations. Together with implementations of numerous genetic operators, selection methods, chromosome structures, evaluation metrics, and experiment templates, this forms a cohesive evolutionary algorithm toolkit that can be extended to accommodate other classes of discrete and continuous CA.

8.1 Further Work

This project reveals many potential avenues for future exploration. We briefly discuss some of these here.

8.1.1 Procedural Generation

The use of CA for procedural generation extends to many domains outside of graphics and games. One particular area of interest is evolutionary algorithms to train CA models for efficient transport network design. This would require some notion of orientation to be embedded within the CA chromosome, thereby making outer-totalistic CA transition functions unsuitable. A better approach may be to consider something similar to the perception vector used by Mordvintsev et al.[19] in their work on neural cellular automata. Here, alongside the number of neighbours, each chromosome would leverage information about the gradient in the x and y directions. Sobel filter convolutions could be used to capture such information.

8.1.2 New CA Topologies

In this work, we focus on CA that exist on square lattices with periodic boundary conditions. This *toroidal* topology simulates an infinite periodic tiling. It would be useful to consider whether the results established in this work extend to other topologies. A simple change would be to consider fixed boundary conditions. A more drastic change would be to consider a lattice of non-uniform cells such as those obtained by performing a Voronoi decomposition. It has been shown neural cellular automata can effectively learn full rule dynamics for Life on Voronoi lattices[52]. It would be interesting to see if these results can also be achieved with evolutionary computation.

8.1.3 Evolutionary Strategies

When learning on Gray-Scott models, we used evolutionary strategies with modern features such as self-adaption. However, there are many other techniques that are used in current state-of-the-art ES. For example, covariance matrix adaptation evolutionary strategies (CMA-ES) make fewer assumptions about the underlying function and may yield more promising results. With the limited experimentation time available in this project, we opted to test improved genetic algorithms over modern ES. However, experimenting with improved ES would be a natural next step.

8.1.4 Neural Networks

Neural cellular automata are at the cutting edge of current research around self-organising systems[19, 52]. Such systems are very successful at complex morphogenesis goals. These systems typically use a CA where the transition function is itself a neural network. Such techniques may also perform well in domains where evolutionary algorithms appear to fall short, like learning Gray-Scott models.

Bibliography

[1] Dieter A Wolf-Gladrow. *Lattice-gas cellular automata and lattice Boltzmann models: an introduction*. Springer, 2004.

[2] David Reher, Barbara Klink, Andreas Deutsch, and Anja Voss-Böhme. Cell adhesion heterogeneity reinforces tumour cell dissemination: novel insights from a mathematical model. *Biology direct*, 12(1):1–17, 2017.

[3] Roger White and Guy Engelen. High-resolution integrated modelling of the spatial dynamics of urban and regional systems. *Computers, environment and urban systems*, 24(5):383–400, 2000.

[4] N Wulff and J A Hertz. Learning cellular automaton dynamics with neural networks. *Advances in Neural Information Processing Systems*, 5:631–638, 1992.

[5] P Gray and SK Scott. Autocatalytic reactions in the isothermal, continuous stirred tank reactor: isolas and other forms of multistability. *Chemical Engineering Science*, 38(1):29–43, 1983.

[6] Chad Adams and Sushil Louis. Procedural maze level generation with evolutionary cellular automata. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2017.

[7] Dean Hickerson. Dean hickerson's oscillator stamp collection. URL https://conwaylife.com/ref/DRH/stamps.html.

[8] Debasis Das. A survey on cellular automata and its applications. volume 269, 12 2011. ISBN 978-3-642-29218-7. doi: 10.1007/978-3-642-29219-4_84.

[9] Alan Dorin, Jonathan McCabe, Jon McCormack, Gordon Monro, and Mitchell Whitelaw. A framework for understanding generative art. *Digital Creativity*, 23, 12 2012. doi: 10.1080/14626268.2012.709940.

[10] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 2002. ISBN 1579550088. URL https://www.wolframscience.com.

[11] Hans Meinhardt. *The algorithmic beauty of sea shells*. Springer Science & Business Media, 2009.

[12] Karl Sims. Reaction-diffusion tutorial. URL http://www.karlsims.com/rd.html.

[13] Andrew Adamatzky, Genaro Juárez Martínez, and Juan Carlos Seck Tuoh Mora. Phenomenology of reaction–diffusion binary-state cellular automata. *International Journal of Bifurcation and Chaos*, 16(10):2985–3005, 2006.

[14] David Eppstein. Growth and decay in life-like cellular automata. In *Game of Life cellular automata*, pages 71–97. Springer, 2010.

[15] Thomas P Meyer, Fred C Richards, and Norman H Packard. Learning algorithm for modeling complex spatial dynamics. *Physical review letters*, 63(16):1735, 1989.

[16] Melanie Mitchell, James P Crutchfield, Rajarshi Das, et al. Evolving cellular automata with genetic algorithms: A review of recent work. In *Proceedings of the First international conference on evolutionary computation and its applications (EvCA'96)*, volume 8. Moscow, 1996.

[17] David Andre, Forrest H Bennett III, and John R Koza. Discovery by genetic programming of a cellular automata rule that is better than any known rule for the majority classification problem. *Genetic programming*, 96:3–11, 1996.

[18] Ron Breukelaar and Thomas Bäck. Evolving transition rules for multi dimensional cellular automata. In *International Conference on Cellular Automata*, pages 182–191. Springer, 2004.

[19] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 2020. doi: 10.23915/distill.00023. https://distill.pub/2020/growing-ca.

[20] John E Pearson. Complex patterns in a simple system. *Science*, 261(5118):189–192, 1993.

[21] Robert Munafo. Pearson's classification (extended) of gray-scott system parameter values, . URL https://mrob.com/pub/comp/xmorphia/pearson-classes.html.

[22] Robert Munafo. Reaction-diffusion by the gray-scott model: Pearson's parametrization, . URL https://mrob.com/pub/comp/xmorphia/index.html.

[23] Fabio Giampaolo, Mariapia De Rosa, Pian Qi, Stefano Izzo, and Salvatore Cuomo. Physics-informed neural networks approach for 1d and 2d gray-scott systems. *Advanced Modeling and Simulation in Engineering Sciences*, 9(1):1–17, 2022.

[24] VIGNERON. Structure de jeu de la vie (automate cellulaire), 2006. URL https://commons.wikimedia.org/wiki/File:JdlV_osc_5.56.gif.

[25] Mohammad Ali Abido. Multiobjective evolutionary algorithms for electric power dispatch problem. *IEEE transactions on evolutionary computation*, 10(3):315–329, 2006.

[26] Andreas Deutsch, Josué Manik Nava-Sedeño, Simon Syga, and Haralampos Hatzikirou. Biolgca: A cellular automaton modelling class for analysing collective cell migration. *PLoS Computational Biology*, 17(6):e1009066, 2021.

[27] Predrag T Tosic. Cellular automata for distributed computing: models of agent interaction and their implications. In *2005 IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 3204–3209. IEEE, 2005.

[28] Norman H Packard and Stephen Wolfram. Two-dimensional cellular automata. *Journal of Statistical physics*, 38(5):901–946, 1985.

[29] L Hernández Encinas, S Hoya White, A Martín Del Rey, and G Rodríguez Sánchez. Modelling forest fire spread using hexagonal cellular automata. *Applied mathematical modelling*, 31(6): 1213–1227, 2007.

[30] Adam P Goucher. Gliders in cellular automata on penrose tilings. *Journal of Cellular Automata*, 7, 2012.

[31] Wenzhong Shi and Matthew Yick Cheung Pang. Development of voronoi-based cellular automata-an integrated dynamic model for geographical information systems. *International Journal of Geographical Information Science*, 14(5):455–474, 2000.

[32] Marco Bartolozzi and Anthony William Thomas. Stochastic cellular automata model for stock market dynamics. *Physical review E*, 69(4):046112, 2004.

[33] Armin R Mikler, Sangeeta Venkatachalam, and Kaja Abbas. Modeling infectious diseases using global stochastic cellular automata. *Journal of Biological Systems*, 13(04):421–439, 2005.

[34] Martin Gardner. The fantastic combinations of jhon conway's new solitaire game'life. *Sc. Am.*, 223:20–123, 1970.

[35] Paul Rendell. Universal turing machine (utm) implemented in conway's game of life. URL http://rendell-attic.org/gol/utm/index.htm.

[36] Achim Flammenkamp. Top 100 of game-of-life ash objects. URL http://wwwhomes.uni-bielefeld.de/achim/freq_top_life.html.

[37] Matthew Cook et al. Universality in elementary cellular automata. *Complex systems*, 15(1): 1–40, 2004.

[38] Alan Mathison Turing. The chemical basis of morphogenesis. *Bulletin of mathematical biology*, 52(1):153–197, 1990.

[39] Julyan HE Cartwright. Labyrinthine turing pattern formation in the cerebral cortex. *Journal of theoretical biology*, 217(1):97–103, 2002.

[40] Lee Smolin. Galactic disks as reaction-diffusion systems. *arXiv preprint astro-ph/9612033*, 1996.

[41] Stephen Wolfram. Theory and applications of cellular automata. *World Scientific*, 1986.

[42] Takeo Yaku. The constructibility of a configuration in a cellular automaton. *Journal of Computer and System Sciences*, 7(5):481–496, 1973.

[43] John T Baldwin and Saharon Shelah. On the classi ability of cellular automata. *TCS*, 1999.

[44] Ron Breukelaar and Th Bäck. Using a genetic algorithm to evolve behavior in multi dimensional cellular automata: emergence of behavior. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 107–114, 2005.

[45] Fred C Richards, Thomas P Meyer, and Norman H Packard. Extracting cellular automaton rules directly from experimental data. *Physica D: Nonlinear Phenomena*, 45(1-3):189–202, 1990.

[46] Edward F Moore. The firing squad synchronization problem. *Sequential machines, selected Papers*, pages 213–214, 1964.

[47] Péter Gács, Georgy L Kurdyumov, and Leonid Anatolevich Levin. One-dimensional uniform arrays that wash out finite islands. *Problemy Peredachi Informatsii*, 14(3):92–96, 1978.

[48] Rajarshi Das, James P Crutchfield, Melanie Mitchell, and James M Hanson. Evolving globally synchronized cellular automata. 1995.

[49] Melanie Mitchell, James P Crutchfield, and Peter T Hraber. Evolving cellular automata to perform computations: Mechanisms and impediments. *Physica D: Nonlinear Phenomena*, 75 (1-3):361–391, 1994.

[50] Ana Bušić, Nazim Fates, Jean Mairesse, and Irene Marcovici. Density classification on infinite lattices and trees. In *Latin American Symposium on Theoretical Informatics*, pages 109–120. Springer, 2012.

[51] Kevin N Gurney. Training nets of hardware realizable sigma-pi units. *Neural Networks*, 5(2): 289–303, 1992.

[52] Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Learning graph cellular automata. *Advances in Neural Information Processing Systems*, 34, 2021.

[53] Robert P Munafo. Stable localized moving patterns in the 2-d gray-scott model. *arXiv preprint arXiv:1501.01990*, 2014.

[54] Saad A Manaa and Joli Rasheed. Successive and finite difference method for gray scott model. *Science Journal of University of Zakho*, 1(2):862–873, 2013.

[55] Chad Adams. *Evolving Cellular Automata Rules for Maze Generation*. PhD thesis, University of Nevada, Reno, 2018.

[56] Nikolaus Hansen, Dirk V Arnold, and Anne Auger. Evolution strategies. In *Springer handbook of computational intelligence*, pages 871–898. Springer, 2015.

[57] Phillip Compeau. Biological modeling. URL https://biologicalmodeling.org/.

[58] J Barkley Rosser. Nine-point difference solutions for poisson's equation. *Computers & Mathematics with Applications*, 1(3-4):351–360, 1975.