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 15, 2022

Abstract

We use evolutionary algorithms to reverse-engineer cellular automata (CA). From observations of CA execution, we deduce transition functions for imitation learning and procedural generation. We tackle two classes of CA: Life-Like CA and Gray-Scott systems. The former is the family in which John Conway's infamous "Game of Life" CA belongs and the latter is a discretization of Turing's chemical model for morphogenesis.

Acknowledgements

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.[1]

1.1 Motivation

Predicting effects is easier than predicting causes. This is the crux of the Inverse Problem of Science. The causal opacity of time makes it easier to estimate observations from a parameterised model of the world than to deduce parameters from observations. Time eliminates information through the forces of selection and entropy. For instance, given knowledge of biological evolution, we may have strong hope of predicting where fossils of certain species lie but to build a rigorous taxomony based on fossils alone is insurmountably harder. A physical model of the universe may allow us to predict the subatomic particles ejected when two protons collide at high speed but building theories based on these collisions is difficult, especially when there are multiple equally valid explanations. Despite this, the pursuit of the Inverse Problem is critical to advancing scientific theory around a system's behaviour. Observations can only validate or falsify hypotheses but model prediction paves the way for novel scientific theories.

In this thesis, we tackle the inverse problem for cellular automata (CAs). These are simple yet powerful models of computation in which a discrete lattice of identical "cells" are simultaneously updated at regular time steps. The state of each cell depends exclusively on the state of the cells in a local neighbourhood around it in the previous time step. This localised interaction makes CAs a useful abstract representation for many physical and biological systems in the real world including tumor growth[20, 21], urban land use[22], and epidemics[23]. Much like these systems, CAs can exhibit chaos, nonlinear dynamics, and the emergence of complexity. As well as simulatory models, CAs are powerful computational engines due to ton a discrete lattice heir inherently parallel structure. This makes their study a useful endeavour in the field of distributed computation too[24].

Top-down investigations into CA behaviour are vast and varied[CITE]. Mathematical analyses seek to classify CAs and prove general results about long-term behaviour from intrisic properties. In the natural and social sciences, CAs are designed to model real world systems. Both of these endeavours seek to analyse the behaviour of a CA from its structure, transition function, and initial conditions. In this thesis, we explore a bottom-up approach where we deduce the underlying properties of a CA by observing its behaviour. In particular, we utilise evolutionary algorithms to search across several classes of CA. Evolutionary algorithms (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 and discover increasingly strong solutions as the fitness of the population increases and selection pressure grows. Using EAs, we tackle multiple optimisation problems from the imitation of particular CA behaviour to the generation of desirable long-term states.

1.2 Objectives

We develop a system to discover cellular automata transition functions from observations. We focus on two classes of CA in particular. The first are binary outer-totalistic CA (see 4.1.2) which are the discrete family of models in which Conway's infamous "Game of Life" CA belongs. For this reason, they are also called "life-like CA". The second are Gray-Scott models which simulate the behaviour of two diffusive chemicals. The density of each chemical in each cell is modelled as a lattice of real vector. Key aims of this project include:

1. **Learning Full Rule Dynamics**
   Deduce the underlying transition rules behind life-like and Gray-Scott CA using genetic algorithms and evolutionary strategies.

2. **Statistical Analysis of Rule Spaces**
   Assess which properties of cellular automata make them more or less predictable. Evaluate how this aligns with existing analyses in the literature.

3. **Practical Application of Learning Pipeline**
   Use the tools from step 1 to produce useful solutions to a real-world optimisation problems.

1.3 Contributions

- **Evolutionary Algorithm Toolkit**
  A versatile toolkit that implements multiple evolutionary algorithms to train and optimise different classes of CA. We use this to successfully predict the update rule of life-like CA from observations on random initial conditions. We show that this can be extended to Gray-Scott models where we predict the parameters of diffusion-reaction equations from simulations of chemical reactions.

- **Cellular Automaton Simulator**
  A system that can efficiently simulate discrete and continuous cellular automata. This allows various fitness functions to be implemented in the EA toolkit. This can also render snapshots of the CA directly during simulation which allows animations to be generated afterwards.

- **Procedural Maze Generator**
  A CA-based maze generation program that uses the EA toolkit to produce difficult mazes. Mazes generated are guaranteed to have a solution and are optimised to exhibit characteristics preferred by users.

1.4 Technical Challenges

- **Efficient Simulation**
  Each evolutionary algorithm runs on large populations over multiple epochs and fitness is calculated by repeated simulation of individual CAs. This means that thousands of simulations need to be run for each experiment so the simulator must be implemented in an efficient manner. This is especially pertinent for the Gray-Scott simulator which uses numerical integration to approximate solutions for the continuous differential equations that underlie chemical interactions.

- **Avoiding Local Optima**
  Evolutionary algorithms are very susceptible premature convergence. We tackle this by adjusting and testing a range of genetic operators, designing improved loss functions, and running large-scale simulations to produce heuristics that can assist with evolution.

- **Media Generation**
  Effectively learning from experiments and iterating strategies required statistical summaries, images, and animations to be produced. These provide quantitative and qualitative feedback for later analysis. Converting numerical data into high-fidelity media was a key point of focus.

Chapter 2

Preliminaries

2.1 Cellular Automata

A cellular automaton (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 finite 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[25], aperiodic tessellations such as as the Penrose tiling[26], and even randomly generated structures like the Voronoi partition[27].
- 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[28] to infectious diseases[29].
- 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.

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

There are two neighbourhoods that are frequently used on Euclidean lattices. 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.

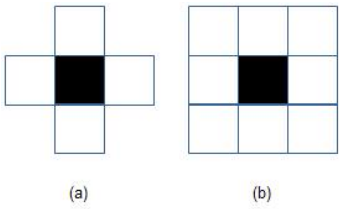


Figure 2.1: (a) von Neumann Neighbourhood and (b) Moore neighbourhood on a 2D square lattice [2]

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 [30]. 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:

$$\phi(\sigma_i(t), n_i(t)) = \begin{cases} 0 & \sigma_i(t) = 1 \text{ and } n_i(t) < 2 \text{ (Death by "exposure")} \\ 0 & \sigma_i(t) = 1 \text{ and } n_i(t) > 3 \text{ (Death by "overcrowding")} \\ 1 & \sigma_i(t) = 1 \text{ and } n_i(t) \in \{2, 3\} \text{ (Survival)} \\ 1 & \sigma_i(t) = 0 \text{ and } n_i(t) = 3 \text{ (Resurrection)} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

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 [CITE] and, as a corollary of the Halting Problem [CITE], this means that Life is undecidable. Given two configurations, it is impossible to algorithmically determine whether one will

follow the other.

Patterns found within Life include still lifes like the *block* which are fixed-point solutions to the transition function 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. It is possible to discover new stable patterns by repeatedly running specific rules on random initial patterns of a pre-determined density (called soups) and classifying the objects remaining after transient reactions have dissipated. Large-scale experiments of this nature are called "soup searches"[CITE].

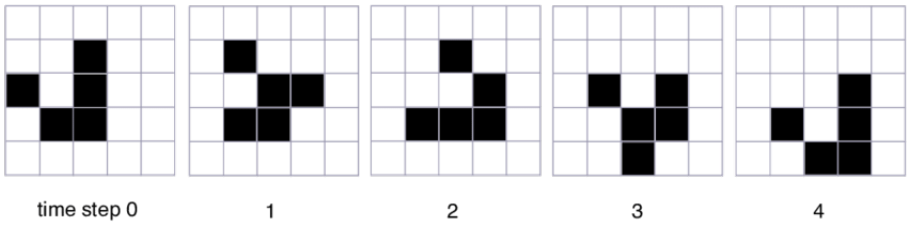


Figure 2.2: The glider pattern in the Game of Life [3]

A CA is considered "Life-like" if it exists on a 2D lattice, has binary state, uses the Moore neighbourhood function. Life-like cellular automata 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).*

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

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.*

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

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 separately 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. A B/S rulestring represents the transition function of an outer-totalistic CA in a form called birth-survival notation.

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:*

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

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

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.

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 class 4 behaviour [4] and is Turing complete [31]. 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.

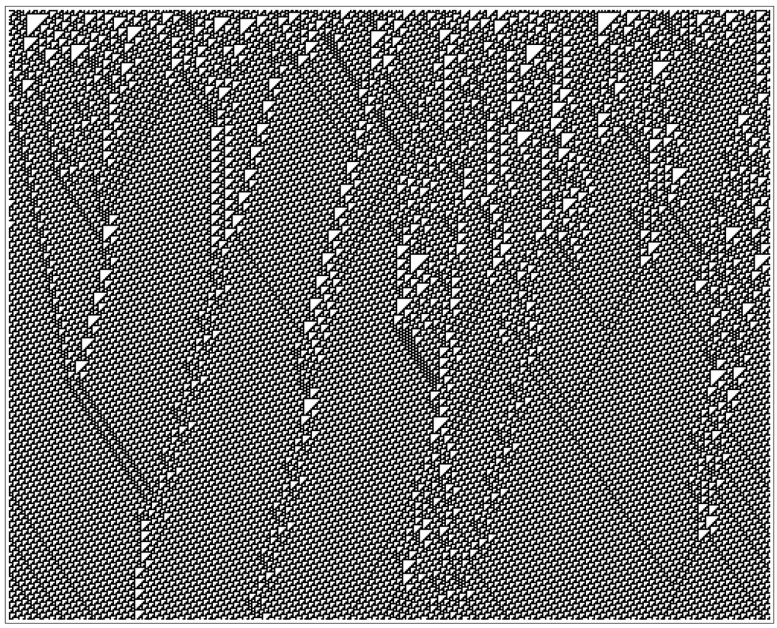


Figure 2.4: Rule 110 progression with random initialisation [4]

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.

In *The Chemical Basis for Morphogenesis (Turing, 1952)[32]*, 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[33] to the distribution of matter in the Milky Way[34].

Figure 2.5: Complex patterns on sea shells(left) can be replicated using Turing patterns (right) [5]

2.2.1 Gray-Scott Model

Turing patterns arise out of two component reaction-diffusion systems. One specific example is the Gray-Scott model[35] 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

$$U + 2V \rightarrow V \tag{2.2}$$

To compensate for U being consumed The system replenished 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)**.**

$$\frac{\partial u}{\partial t} = -uv^2 + f(1-u) + r_u \nabla^2 u \tag{2.3}$$

$$\frac{\partial v}{\partial t} = uv^2 - (f+k)v + r_v \nabla^2 v \tag{2.4}$$

u and v represent the densities of each component in a given cell. $\dot{u}$ and $\dot{v}$ represent the change in these densities. 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 2.2 where 1 U molecule reacts with 2 V molecules which is why the term is a product of u^1 and v^2. The second term represents external inputs and outputs. In 2.3, the feed rate is

multiplied by $1-u$ so that replenishment depends on current density. In 2.4, 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.

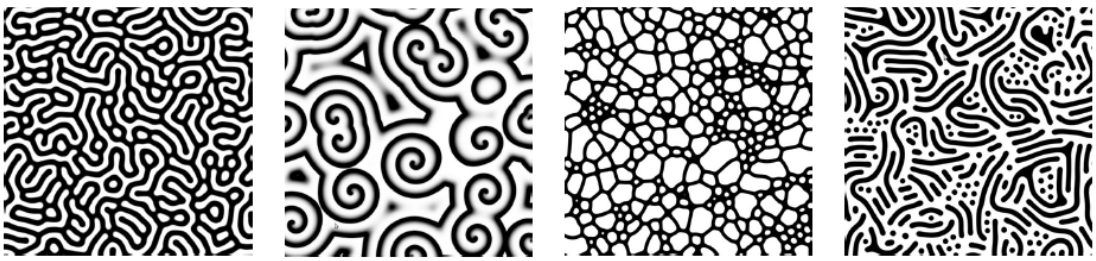


Figure 2.6: Turing patterns arising out of Grey-Scott model simulations[6]

Chapter 3

Related Works

This chapter summarises recent work on the analysis and learning of cellular automata. We focus on two classes of automata. The first are binary outer-totalistic CA, also known as life-like CA (see Def 2.3). The second are continuous reaction-diffusion CA which model simple chemical reactions. As we will see, these are a natural extension of life-like CA since, under certain constraints, life-like CA can also be interpreted as discrete reaction-diffusion simulations with each cell accommodating either the reactant or the substrate - a binary choice.

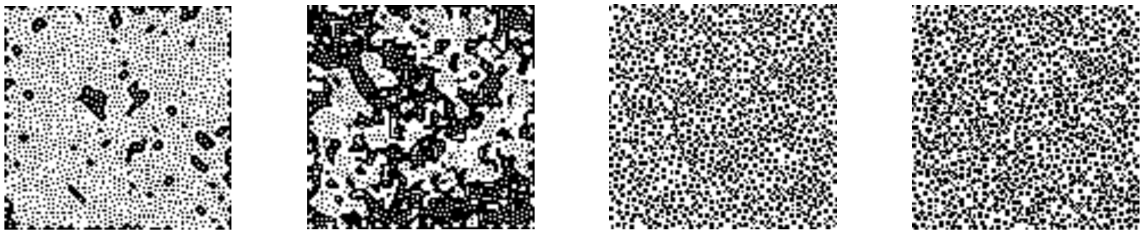
3.1 Life-Like CA: Exploration

The choices of lattice geometry, neighbourhood function, state variable, and transition rule define the behaviour of a CA. Fixing the former three factors, Wolfram[36] classified CAs based on transition rules as follows:

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

Early attempts to categorise 2D cellular automata by Packard and Wolfram[37] extend Wolfram's original 4 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, we could say that they are inherently *via negativa* definitions, based on what a CA cannot do regardless of initial conditions. 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[38], 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.[7] 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



(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 [7]

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 and quantitative factors like 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. 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 automata have been proven to span multiple classes[39].

This dilemma is alleviated to some degree by Eppstein's four-way classification[8] 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.

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 [CITE]. This is called soup searching. In a similar vein, we will use soup searches to approximate the fertility and periodicity of all rules in the life-like CA rulespace.

3.2 Life-Like CA: Learning

A seminal work by Meyer et al.[40] looks at learning 2D CA neighbourhood functions using genetic algorithms. Later works by Mitchell et al. [9] 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

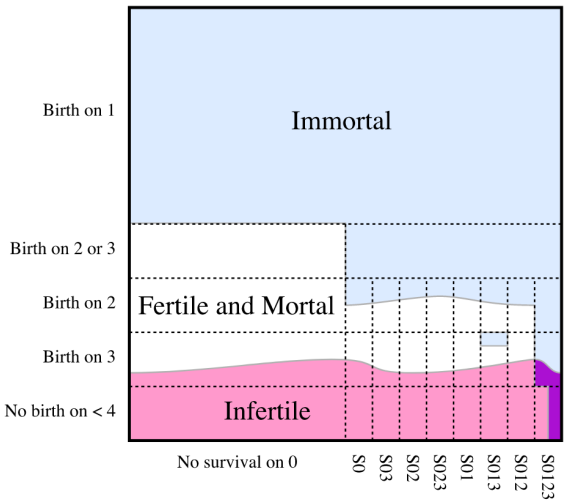


Figure 3.2: Map of fertile, infertile, mortal, and immortal regions in binary-state RDCA rulespace [8]

genetic programming to a broad variety of tasks including the CA majority classification problem [10]. Incremental improvements have been made to since then with Breukelaar and Bäck notably delivering experimental evidence that the CA inverse design problem using evolutionary computation is more tractable in higher dimensions[41]. 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)*[40], 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.[42] used results from this work to predict the dendritic solidification structure of NH_4BR.

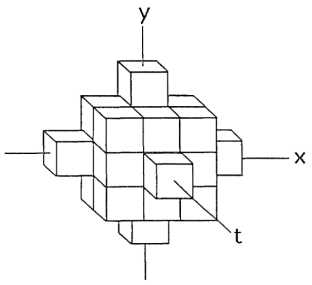


Figure 3.3: 20-cell, two step neighbourhood in space and time

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 2 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 $s_1, ..., s_m$. The fitness function

used is

$$F = I - \frac{2^m}{N}$$
$$\text{where } I = \sum P(s, s_1, ..., s_m) \log_2 \frac{P(s, s_1, ..., s_m)}{P(s)P(s_1, ..., s_m)}$$

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 GAs to efficiently traverse an opaque search space. The algorithm precisely learns neighbourhoods interior to the master template such as the 1 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 1 time step von Neumann neighbourhood, the algorithm learns a neighbourhood set that produces correct behaviour 96% of the time.

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[43] 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 CAs 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 as follows.

Definition 3.1 (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.* $\forall t > T$*:*

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

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 below, allows consensus to be reached in O(n) time and, for n=149, achieves success on 81.6% of inputs[44]. Modifications throughout the 1990s incrementally improved this classifier[45]. 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.

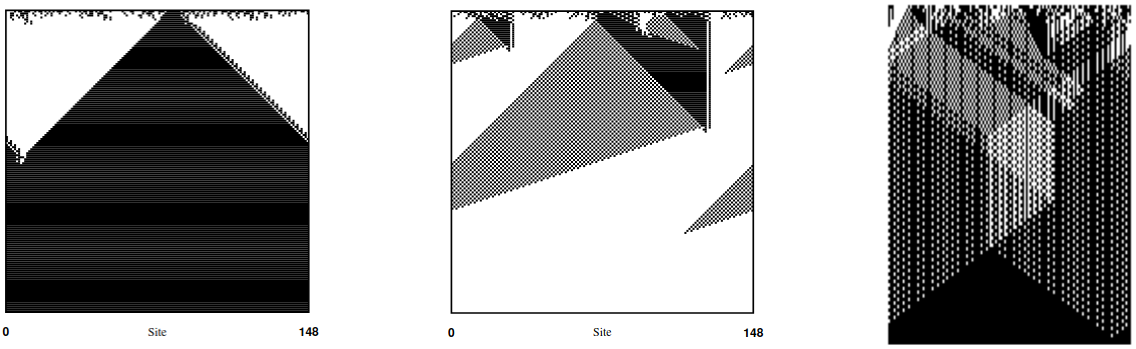
Definition 3.2 (GKL Classifier)**.** *A GKL density classifier is an elementary CA on periodic boundary conditions with transition function*

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

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

A seminal series of work by Mitchell, Crutchfield, and Das[9] 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[CITE].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". 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 automaton.

A later work by Andre, Bennett, and Koza[10] 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 automaton.



(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.[9] and (c) by Andre et al.[10]

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[46]. However, evolutionary computation still surprises in its ability to find approximations of ever-increasing performance.

3.2.3 Learning 2D Transition Functions

Few attempts have been made to extend evolutionary algorithms to multidimensional cellular automata. A key series of work by Breukelaar and Bäch[11] 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 operator alone. In other cases, like the OR operator, we can deduce that the result will be true from a single operand being true. For the majority problem, tests showed that results with a fitness of up to 70% could be evolved using a von Neumann neighbourhood. For the logical problems, a von Neumann neighbourhood could elicit results with over 90% accuracy and a Moore neighbourhood allowed a transition function with perfect accuracy to be evolved. 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 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. 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. 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.

Early work by Wulff and Hertz in 1992[47] uses a lattice-shaped neural network style structure to learn CA dynamics. Each node in this lattice is a Probabilistic Logic Node,

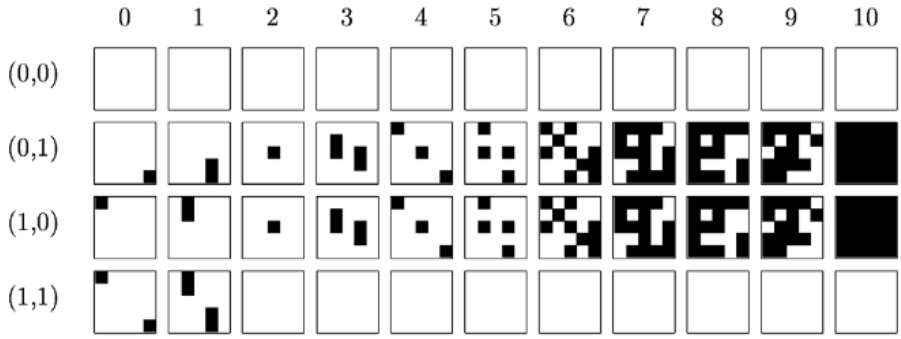


Figure 3.5: Iterations of XOR CA for 4 possible input states

Figure 3.6: Goal bitmaps for morphogenesis experiment[11]

also known as a $\Sigma - \Pi$ unit [48]. These units are capable of representing any boolean function. That is to say $\forall f : \mathbb{R}^N \rightarrow \{-1, 1\}, \exists\ w_1, w_2, ..., w_N \in \mathbb{R}$ such that,

$$f(S_1, S_2, ..., S_N) = sgn\left[\sum_{j=1}^{N} w_j \prod_{i \in I_j} S_i(t)\right] \tag{3.1}$$

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.3 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 automata were tested on. However, at the time, it was believed that 1D CAs could not exhibit class 4 behaviour so 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 with the network only being able to do so for 4 out of the 9 candidates in the 1D case. This work also divided class 3 elementary CAs 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 4 CAs.

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)*[12], 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. Later works build upon

these techniques to learn full dynamical systems such as the Boids algorithm[49]. 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 CAs 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.

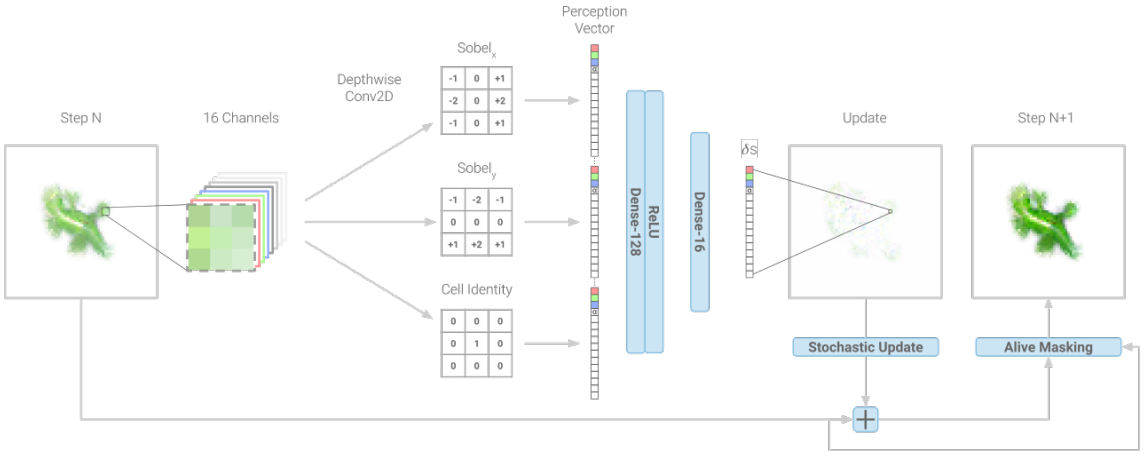


Figure 3.7: Single training step for a neural cellular automaton[12]

Currently, no attempts have been made to use evolutionary computation to learn full rule dynamics of life-like cellular automata.

3.3 Gray-Scott Systems: Exploration

Pearson[13] classifies patterns arising from Gray-Scott models into 12 categories based on temporal factors such as stability and decay rate and spatial factors such as regularity and emergence of subpatterns like spots and stripes. 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 $\pm 1\%$ random noise.

Pearson discovered a threshold near which this interesting behaviour could 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 3.8 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.

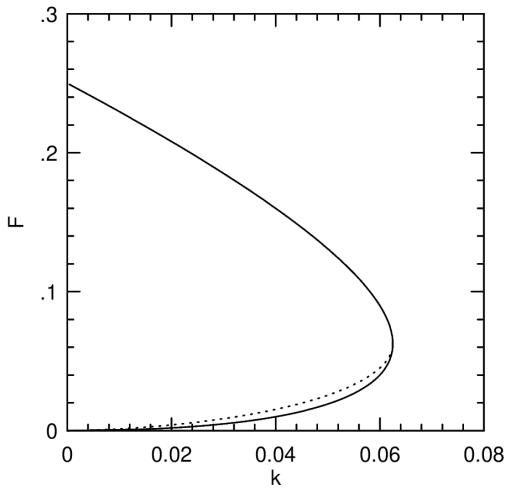


Figure 3.8: Phase diagram of Gray-Scott systems[13]

However, Pearson's analysis is limited by the initial condition he uses. 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, Munafo[50] reveals 5 new types of pattern differentiated from others primarily by oscillation and spot shape. In imprecise terms, the feed rate controls oscillation, stability, and chaos while the kill rate controls the shape and quality of objects formed.

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.

1. 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.

2. 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. However, these are not class 2 or 4 as the rate of change approaches a non-zero constant.

3. Class 4-a: These are hypothetical class 4 systems that are subject to the halting problem. No example has yet been proven.

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

3.4 Gray-Scott Systems: Learning

As a system of partial differential equations, Gray-Scott systems are typically solved using numerical analysis. This includes finite difference methods[51] and finite elements[52]. There are not many attempts learn solutions using machine learning, possibly due to the

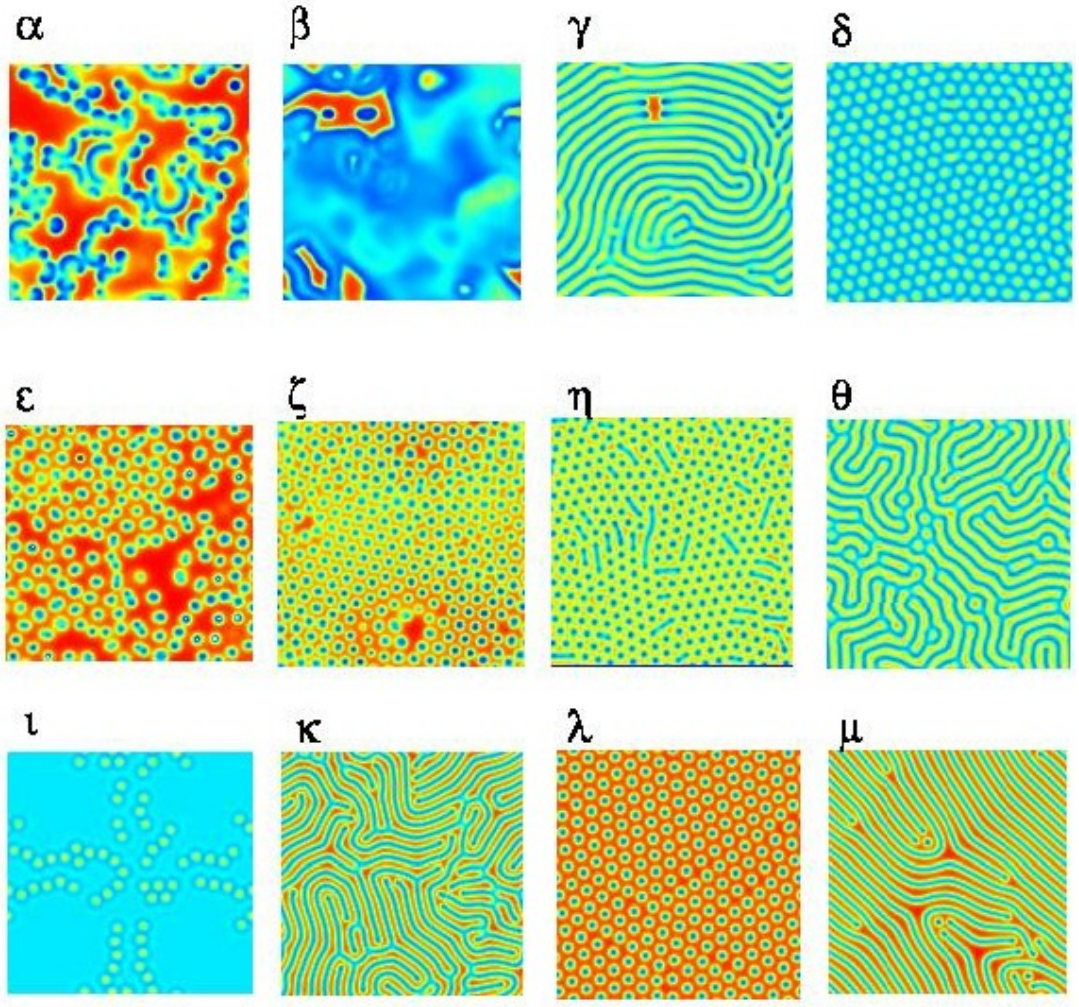
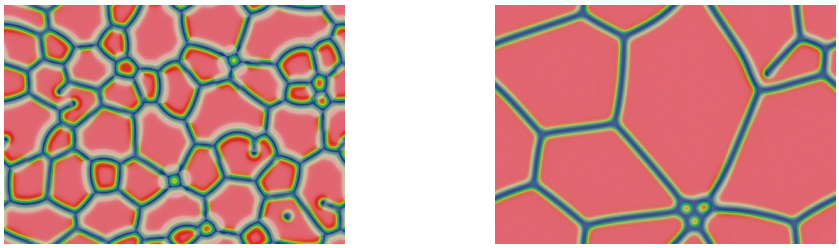


Figure 3.9: Pearson's 12 categories of Gray-Scott systems[13]

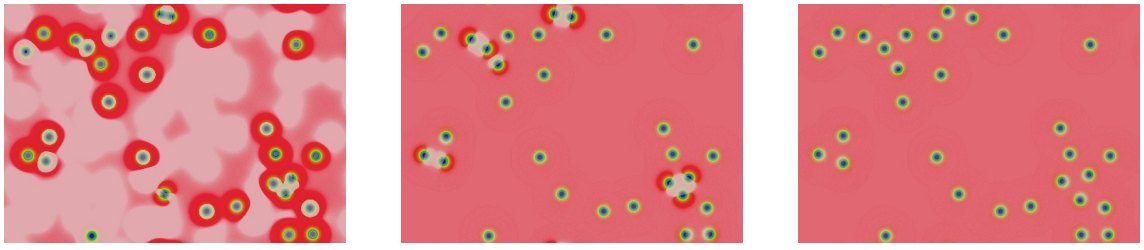


(a) f = 0.090, k = 0.059 (b) f = 0.102, k= 0.055

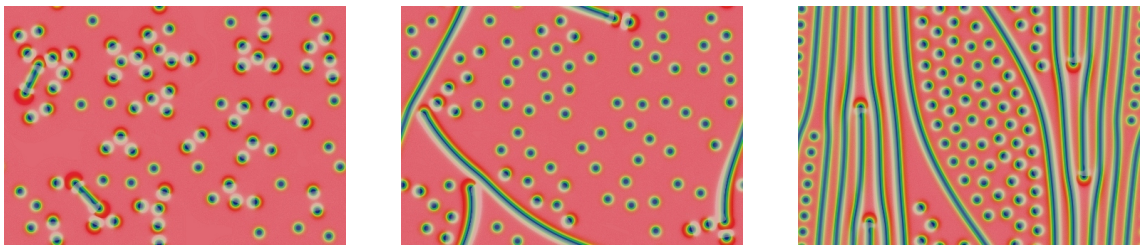
Figure 3.10: The ρ class of Gray-Scott pattern resembles a set of soap bubbles under surface tension. These clearly do not resemble any of the 12 Pearson categories. [14]

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)* where 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

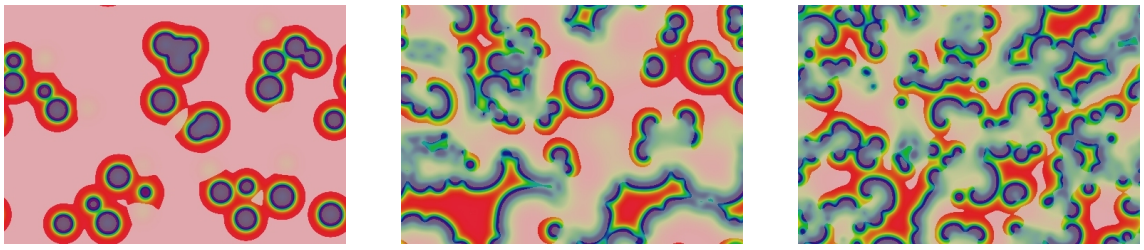


f = 0.046, k = 0.067

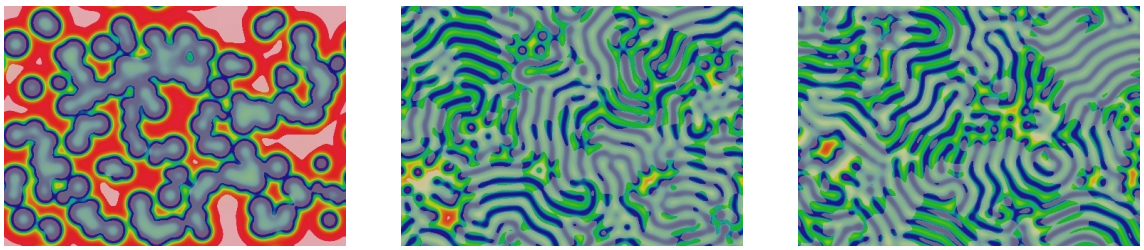


f = 0.066, k = 0.065

Figure 3.11: Class 2 behaviour (top) against class 2-a behaviour (bottom). Time moves left to right. [14]



f = 0.014, k = 0.049



f = 0.022, k = 0.053

Figure 3.12: Class 3 behaviour (top) against class 3-a behaviour (bottom). Time moves left to right. [14]

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.

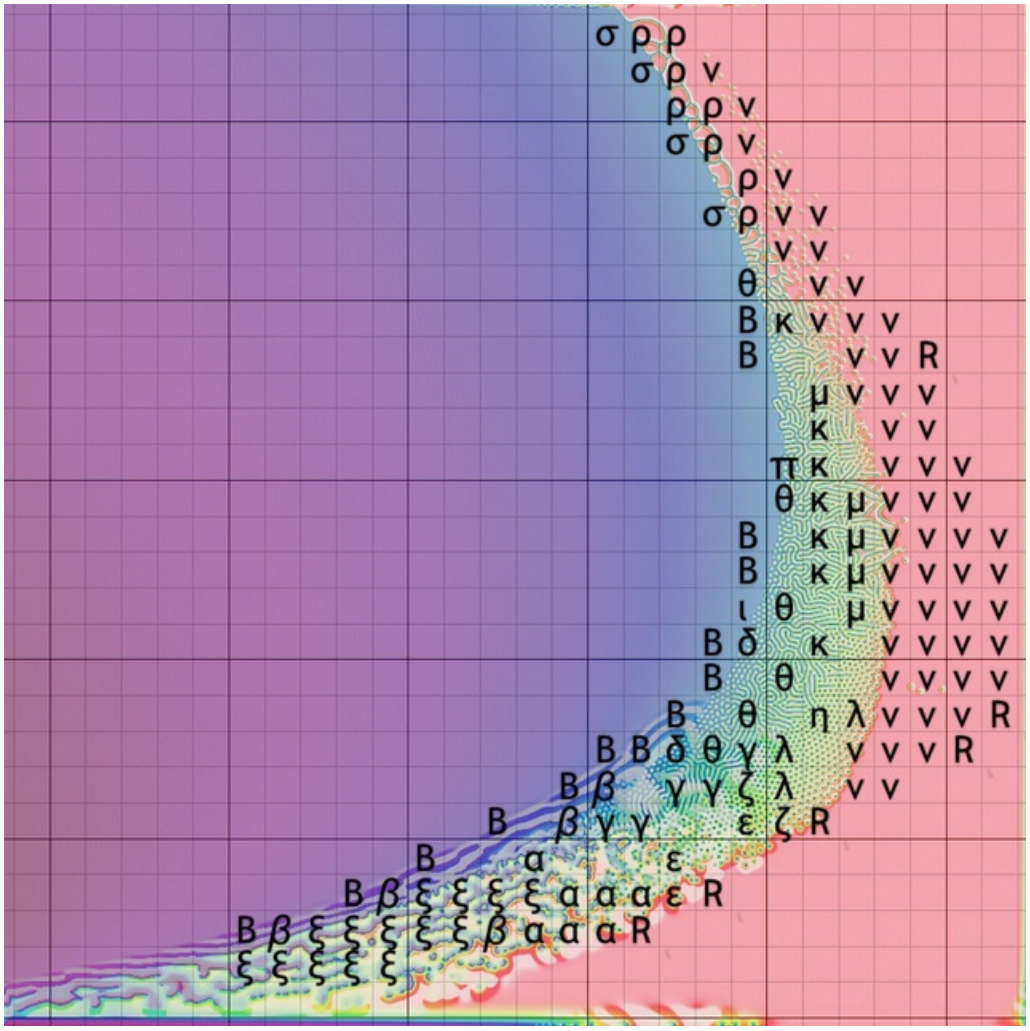
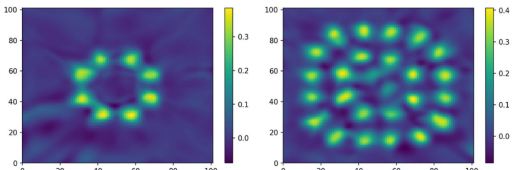
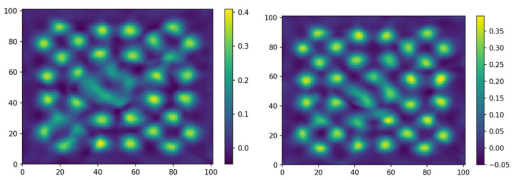


Figure 3.13: Map of Gray-Scott parameter space depicting all 19 Pearson-Munafo classes [15]

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, the computational power required here is significantly reduced compared to a "blind" neural network due to the physical constraints imposed in the loss function. This also makes the PINN better able to avoid local optima.

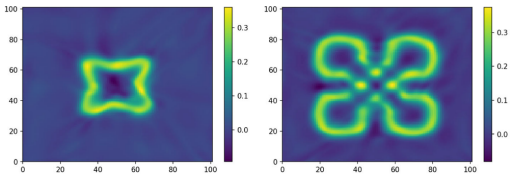


(a) $t = 1250$ (b) $t = 2750$

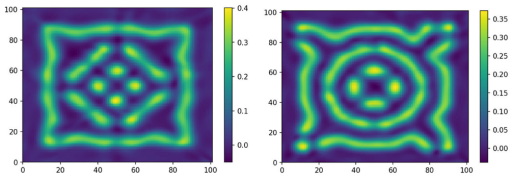


(c) $t = 3750$ (d) $t = 4750$

(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 [16]

Chapter 4

Part I: Life-Like CA

To learn life-like CA, we built an evolutionary algorithm toolkit. This is written entirely in Python making use only of basic data manipulation and scientific computing libraries such as NumPy, SciPy, and Pandas as well as media libraries like Matplotlib and Python Imaging Library.

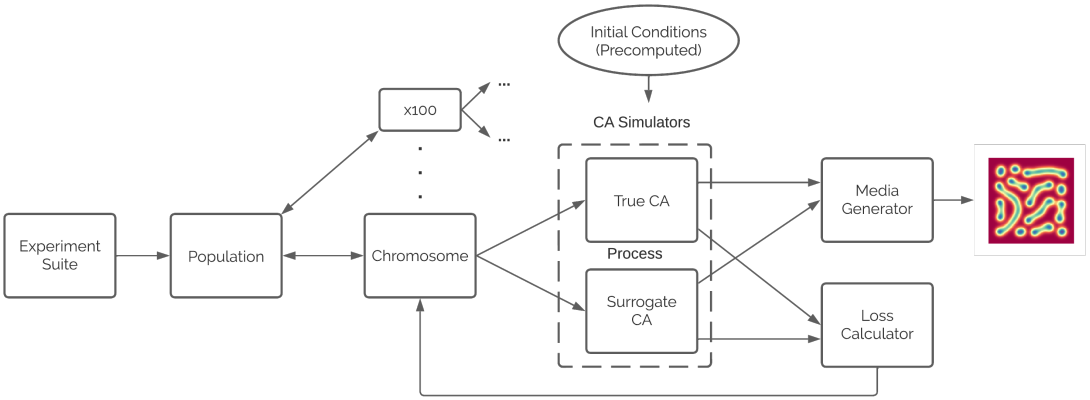


Figure 4.1: Process for learning Life-Like and Gray-Scott CAs

It features the following key classes:

- Experiment Suite: Includes methods and configurations for running evolutionary experiments.
- Population: Handles population-wide actions like crossover and elitism as well as tracking evaluation metrics like convergence and number of unique individuals seen.
- Chromosome: Handles individual actions like mutation, and fitness calculation.
- Simulator: Runs binary-state cellular automata and caches state data when needed

When learning a new class of CA or optimising for a new objective, an evolutionary algorithm is created by writing a new instance of the Population and Chromosome classes with implementations of the relevant evolutionary actions and objective function. An instance of the Simulator class must be created to implement the transition function from the parameters encoded in the chromosome. Experiments to test the algorithm can be shared across different objectives but the user may choose to write custom experiments depending on the goal at hand. Implementation of these experiments and their results are covered in detail in Section **??**. We first explore the algorithms behind the learning

process, the implementations of the Population, Chromosome, and Simulator classes, and delve into a practical application of the learning algorithm in the form of a maze generation program.

4.1 Learning Process

4.1.1 Simulator

Due to the undecidability of various CA rules, the state of an automaton after a certain number of steps cannot, in general, be calculated without simulating each transition in turn. For this, a CA simulator was built.

The simulator stores the state of a 2D square CA of side length N in an NxN NumPy array. The CA is initialised with birth set B and survival set S which are given as direct arguments or calculated from the chromosome as shown in 4.1. 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

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

Then, 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 using the birth and survival sets as follows

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

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 $X^{(t+1)} = X^{(t)}$, the cached state, then no further steps are calculated since a fixed point has been reached and $X^{(t+n)} = X^{(t+n-1)} = ... = X^{(t)}$. Otherwise, the current state is cached and the simulator continues until n steps have elapsed or a fixed point is reached at some later stage. The simulator does not automatically detect periods of length greater than 1. The simulator allows the initial state $X^{(0)}$ to be set randomly with a particular density or set explicitly to a particular matrix. The latter is useful when simulating a surrogate CA which mimics the true CA and has to be updated regularly with known data. The simulator also allows the CA states to be saved at regular intervals as NumPy arrays and as images which are concatenated together into animated gifs.

4.1.2 Genetic Algorithm

A genetic algorithm (GA) is a particular type of evolutionary algorithm that acts on a population of indirect encodings called "chromosomes". The structure of the chromosome is called the "genotype" and the structure of the corresponding candidate solution is called the "phenotype". In this case, our genotype is a binary string encoding a transition function for a life-like CA. The corresponding phenotype is the behaviour of that CA over time. Genetic algorithms use actions called selection, mutation, and crossover to optimise the population and each of these is inspired by principles in biological evolution. We begin by formalising the structure of a genetic algorithm.

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 objective function, also known as a fitness function, to compare and select the top candidates. We seek to maximise fitness. Alternatively, selection may be based on minimising a loss function. Recombination produces a set of children that have similar properties to some subset of the parents. This exploits the cumulative generational knowledge in the parent candidates. Mutation explores new areas of the search space by perturbing properties of the parents and children. 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, certain 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.

The optimization goal in this case is to find the transition rule that generated the observations made. A transition rule can be represented in a number of ways. Most intuitively, consider a birth and survival sets which dictates the number of neighbours that elicit a dead cell to become alive or a living cell to remain alive respectively. This can be encoded in a binary string which itself can be stored in integer form as shown in equation 4.1.

$$\begin{aligned}
&\text{Number of neighbours: } 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \mid 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8 \\
&\text{Binary representation: } 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0 \mid 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0\ 0 \\
&\qquad\qquad\qquad\qquad\quad \uparrow \qquad\qquad \uparrow\uparrow \\
&\text{Set representation: } \quad \text{B:}\{3\} \qquad\quad \text{S:}\{2,3\} \\
&\text{Integer representation: } \texttt{0b000100000001100000} = 16480
\end{aligned} \tag{4.1}$$

Each binary string chromosome has length 18, so the discrete search space is of size $2^{18} = 262144$. A population of μ chromosomes is chosen randomly from a distribution that is uniform over the density of the binary representations. As shown by [CITE], this is preferable to a distribution that is uniform over the integer representations since the density of the rule is more correlated with complexity properties than the integer value of the rule [EVIDENCE]. When initialising a random chromosome, a density ρ is picked uniformly from $[0.0, 1.0]$. Then, each bit is a sample from the $Bernoulli(\rho)$ distribution.

At each iteration, the algorithm explores the search space through crossover and mutation, evaluates candidates through fitness calculations, and concentrates learned information through selection. In accordance with the $\frac{1}{5}$ rule [CITE], we produce λ new children in each expansion stage and reduce down to μ elite candidates in each contraction stage with $\lambda \approx 4\mu$.Single-point crossover is used to produce new children. A crossover point c is picked between 1 and 17 and each of the parents is split at c. The left half of one parents chromosome is concatenated with the right half of the others chromosome. This process is repeated picking pairs of parents with replacement until enough children have been created. Mutation is applied by flipping each bit in a chromosome with probability $\frac{1}{18}$ such that the expected number of bit flips per chromosome is 1. This is inspired by biological point mutation where individual base pairs in a biological genome are altered due to copying errors or environmental exposure. In reality, this is implemented more concisely by generating a mutation mask with expected density $\frac{1}{18}$ and applying XOR between the chromosome and mutation mask.

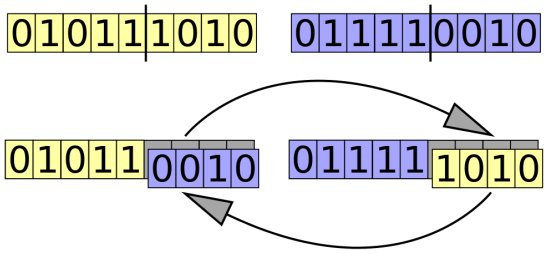


Figure 4.2: Visualisation of single-point crossover on 9-bit chromosomes. [17]

When producing the next generation we consider selection criteria based on fitness and age. In a $(\mu + \lambda)$ selection setting, we produce a population of λ children from μ parents and the next generation is selected from the collective. In a (μ, λ) setting, 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. 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.

4.1.3 Fitness

Fitness is calculated by running multiple CAs 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 CAs are tested on the same set of initial conditions sampled uniformly on densities between 0 and 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 between $X_{\delta_{k+1}}$ and 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 between 0 and 1. The fitness is defined by taking the mean of the losses and subtracting from 1.

Definition 4.1 (Life-Like Fitness Function 1)**.** *We define the fitness F as*

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

where $\oplus$ is the XOR operator

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 regards to step sizes, we consider 3 possibilities.

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

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 4.3. 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 $(1/(D_{max} - D_{min}))^K$. 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 another observation.

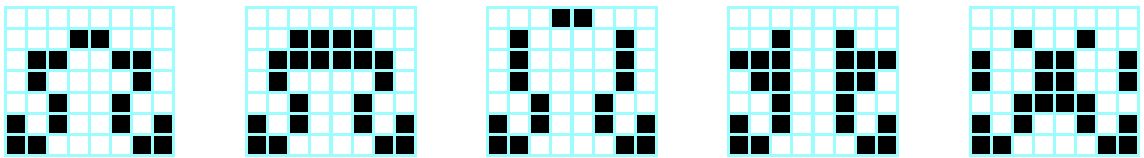


Figure 4.3: *Fumarole*, a 5-period oscillator in the Game of Life. [18]

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 4.4 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 figure 1. To mitigate this affect, a multi-resolution fitness function is proposed which uses convolutions to capture density across broad regions.

Definition 4.2 (Life-Like Fitness Function 2)**.** *We define the multi-resolution fitness F as*

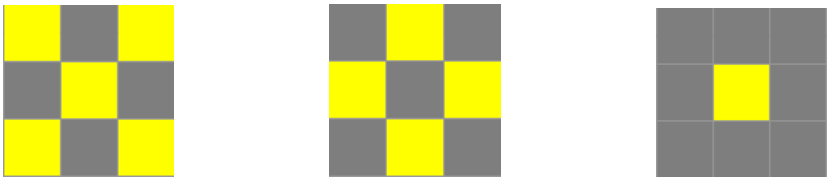
$$F = 1 - \bar{L}$$

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

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

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 to ensure 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.



(a) Goal State

(b) Prediction 1, fitness = 0

(c) Prediction 2, fitness = 0.55

Figure 4.4: Example of fine-grain loss failing to capture macroscopic properties

4.2 Maze Generation

As a practical application of the genetic algorithm, we built a maze generation program. This uses a cellular automaton rule to randomly generate a unique maze-like structure which is modified to ensure a solution path exists. A genetic algorithm, based on the one designed in subsection 4.1.2 is used to find the chromosome that tends to produce the "best" maze according to a user-inputted definition of "best" using quantitative factors such as length of solution and number of dead ends. The maze is made up of cells in one of two states. The "live" or 1 state represents walls and the "dead" or 0 state represents possible path cells. There are also two special states which represent the start and goal cell.

This has some similarities to a work by C. Adams[53] which also looks at the application of CAs in maze generation. 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).

4.2.1 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 for a fixed number of iterations, typically 50, using the birth and survival sets encoded in the chromosome. This process is explained

in detail in 5.1.1. The region search stage uses iterated breadth first searches to find all disconnected regions within the maze. The region merge stage connects these regions randomly until a connected path exists from the start cell to the goal cell. In order to perform the merge stage, two data structures are populated during the find stage. The first is a hashmap from each cell to the region number of the region it occupies. The second is the reverse, a hashmap from each region number to the set of cells in that region.

Algorithm 2 Region Finding 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
```

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 will longer solution paths are considered fitter and the merging algorithm connects regions in horizontal bands sweeping left-to-right (as in [53]) 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 unsuccessful. If a chromosome does not yield a minimum percentage of successful simulations, it is assigned a fitness of 0 and usually removed from the population in the following iteration.

Algorithm 3 Region Merging 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 $d \in$ ZeroNeighbours(c)}
      visited ← visited $\cup$ {regions[r] for $r \in$ 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.

4.2.2 Genetic Algorithm

The algorithm, as before, initialises a random population of chromosomes and evolves them using bitwise mutation, single-point crossover, and $(\mu+\lambda)$ truncation selection. The aim of the fitness function is to assess the maze using quantitative metrics that can be calculated in a computationally efficient way. 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. It can also easily be computed 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. This calculation takes linear time with respect to the number of cells in the automta. In the final algorithm, two metrics are used: the number of dead ends and the solution path length. These factors work well as they 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. Assuming a maze with at least one solution, both metrics can be calculated simultaneously in a single breadth-first search traversal. A cell is considered to be a dead end if all its neighbours are wall cells or vacant cells that have already been visited.

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. [EXPAND HERE] However, this is not normalized as the solution length and number of dead ends are not on the same scale. Furthermore, 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 where each chromosome is ranked separately by each metric and the fitness function is defined as $f(c_i) = r_s + \lambda r_d$ 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. The top μ candidates by fitness are picked.

4.3 Software Engineering Design

Design decisions regarding algorithms and process have been discussed throughout Chapter refchap:part-1. In this chapter, we summarise key software engineering design decisions when writing the evolutionary algorithm toolkit.

4.3.1 System Design

Figure 4.5 depicts the sequence of events that is triggered in a typical experiment learning life-like CA. There are 3 major loops that run during an experiment. The outer loop performs generational learning. The Population is updated on each of these iterations. The inner loop calculates loss. Each of the two CAs 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 from which to begin. The final loop reruns top solutions to generate statistics and media.

4.3.2 Chromosome Class

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 the binary form and sets form of the rule simultaneously. However, this requirement means both sets must be updated every time the binary rule string is updated and vice versa.

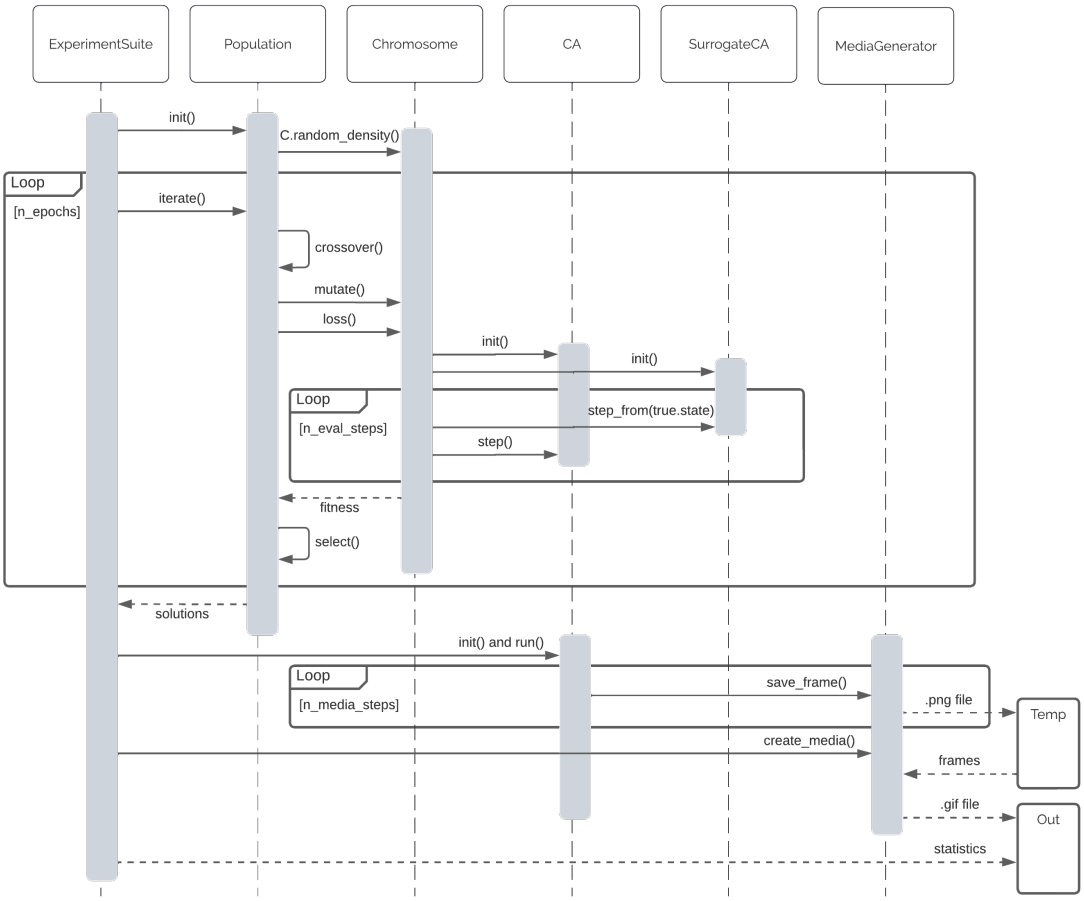


Figure 4.5: UML sequence diagram of life-like CA learning

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 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`, it 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 decision was regarding initialisation. Two common initialisation methods for chromosomes are based on sampling uniformly across density and sampling uniformly across value. 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.

4.3.3 Media Generation

To extract insights from the toolkit, metrics and media must be generated. 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 automaton 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 automaton is also saved as a png image and in its original array form as an npy file. The npy file is saved in case a later analysis requires the CA to continue running from where it finished. The temporary frame files are deleted to save memory. Analysis and visualisation of population-wide properties is done manually after the experiments have completed.

Chapter 5

Part II: Gray-Scott Systems

To learn Gray-Scott models, we extend our evolutionary algorithm toolkit further to include more diverse methods such as evolutionary strategies and particle swarm optimization.

5.1 Learning Process

5.1.1 Simulator

In this setting, we run a discretized simulation of a continuous system so there are more subtleties to consider. We use two NumPy arrays 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 in Def 2.5. We first define discrete approximations for the derivative of each density.

Definition 5.1 (Gray-Scott Model)**.**

$$\dot{u} = -uv^2 + f(1-u) + r_u\mathcal{D}(u)$$
$$\dot{v} = uv^2 - (f+k)v + r_v\mathcal{D}(v)$$

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 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 by in Compeau's Biological Modelling book[54] is

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

Another popular choice is the nine-point stencil[55].

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

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 briefly 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

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

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

$$0 = \frac{u(x+h)+u(x-h)-2u(x)}{h} - u^{(2)}(x)h - 2\frac{u^{(4)}(x)}{4!}h^3 + ...$$
$$\implies \frac{\partial u}{\partial x^2} = \underbrace{\frac{u(x-h)-2u(x)+u(x+h)}{h^2}}_{=:\,\Delta^{(3)}u(x)} + O(h^2)$$

By combining two of these we get a two dimensional 5-point stencil.

$$\begin{aligned}
\Delta u(x,y) &= \frac{\partial u}{\partial x^2} + \frac{\partial u}{\partial y^2} \\
&\approx \frac{u(x-h,y)-2u(x,y)+u(x+h,y)}{h^2} + \frac{u(x,y-h)-2u(x,y)+u(x,y+h)}{h^2} \\
&= \frac{u(x-h,y)+u(x+h,y)-4u(x,y)+u(x,y-h)+u(x,y+h)}{h^2} \\
&= \frac{1}{h^2}\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix} u(x,y) = \Delta_h^{(5)}u
\end{aligned}$$

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

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

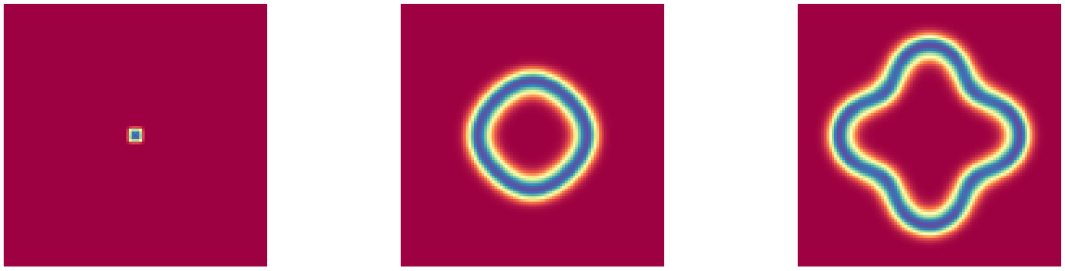
In this sense, the 9-point stencil approximates the second partial derivative across 8 directions. The positive and negative x and y directions as well as the 4 diagonal directions between them. The constant factor of $\frac{1}{2}$ is consumed by the diffusion constant. We can now convolve $\Delta_h^{(9)}$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

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

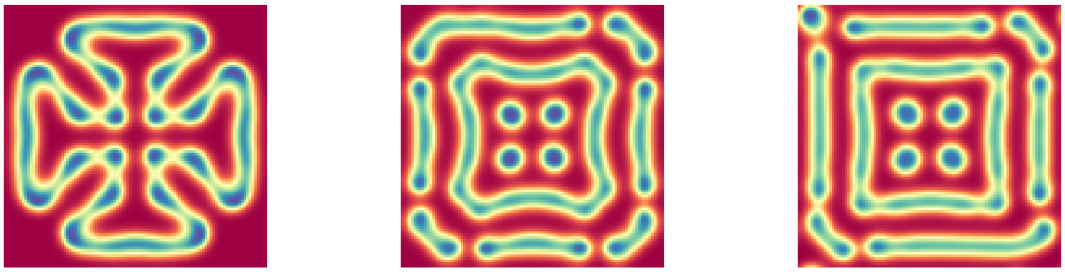
There are a number of constants that affect the behaviour of simulation. These include the diffusion constants, time delta, initial densities, feed rate, and kill rate. We pick a diffusion constant ratio of $D_u = 2D_v$ as this ratio has been shown, in the literature, to elicit complex behaviour[CITE]. The temporal resolution can be adjusted using the time delta δt. Through spot checks, we find values in the range 0.5 to 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[13], creates a square perturbation with $\pm 1\%$ Gaussian noise. The *splatter* setting produces n perturbation "seeds" of size 3×3. 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 much more 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) $t = 2000$

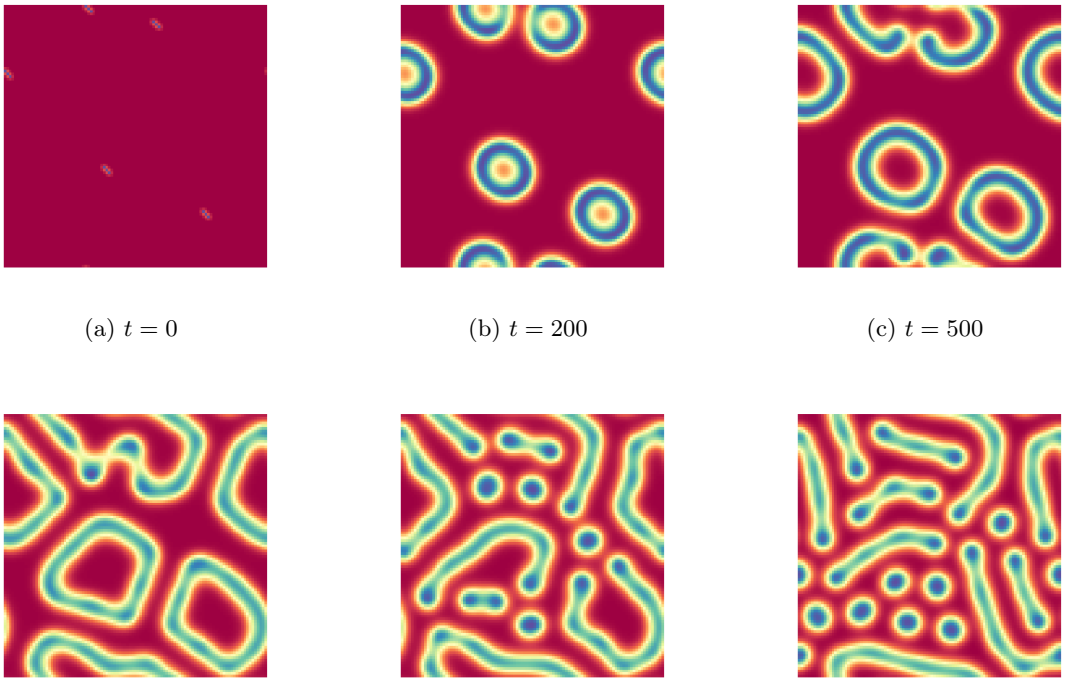


(d) $t = 3000$ (e) $t = 4000$ (f) $t = 5000$

Figure 5.1: Gray-Scott simulation under *patch* initialisation ($f = 0.03$, $k = 0.06$)

5.1.2 Chromosome

The chromosome is made up of two vectors, *state* and *control*. The state vector contains the parameters of model, feed and kill rate, as two real numbers, f and k. The control vector encodes the volatility of these parameters as real numbers df and dk. The toolkit allows the user to initialise chromosomes in one of two ways. The first choice is to sample a random uniform distribution with $0.0 \leq f \leq 0.30$ and $0.0 \leq k \leq 0.08$. The boundaries are based on the nontrivial region depicted in 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 3.8. Values of f is 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 $k = 0$.



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

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

Figure 5.2: Gray-Scott simulation under *splatter* initialisation ($f = 0.03$, $k = 0.06$)

5.1.3 Fitness and Selection

Once the population of chromosomes has been initialised, we consider an objective function. As before, we simulate a true CA based on the goal parameters and maintain a surrogate cellular automata 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. In roulette selection, each member of the population has a probability of progressing to the next generation equal to their fitness divided by the sum of the fitnesses of every individual in that population. In truncation selection, individuals are ranked by fitness and a fraction of the top ranking candidates are picked.

Selection comes in two flavours, *plus* and *comma*. Under plus-selection, parents and children alike are considered for progression to the next generation. This is the standard setting we used for life-like CA too. Under comma-selection, only offspring can progress to the next generation. We implement this setting as it enforces a "lifetime" of one generation on all candidates which promotes exploration outside of local optima. The downside to this is that 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.

Crossover and mutation are dependent on the evolutionary algorithm being used. The two key evolutionary algorithms implemented are an evolutionary strategy and a genetic algorithm. We explore these in turn.

5.1.4 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[CITE]. 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 more flexibility in tuning both processes.

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 define 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 $\rho \leq \mu$. Common recombination operators include:

- Discrete: For each of the offspring's state variables, one of the ρ parents is picked from which to inherit.
- 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

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

Where we define the intermediate recombination operator as the arithmetic mean

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

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 $\tau_1 N_i(0, 1)$ and a specific mutation applied to each component individually of $\tau_2 \boldsymbol{N_i(0, I)}$ with $\tau_1 = n^{-\frac{1}{2}}$ and $\tau_1 = n^{-\frac{1}{4}}$.

5.1.5 Genetic Algorithm

The genetic algorithm performs crossover, recombination, mutation, and selection as usual. The crossover method used is a combination of uniform crossover and blended crossover (BLX-α). Uniform crossover is used on the state property of each chromosome. For two

parents x and y, a property of a child c is defined by a value sampled uniformly from the interval between the corresponding property in the parents.

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

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.

Definition 5.2 (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*

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

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 another uniform crossover and 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.

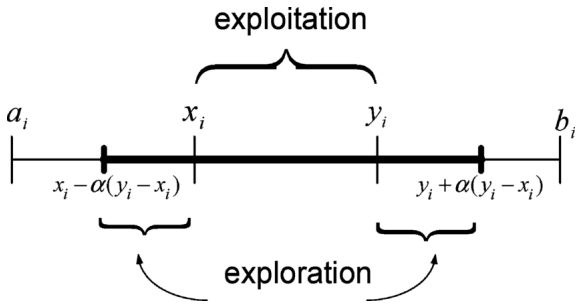


Figure 5.3: Blended Crossover [19]

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

5.2 Software Engineering Design

Chapter 6

Evaluation

6.1 Metrics

In order to assess the effectiveness of evolutionary algorithms and analyse the types of solutions they produce, we design metrics with which to categorise CA and assess EAs. We also use properties discussed in Section 3.1

6.1.1 Convergence Metrics

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. If a state previously visited at time step t is produced again at step $t + \delta$, the rule is said to have converged at $t + \delta$ time steps with period δ. We examine each rule to find the percentage of initial conditions that converge within 100 steps and the mean oscillation period of those that converge.

First we consider the two extremities. 23.2% of rules converge for all initial conditions and 49.8% of rules converge for only 2 out 100 initial conditions. Note this is the minimum convergence number in our setting since all rules will converge for the trivial initial conditions with density 0 and density 1. The proof of this is detailed in 6.1.1. This leaves 27.0% or 70708 of the original rules remaining. While the original dataset had a median of 3% convergence, the reduced set of rules present a more even spread with a median of 13%.

Definition 6.1 (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 6.1. *A quiescent life-like CA is at a fixed point or oscillates with period 2.*

Proof. Consider an arbitrary cell c_i in $\underline{\mathbf{0}}$. $\sigma_i(0) = 0$ and $n_i(t) = 0$

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. □

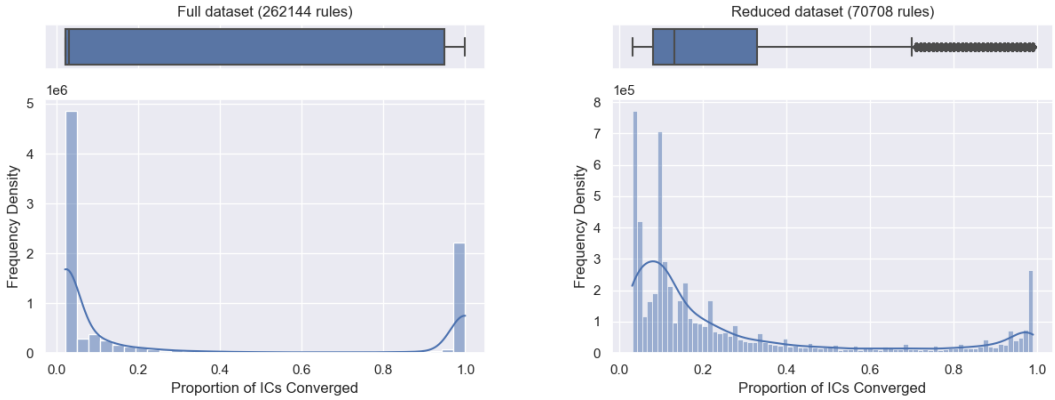


Figure 6.1: Distributions of convergence of full and reduced set of life-like CAs

6.1.2 Similarity Metrics

It is useful to evaluate the manner in which the population converge towards solutions during evolution. Although we use the word convergence to describe this, it is important to note the difference between a CA converging to fixed point or periodic solution and a population converging towards an optimal transition function. To evaluate the convergence of a population, it is useful 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 6.2 (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

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

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 counts mutual presence and mutual absence.

Jaccard Distance d_J, on the other hand, only counts mutual presence. For two binary strings A and B

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

Although this is useful in settings with many false negatives that ought to be ignored, our use case benefits from knowing when a gene is *not present* in both chromosomes as much as knowing if it *is present* in both. For this reason we use the Simple Matching Distance.

6.2 Maze Generation

We begin by evaluating the effectiveness of the 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.

6.2.1 Hyperparameter Tuning

We tune a bias parameter λ to find which bias maximizes $\lambda\hat{p}_\lambda + (1-\lambda)\hat{d}_\lambda$ where $\hat{p}_\lambda$ and $\hat{d}_\lambda$ are the optimal values discovered by the algorithm under a bias of λ.

6.2.2 Roulette vs Truncation Selection

As the population evolves, the variance in fitness decreases. 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 increase the likelihood of premature convergence since genes within incrementally worse solutions that have potential to produce global optima are lost.

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. As

6.3 Life-Like CA

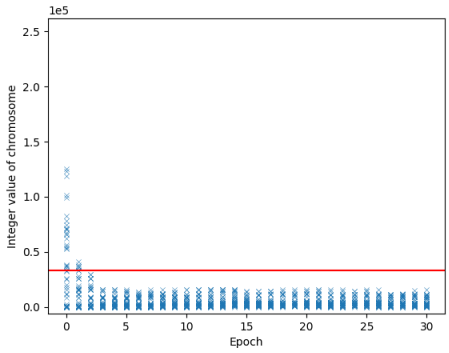
When evaluating the effectiveness of evolutionary techniques at learning life-like CA rules, it is important to contextualise quantitative properties like convergence rate, fitness, and diversity. We begin by performing an exploratory statistical analysis to gather data on the life-like CA rule space. In conjunction with the analyses of Wolfram[36] and Eppstein[8], this will shed light into the characteristics that make a rule easier or harder to predict.

6.3.1 Special Case Evaluation

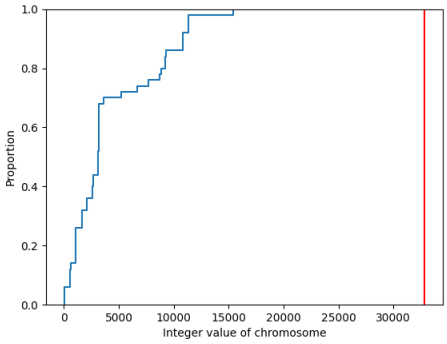
We begin by attempting to learn for a few specific rules to test the efficacy of the learning process. Consider Conway's Life rule B3/S23. In binary this is '0001000000011000000' and as an integer it is 16480. We use the following hyperparameters.

Epochs	30
Population Size	100
Elitism Rate	0.2
Mutation Rate	0.05
Evaluation Steps	10
Minimum Step Size	1
Maximum Step Size	10

These have been obtained by looking at common parameters used in the literature and validating on a few examples to ensure quality of learning while minimizing computation time. A more rigorous hyperparameter tuning is covered in Subsection 6.3.2. Moving forward, we assume this configuration unless otherwise stated. We begin by testing each CA on 20 initial conditions sampled from a distribution uniform on density. This early attempt at learning Life yields poor results.



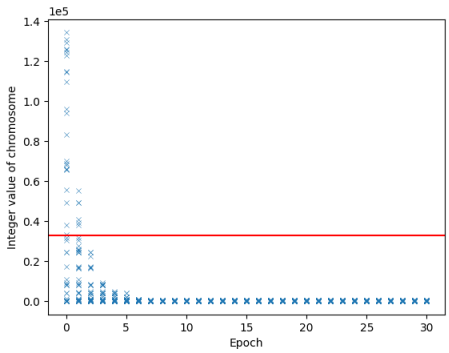
(a) Scatter plot of chromosome values over learning period.



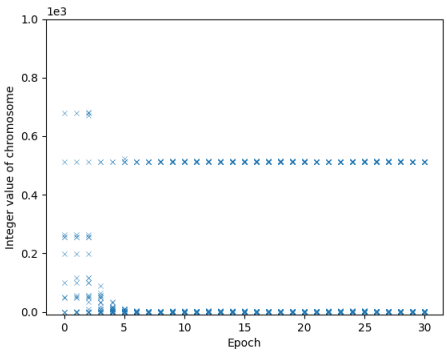
(b) Empirical CDF distribution of chromosome values at epoch 30

Figure 6.2: Learning Life using a genetic algorithm (20ICs). Red line indicates the goal.

Figure 6.3(a) reveals that the algorithm converges to local optima below the global optimum. Figure 6.3(b) reveals that, at epoch 30, all individuals are below the goal with the majority lying around a value of 4000. To get a finer perspective on this, we run the experiment again with 100 initial conditions tested per CA.



(a) Scatter plot of chromosome values learnt with 100 ICs per CA



(b) Same as figure (a) but zoomed into values < 1000

Figure 6.3: Learning Life using a genetic algorithm (100ICs)

Here we see two clear locally optimal regions around 0 and 500 that all individuals

have converged to. Running a value count on the final population reveals there are only 5 unique individuals $\{0, 1, 513, 514, 2\}$ with frequencies $\{18, 14, 9, 5, 4\}$ respectively. The top 4 correspond to the rules B/S, B/S8, B8/S, and B8/S8. Rules with empty birth sets mean that no dead cell can become alive so the system monotonically tends towards quiescence with all cells dead. The case is symmetrical with an empty surival set where the system tends towards the $\underline{\mathbf{1}}$ quiescent state. Similarly, a birth or survival set of $\{8\}$ is extremely unlikely to get triggered. These chromosomes with sparse birth and survival sends tend to promote inactivity in the predicted CA which grants them an edge over volatile chromosomes that quickly wipe themselves out.

6.3.2 Hyperparameter Tuning

We begin by performing some crude hyperparameter testing on the genetic algorithm. 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 and each the loss of each rule is calculated by simulating on 100 random initial conditions. By running the algorithm on populations of size 10 and 100 with $D_{max} \in \{1, 10, 100\}$ for 30 epochs, we find the highest proportion of experiments converging to the precise goal rule within 30 epochs when the population is of size 100 and D_{max} is 10. With this configuration, 32% of goals are precisely learnt. It is clear that population size makes a considerable difference on performance so for all future tests, we maintain a population size of 100.

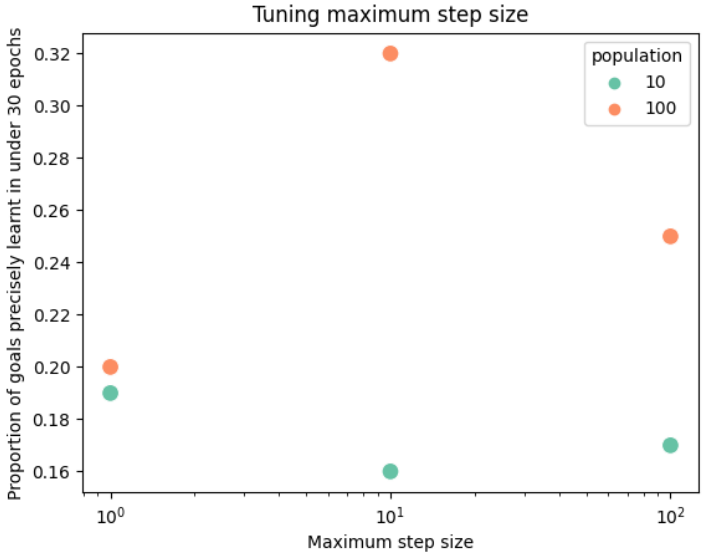


Figure 6.4: Percentage of runs that converge within 30 epochs

In order to reduce training time when testing different hyperparameters, we reduce the size of the training set of initial conditions from 100 to 20. Although this somewhat compromises on the strength of the overall algorithm, it allows us to quickly compare multiple different configurations. One such choice is between loss functions. We consider the performance of the single resolution loss against the multiple resolution loss on a population of 100 over 30 epochs with 20 random initial conditions tested per individual.

6.3.3 Class-Wide Evaluation

6.4 Gray-Scott Models

Chapter 7

Conclusions

Chapter 8

Ethical Considerations

As a relatively mathematical project, there are no major ethical factors to consider here. However, there are still some legal and societal issues that are worth discussing, especially given that technologies falling in the broad field of self-organising systems, distributed systems, and automated systems can be misused.

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

The only area of societal or professional 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 and the academic benefits of researching self-organising cellular automata far outweigh the negligible potential contribution that this research could have to improving malicious swarm robotics systems.

Bibliography

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

[2] 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.

[3] 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.

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

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

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

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] Josep Panadero. Computational science.genetic algorithm.crossover.one point, 2006. URL https://commons.wikimedia.org/wiki/File:Computational.science.Genetic.algorithm.Crossover.One.Point.svg.

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

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

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

[21] 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.

[22] 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.

[23] S Hoya White, A Martín Del Rey, and G Rodríguez Sánchez. Modeling epidemics using cellular automata. *Applied mathematics and computation*, 186(1):193–202, 2007.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] 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.

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

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

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

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

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

[35] 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.

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

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

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

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

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

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

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

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

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

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

[51] 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.

[52] Om Prakash Yadav and Ram Jiwari. A finite element approach for analysis and computational modelling of coupled reaction diffusion models. *Numerical Methods for Partial Differential Equations*, 35(2):830–850, 2019.

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

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

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

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