Developmental Graph Cellular Automata

Riversdale Waldegrave

Doctor of Philosophy

University of York

Computer Science

August 2025

Abstract

During biological morphogenesis, complex and highly specific forms emerge through a developmental process. This often starts with a single cell and can end up creating organisms with trillions of cells. There is no global top-down control of this process but rather it is distributed amongst all the cells, each acting autonomously but coordinating their behaviour through signalling. Abstracting away from biology, this kind of distributed and coordinated development could have many applications, from the self-assembly of physical structures, to the creation of complex computational objects such as neural networks.

Cellular Automata (CA) can be used to explore the capabilities of distributed computation. CA consist of a number of discrete cells connected in some configuration, each with a state and implementing the same update rule, which is applied in parallel at each timestep. This can give rise to complex emergent global behaviour even when the local update rule is comparatively simple. Traditionally, CA are configured on a grid, but it is also possible to structure them as arbitrary graphs. In either case, the cells all exist from the start, in a fixed relationship to each other.

This thesis seeks to adapt CA for constructing graphs via a developmental process. This involves adding, removing and reconfiguring the connections of cells as the system is running. This allows complex graph structures to be created deterministically from the repeated application of a simple rule to a small 'seed graph'. This can be thought of as an abstract model of biological development and has implications in the field of Artificial Life.

The thesis discusses the design of the model, aiming to strike a balance between simplicity and expressive power. We find that Developmental Graph CA can give rise to rich dynamics and complex patterns of growth. The evolvability of growth rules is investigated. It is found that both the dynamical behaviour and the graph structures produced by the developmental process are amenable to evolution. The organisation of the attractor cycles produced by the model is investigated and used as an analogy of self-reproduction or cell differentiation. Finally, a method of harnessing the computational abilities of the system using the Reservoir Computing paradigm is introduced.

Data and code related to this thesis is available at: https://github.com/rvrsdl/DGCA

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References.

Parts of the research described in this thesis have previously published as follows:

- The majority of Chapter 4 has been published as:

  R. Waldegrave, S. Stepney, and M. A. Trefzer, 'Developmental Graph Cellular Automata', in Proceedings of the 2023 Artificial Life Conference, MIT Press, Jul. 2023. doi: https://doi.org/10.1162/isal_a_00658

- The majority of Chapter 5 has been published as:

  R. Waldegrave, S. Stepney, and M. A. Trefzer, 'Exploring the Rich Behaviour of Developmental Graph Cellular Automata', in Proceedings of the 2023 Artificial Life Conference, MIT Press, Jul. 2023. doi: https://doi.org/10.1162/isal_a_00666

- The majority of Chapter 7 has been published as:

  R. Waldegrave, S. Stepney, and M. A. Trefzer, 'Creating Network Motifs with Developmental Graph Cellular Automata', in Proceedings of the 2024 Artificial Life Conference, MIT Press, Jul. 2024. doi: https://doi.org/10.1162/isal_a_00734

- The majority of Chapter 9 has been accepted for presentation at the Artificial Life 2025 conference, to be held in Kyoto in October 2025.

Acknowledgements

I am very grateful to my supervisors, Susan Stepney and Martin Trefzer, for their guidance over the past four years, for innumerable interesting discussions, and for their patience and tenacity in coaxing this thesis into existence. Most of all I am grateful to them for introducing me to so many new ideas and uncovering the mysteries of dynamical systems, self-organisation, and morphogenesis amongst many others. With apologies to Keats, on first looking into Wuensche's *Atlas Of Basin Of Attraction Fields Of One-dimensional Cellular Automata*, I did feel just like "some watcher of the skies when a new planet swims into his ken".

I would also like to thank members of the Non-Standard Computation Group and Bio-Inspired Systems and Technologies Group at York, including David Griffin, Penn Rainford, Andrew Walter, and Chester Wringe, for their discussions, advice and camaraderie.

Finally, I am grateful to my family for their support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes the motivation for the work presented in this thesis, and explains the terms in the title *Developmental Graph Cellular Automata*. It reviews the importance of networks in contemporary computer science (particularly artificial intelligence) and the relevance of networks which develop over time. It discusses how cellular automata can help to implement a biologically inspired model of network growth. It then discusses in more general terms evolutionary developmental biology, a field from which this work draws inspiration. Finally it outlines the overarching aims of the field of artificial life, the field to which this thesis aims to contribute.

1.1 Motivation

1.1.1 Developmental Graphs

We begin by discussing the fist two words of the thesis title: "Developmental Graphs". At a high level, the mathematical formalism of graphs simply describes relationships between entities. This formalism has diverse applications, from modelling social networks to epidemiology to analysis of molecules. Graphs are also of interest as purely abstract mathematical constructs, as attested by the rich field of graph theory. Graphs are ubiquitous in computer science, not only for practical purposes such as describing computer networks, or the abstract syntax trees of computer programs, but also in terms of the many algorithms which deal with graph theoretic problems. Perhaps most obviously in recent years, with the successes of connectionist artificial intelligence (AI), graphs are particularly prominent in the form of artificial neural networks (ANNs).

ANNs take their inspiration from biological neural networks. Making use of a highly simplified model of the behaviour of biological neurons (e.g. McCulloch and Pitts (1943)), ANN units take the weighted sum of their inputs, and apply a nonlinear activation function to generate an output. These units can be connected in networks of varying complexity, from the relatively simple (although sometimes very large) architecture of fully-connected

feed-forward layers to the more biologically similar architectures of recurrent neural networks (RNNs). More advanced ANNs use complex assemblies of modules such as long-short term memory (LSTM) cells (Hochreiter & Schmidhuber, 1997) and transformer modules (Vaswani et al., 2017). However complex the topology of the network, ANNs are generally initialised with random connection weights, which are then adapted through some training process such as backpropagation.

This random initialisation has been lamented by some neuroscientists such as Hiesinger (2021), who point out that connectionist AI took inspiration from the neuroscience at a time when it was generally believed that the wiring of biological brains was largely random. This is true of Minsky's early work in randomly connected neural networks (Minsky, 1952), and perhaps the direction was set even earlier by Turing (1948) on "Unorganised Machines".

The prevalent view at the time that the wiring of brains must be largely random was driven by the observation that there was simply not enough capacity in the genome to encode a complete description of the wiring of billions of neurons with trillions of synapses. In fact, it is now understood that whilst the genome may not encode the final wiring diagram, it does largely determine the growth behaviour of neurons, which in turn results in an extremely intricate and far-from-random connectome. Hiesinger terms this "algorithmic growth" and his book serves as a plea for more cross-fertilisation of ideas between neuroscience and AI. He writes:

> "[...] if ANN developers do not typically think about their networks in terms of self-organizing systems, they think even less about growing them. ANNs from the 1958 perceptron to today's most complicated recurrent networks have this in common: somebody decides on a number of neurons, connects them randomly[1] and then starts training. The brain's neural networks of course do not start with a fixed number of neurons nor are they randomly connected before they start to learn."
>
> — Hiesinger (2021)

Hiesinger's call for connectionist AI to focus on the processes by which networks gradually develop and grow over time is the inspiration for the work presented in this thesis. The aim is not to create a biologically accurate model of neuronal development but rather to create an abstract system which can be used to explore the types of networks that can be created using an algorithmic growth process and the dynamics they can display. This focus is common to much of the work in the field of Artificial Life, which does not seek to examine *life as it is* (as a biologist would) but to gain insights into underlying processes which drive *lifelike* systems. This is further discussed in Section 1.3 below. Although the work presented here is inspired by neurogenesis, the focus of the research is not on the use of networks as ANNs, but rather on the network growth processes themselves. This accounts for the first two words of the thesis title: "Developmental Graphs"

[1] Although the topology of ANNs is usually designed rather than random, it is assumed that Hiesinger is referring to the randomness of the connection weights.

1.1.2 Cellular Automata

The motivation behind the second two words of the thesis title ("Cellular Automata") is now explained. The study of cellular automata (CA) as a type of discrete dynamical system is a well-established field within computer science. The behaviour produced by CA is frequently described as "lifelike", with the most famous implementation being Conway's Game of Life (Gardner, 1970). The patterns produced by this system exhibit lifelike characteristics, such as coherence, movement, growth, interaction. These are examples of *emergent* behaviour, meaning that they are not explicitly programmed but rather emerge from the interaction of lower level rules.

CA are an attractive model for exploring biologically inspired growth processes for several reasons. Firstly, in a typical CA each "cell" follows the same set of rules or program, which is similar to the fact that cells in a biological organism all contain the same genetic code. Just as in biology, CA cells' behaviour is a function of the interaction of their rules with their state and the environment. In biology, the transmission of information between nearby cells (paracrine signalling) or direct neighbours (juxtacrine signalling) is a crucial mechanism for coordinating cellular behaviour including growth and morphogenesis. Typically CAs only pass information between directly neighbouring cells and do not use the long distance signalling that is available to some organisms (via the diffusion of chemical signals and the endocrine system). Nevertheless, complex global behaviour can emerge in CA as a result of purely local information sharing. These bio-similar features of CA make them a useful formalism for examining behaviour where there is no top-down controller but rather a distributed processing capability from which global patterns emerge.

However, there are obvious problems with using standard CAs in an abstract model of biological growth. Generally all the cells already exist in a fixed relation to each other. This is exactly the problem Hiesinger laments in ANNs above. This would seem to make it difficult to use CAs to model organic growth, where the number of cells (and their relation to one another) changes over time. One way around this is to think of the CA cells as the the *space* within which organisms exist. Organisms are then *patterns* of cell states, which can move around in this space and dynamically restructure themselves and even interact with other organisms: this is the metaphor used in Game of Life, for example. However, the structure of the space, whether it is a regular grid or something more exotic such as a Penrose tiling (Owens & Stepney, 2010) or even a continuous space as used in Lenia (Chan, 2019), clearly places constraints on the structure of the organism.

In biology, organisms are obviously also subject to the spatial constraints of our three dimensional universe, but in the case of complex biological structures such as the brain these can sometimes be abstracted away. For example, if we think of neurons as points in space at the centre of each soma, we can say that these points are not only connected to spatially adjacent points, but may also be connected to distant points. This is related to the concept

of *dimension* in graph theory, namely the minimum Euclidean n-space that a graph can be embedded into if every edge has unit length (Erdős et al., 1965). Under this definition, a biological brain is an extremely high dimensional object. In short, it would be difficult to represent anything approaching the complexity of the connectome of the simplest organism as a pattern of states in a conventional grid-based CA.

Graph CA, in which cells are not arranged in a planar grid but can be configured as an arbitrary graph, can express more complex patterns of connectivity. Developmental Graph CA (DGCA), the model introduced in this thesis, also allows the addition and removal of cells during the growth process. This allows us to switch metaphors away from the (pre-existing) cells of the CA representing a fixed and discretised "space" and to a more natural metaphor of the CA cells representing the actual cells of the organism.[2] As with biological cells (and in particular neurons), these DGCA cells can divide, die and reconfigure their connections with other cells, with these dynamics all arising from the shared rules and the interactions between cells. This allows the creation of network structures of arbitrary complexity over time by the repeated application of a developmental rule, just as in Hiesinger's metaphor of the *algorithmic growth* of the brain.

1.2 Evolution and Development

Genetic code is often described as the "blueprint" for living organisms. This description can be misleading as it implies that there is a simple mapping from genotype to phenotype. The genome is clearly not a "blueprint" in the sense that it contains a scaled down model or plan of the organism, or in the sense that it gives interpretable information about how it works. A somewhat more accurate metaphor would be to describe the genome as the program that must be executed in order to produce the organism. A program generally does not resemble the thing it encodes. A recipe for making meringues does not look like a meringue; rather, the meringue is what emerges from following the steps in the recipe. The program metaphor also hints at the idea that the genome must be decoded or unfolded over time, a process which might include intermediate steps. However, this metaphor also has its shortcomings: who or what is executing the program, and on what substrate? A computer program is executed on hardware designed for that purpose and has some centralised controller (for example the operating system). Similarly, a cook follows a recipe to produce a dish. In both cases there are three separate components: the program (or recipe), the thing doing the executing (computer and operating system / cook), and the object that is being acted upon and transformed into the desired output (data / ingredients). In the case of biology, it is more difficult to disentangle these elements. The program does not exist as a separate entity

[2] Alternatively, the addition and removal of nodes could be analogised as the "space" itself expanding and changing topology. This has more in common with the framing used by Wolfram (2020) in which graph rewriting is a reconfiguration of space.

from the organism, but rather is present in every cell. There is no extraneous entity which constructs the organism; rather, the program is executed in parallel throughout the organism, which builds *itself*. In other words, organisms are *autopoietic* rather than *allopoietic*: the system which does the construction is not separate from the system which is constructed (Maturana & Varela, 1980).

The field of evolutionary developmental biology (evo-devo) aims to examine the interaction between evolution of a species (phylogeny) and the development of an individual (ontogeny) (Carroll et al., 2011). It focuses particularly on genes which regulate the development of anatomical structures at different scales such as the homeotic genes. Minor mutations in this class of genes can lead to large scale changes in body plan. This allows the same "building blocks" to be used in different ways. For example, increasing the number of vertebrae in a snake gives them their overall body shape. This can also allow the repurposing of structures for other uses, such as the morphology of legs being reused as antennae in insects (Gould, 2002). Whilst a naive description of Darwinian evolution suggests that random genetic mutations might lead to "random" phenotypic changes, the field of evo-devo highlights that some mutations in fact alter the developmental process which can lead to complex, unexpected and sometimes highly structured changes in the phenotype. Like Hiesinger's *algorithmic growth*, evo-devo emphasises the importance of genetic information "unfolding" over time, resulting in a highly nonlinear mapping between genotype and phenotype.

This idea is one of the inspirations for the work presented in this thesis. It is well known that the extremely simple rules of elementary CA can produce very complex patterns which are computationally irreducible (Wolfram, 2002). In other words there is no quicker way to predict the pattern that will be produced say 1000 timesteps into the future, than to actually run the CA for 1000 timesteps. Similarly, it has been suggested that both the development of an organism through the iterative unfolding of genetic information *and* the evolution of that genetic information may be computationally irreducible (Beckage et al., 2013; Wolfram, 2025). Both evolution and development can be thought of as "bootstrapping" processes, in which the system responds at each timestep to a particular situation, and in responding it creates the situation for the next timestep. During development, this can result in the surprising emergence of complexity even if the rules and the initial state are very simple. The DGCA system presented in this thesis also exhibits these properties, with large and complex graphs developing using simple rules. As with evo-devo, any mutation of the developmental rules can lead to complex and unpredictable consequences for the structures produced.

1.3 Artificial Life

In a critique of the methodology of modern biology, Rosen (1991) argues that since the Newtonian revolution in physics, biology has become "deeply entrenched in a philosophy of

naive reductionism". Influenced by the "billiard ball" view of reality espoused by Newtonian physics, there is an implicit belief that the workings of an organism can be explored by characterising its component parts (previously at the level of the cells, now more commonly at the level of individual molecules). According to Rosen, this misses the crucial point about living organisms: that they are systems, in which the interaction between the parts is often more important than the parts themselves. The appropriate way to study biology is therefore through systems theory, and through the development of an entirely new type of physics he calls "the physics of organised matter."

Rosen claims that biology's attachment to reductionism is due to the fact that it is trying to solve two different problems: the fabrication problem, meaning how the system is constructed, for which deconstruction *is* the appropriate tool; and the physiological problem, meaning how the system works, for which deconstruction is the wrong tool. For the latter, he advocates studying the system by abstracting away from the matter of which it is constructed, just as in mathematics the notion of cardinality is abstracted from quantities of actual objects.

Rosen's ambitions for biology can therefore be summarised as follows:

- take a system-level rather than a component-level view, focusing on interactions rather than parts;
- abstract away from the detail to find general principles;
- take a synthetic rather than deconstructionist approach.

Rosen was writing in 1991, and arguably since then biology has gone much more in the direction he hoped for, particularly with the rise of the so-called "-omics" fields of study, which generally deal with biological phenomena at more of a systems level. Similarly, the field of *systems biology* emphasises the discovery of principles that might apply to wide classes of biological phenomena.

The field of ALife can also help with all of these aims: it pursues a systems-level understanding of biology, and abstracts away from the detail of the particular "hardware" on which it is implemented. Although the inspiration comes from biology, there is often a drive to abstract away much of the detail and complexity of observed biology in order to discover whether radically simpler models can provide insights into the underlying mechanisms of life. This is analogous to the search for the laws of physics, where simplifications can help to clarify what is important: concepts such as point masses and frictionless environments are useful modelling abstractions even though they may not appear in the real world. ALife's aims are not limited to discovering principles of actually extant living systems, however, but also encompass imagining entirely novel approaches which can produce "lifelike" behaviour. It has been described as "an exploration of life not as it is, but as it could be" (Langton, 1989).

Researchers in ALife often aim to create systems which exhibit certain qualitative similarities with biological phenomena, particularly in areas such as the emergence of complexity and open-ended evolution. It is also fundamentally about synthesis rather than analysis: "rather than take living things apart, Artificial Life attempts to put living things together." (Langton, 1996)

These aims mean ALife is less tethered to reality than the natural sciences. In the natural sciences the aim is often to create a *model* of reality which has explanatory and/or predictive power. The quality of a model can be evaluated by its explanatory power as well as its simplicity. A model which can explain more whilst also being more "elegant" is usually considered superior (Deutsch, 1998). Rosen (1991) describes the *modelling relation* by which observed natural phenomena are *encoded* into a formal system which can be run, with the result then being *decoded* into a prediction about the future state of the natural system. Since ALife does not have the "ground truth" of a natural system which it is trying to model, it can be more difficult to evaluate what makes a "good" model. In many cases, ALife research involves the creation of completely abstract systems which are linked to natural systems only via analogy rather than strict encoding and decoding steps. The evaluation of such systems is necessarily subjective: do they produce "interesting" behaviour; or can they provide new insights or intuition into how some lifelike process proceeds? A good example might be the "boids" model developed by Reynolds (1987), which uses very simple rules to create realistic flocking behaviour. This work highlights how seemingly complex collective behaviour such as a murmuration of starlings can emerge from simple individual behaviour. The model also allows experimentation in the sense of testing the macro-level behaviour produced by changes in the micro-level parameters.

In keeping with these aims of the field of ALife, the work presented in this thesis is not intended to be a realistic model of biological neurogenesis (or any other kind of biological development). Rather it is intended as an exploration of the possibility of developing complex networks using bio-inspired growth processes.

1.4 Applications

Whilst this thesis is motivated by the study of growth processes in the abstract, it is worth briefly mentioning possible practical applications of the work. The ability to create, maintain and develop self-organising structures has applications in architecture and engineering (physical structures) as well as in computer systems and AI (virtual structures). Some of the earliest ALife research was funded by NASA with the aim of investigating the viability of self-building and self-maintaining structures on other planets (Levy, 1993). Graphs are a highly expressive abstraction which could be used to represent a wide variety of physical structures, by showing how different physical components are arranged and connected to each other.

Aside from physical structures, graphs have many uses in computer science, most obviously as ANNs. Self-organising ANNs may have advantages in terms of being able to respond to their environment, or being robust to damage or able to self-repair. This may be most relevant on highly constrained hardware: for example it may be advantageous for a circuit implemented on an FPGA to be able to restructure itself if part of the chip is damaged, a concern relevant for chips used on satellites which may be damaged by radiation and cannot easily be replaced. Similarly, autonomous restructuring in response to the nature of the task changing may be useful.

The developmental approach to growing graphs could also be viewed as a compression algorithm, if the growth rule is smaller than the full description of the graph it creates. This could have applications in transmitting graphs over constrained communication channels.

1.5 Thesis Structure

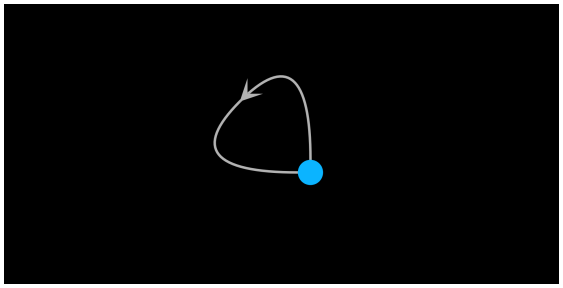
The remainder of this thesis is structured as follows. Chapter 2 is a literature review, outlining relevant work in the fields of dynamical systems, morphogenesis, and genetic algorithms. Chapter 3 sets out the research questions this thesis aims to answer and the novel contribution it makes. Chapters 4 to Chapter 10 form the body of the thesis, presenting the research conducted.

- Chapter 4 introduces a minimal implementation of Developmental Graph Cellular Automata (DGCA) and explores its dynamical behaviour.
- Chapter 6 describes DGCA in more formal terms and introduces a more advanced version of the system which allows whole structures to be replicated more easily.
- Chapter 7 conducts experiments with this new version of the system, exploring the structures that can be created and highlighting that it can produce graphs with desired microstructure.
- Chapter 8 introduces a formalism for analysing the graphs produced at the level of connected components, and introduces a new metaphor in which disconnected graph components are treated as separate “cells”.
- Chapter 9 uses this view to consider cell division and differentiation, focusing on the question of whether the same growth rule can produce different dynamics.
- Chapter 10 presents preliminary work on using the developing graphs for computation, in the framework of Reservoir Computing.

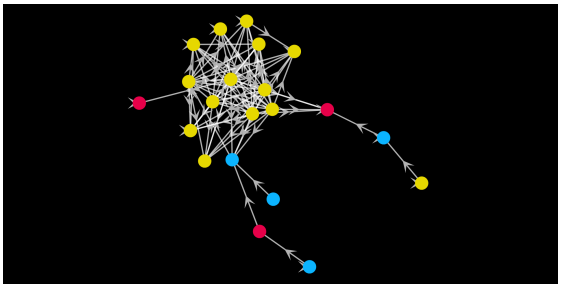
Finally, Chapter 11 reflects on the conclusions of the research and directions in which it could be extended in the future.

1.6 Preview

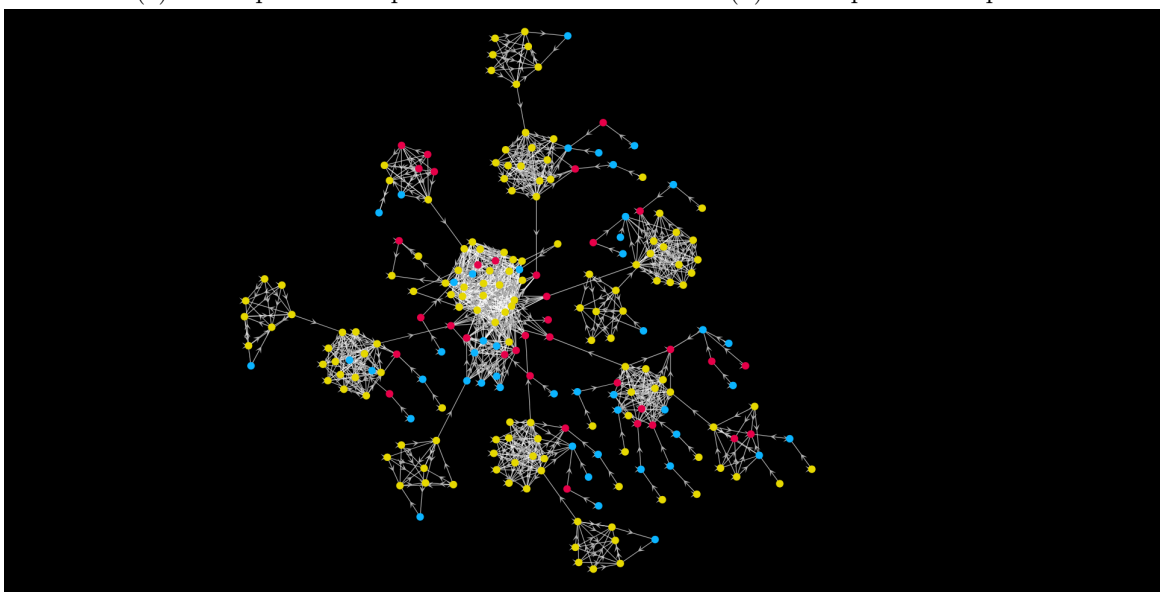
Since the DGCA system is not introduced until Chapter 4, a preview of the graph development it can produce is shown in Figure 1.1 to give the reader a sense of the capabilities of the system. These images are taken from a web-based interactive demonstration of DGCA which we have created. This can be explored at https://rvrsdl.github.io/interactive_dgca.html



(a) Developmental step 0



(b) Developmental step 10



(c) Developmental step 20

Figure 1.1: Preview of DGCA growth using the interactive demonstration at https://rvrsdl.github.io/interactive_dgca.html, with the preset "clusters" growth rule. Starting with a single-node seed-graph (a), the graph grows to 21 nodes by step 10 (b), and 222 nodes by step 20 (c), with a distinctive form of tightly connected clusters linked by single edges. This growth rule was found by manual experimentation.

Chapter 2

Background

2.1 Introduction

This chapter introduces various concepts used in the rest of the thesis, and serves as a literature review. The body of the thesis describes an iterative (or "developmental") algorithm for generating graphs, based on cellular automata, and which is evolvable. The background concepts which need to be introduced in this chapter are therefore: cellular automata, graph generation algorithms, and genetic algorithms, in particular those which use developmental encodings. The literature on each of these topics is extensive, so this review will focus only on those aspects which have a bearing on the work presented in this thesis. Aspects which relate to the field of Artificial Life will be particularly highlighted.

The chapter begins with a tour of selected types of cellular automata (CA), building up from the simplest (elementary CA) to recent developments in graph and neural CA. This is framed within a broader discussion of discrete dynamical systems, which also touches on random boolean networks (RBNs), dynamical systems with dynamical structure ($(DS)^2$) and the computational uses of dynamical systems.

The review then moves on to the literature on graph generation algorithms. Generative grammars are introduced in general before graph grammars are discussed in more detail. Alternative approaches such as Cellular Encoding are also examined.

Finally, relevant aspects of genetic algorithms are outlined, in particular the concept of developmental encoding and evo-devo.

Note that in addition to the works discussed in this chapter, further works are referenced as necessary in the technical chapters of the thesis.

2.2 Discrete Dynamical Systems

Dynamical systems may be continuous or discrete with respect to time and state. Discrete time dynamical systems may be represented using equations of the form

$$\mathbf{v}_{t+1} = f(\mathbf{v}_t) \tag{2.1}$$

where $\mathbf{v} \in \mathbb{R}^N$ in the case of a continuous state system or may take on some set of discrete values in the case of a discrete state system (eg. $\mathbf{v} \in \{0,1\}^N$). When $N > 1$ this equation can be used to represent multidimensional dynamical systems which are spatially discrete (the state of each "unit" in the system is represented by one entry in the vector). However, spatially continuous dynamical systems are also possible, and this class of system comprises many natural phenomena. Dynamical systems which are continuous in time and space are often represented using systems of differential equations. A common example is the Belousov–Zhabotinsky reaction (Belousov, 1959; Zhabotinsky, 1964), which is an instance of a reaction-diffusion system. These chemical reaction systems were investigated by Turing (1952) as an explanation for certain forms of biological morphogenesis.

Spatially discrete dynamical systems are commonly observed where there are multiple (usually identical) units which respond to their neighbours. Examples include:

1. recurrent neural networks, both biological and artificial (Durstewitz et al., 2023);
2. flocks of birds, both natural, as with murmurations of starlings, and artificial as with the "boids" model of Reynolds (1987)
3. cellular automata, which are discussed below.

. Of these, the first two have continuous state (neuron activations / firing rate, and bird position), whereas most cellular automata have discrete state.

This thesis is concerned with dynamical systems which are discrete in terms of time, state and space. Some relevant examples of this type of system are discussed below. Spatially discrete dynamical systems may be structured as grids, as with most CA, or more generically as graphs, as with RBNs. Graphs are the most flexible formalism for spatially discrete dynamical systems as they allow the construction of arbitrary neighbourhood relations without the constraints of Cartesian space. It is for this reason that this thesis focuses on discrete dynamical systems on graphs.

2.3 Cellular Automata

CA consist of a set of discrete units (cells) which are configured in some relationship to each other such that each has a fixed number of neighbours. Each cell may be in one of a finite number of states at each point in time. At each timestep, all cells synchronously update

their state based on a shared update function or rule. The input to the update function may include the cell's own state as well as the states of its neighbours, as shown in Equation 2.2 (which is an elaboration of the generic discrete dynamical system Equation 2.1):

$$v_i(t+1) = f(\{v_j(t) \mid j \in \mathcal{N}(i)\}) \tag{2.2}$$

where $v_i(t)$ is the state of cell i at time t, f is the update function, and $\mathcal{N}$ is the neighbourhood function: $\mathcal{N}(i)$ returns the cells which are neighbours of cell i. The neighbourhood can be defined in various ways, as discussed below.

CA were introduced by von Neumann (1967) in an attempt to create a minimal abstract model of a self-reproducing machine. The idea was partly the result of von Neumann's discussions with Stanisław Ulam who suggested using a 2D grid of cells to implement this system (Wolfram, 2002). In fact, the first published work using the word "automata" to refer to this class of system appears to be Ulam (1950), where they are described as follows:

> "Given is an infinite lattice or graph of points, each with a finite number of connections to certain of its 'neighbors'. Each point is capable of a finite number of 'states'. The states of neighbors at time t_n induce, in a specified manner, the state of the point at time t_{n+1}. This rule of transition is fixed deterministically or, more generally, may involve partly 'random' decisions."
>
> — Ulam (1950)

This is a fairly complete, if rather general, description of CA. This work also mentions that an interesting use of the model would be to "establish the existence of subsystems which are able to multiply". This aim was elaborated in von Neumann's 1952-3 work on a 29-state CA on which he implemented a self-replicating "universal constructor" (not published until von Neumann (1967)). This featured a single line of cells which directed the construction of the machine. This was variously compared to the tape of a Turing machine, or to genetic code. Mutations in this tape allowed the possibility of evolution over generations of replication.

Subsequently, much simpler forms of replication using CA were discovered, such as Langton loops (Langton, 1984), which uses a 7 state CA. The "Evoloop" model added a genetic tape to Langton loops, allowing evolvability (Sayama, 1999).

The early history of CA shows that right from the beginning they were intended to simulate biological self-reproduction and evolvability, aspects which will be explored in this thesis. They have since found many other uses, including physical and biological simulations (Valentim et al., 2023), cryptography (Mariot, 2024), and generative art (Shin, 2016).

The subsections below detail the main types of CA, building up from the simplest elementary CA to later innovations in graph and neural CA.

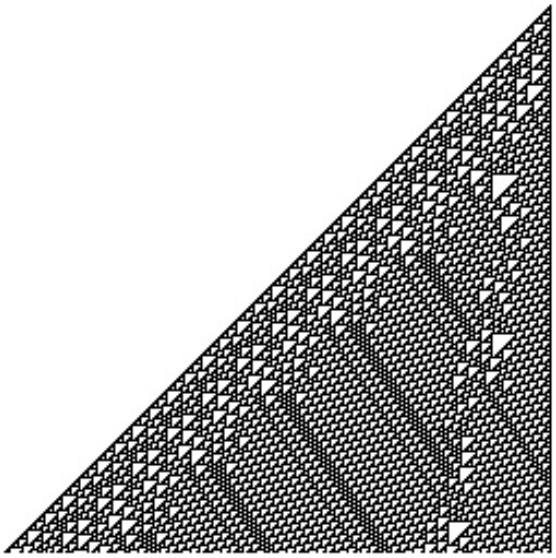


Figure 2.1: A space-time diagram of the behaviour of ECA Rule 110 showing the complex dynamics of Class IV rules. Time proceeds down the vertical axis. Figure reproduced from Wolfram (2002)

2.3.1 Elementary CA

Elementary CA are one dimensional CA with cells arranged in a single line and able to take on only two states. They were conceived as the simplest possible implementation of a CA. Since each cell has two neighbours (plus itself), each of which can be in one of two states, there are eight (2^3) possible patterns that a cell needs to respond to. For each pattern, one of the two states must be chosen as the next state, meaning there are a total of 256 (2^{2^3}) possible rules. Wolfram (2002) conducted a comprehensive investigation into the behaviour produced by each of these rules, and identified four classes of behaviour: those which result in all cells ending up in the same state; those which result in periodic patterns; those which lead to deterministic chaos; and those which result in complex behaviour, allowing the creation of structures which can move through space and time. This fourth class of behaviour has been described as operating at "the edge of chaos". Several of the rules in this class (such as Rule 110, shown in Figure 2.1) have been shown to exhibit dynamics which can be harnessed for Turing complete computation (Cook, 2004). Langton (1990) extended Wolfram's work to parametrise the rules so that these edge of chaos dynamical regimes could be explored more easily. This work highlighted the richness of the dynamics produced by this extremely simple system.

2.3.2 Grid-based CA

The most common type of CA are those based on a two-dimensional grid configuration of cells. As stated above, this was the type of CA originally invented by Ulam and von Neumann in the 1940s. The most famous implementation of a 2D grid-based CA is Conway's Game of

Life (first published in Gardner (1970)). Cells may be in one of two states (termed "alive" or "dead") and use an "outer-totalistic" rule to update their state. Totalistic rules are those in which the update rule uses only the *count* of neighbours in each state, rather than the position of each of these states (ECA rules are therefore not totalistic since they treat left and right-hand neighbours separately). Outer-totalistic rules are those which use the totals for the neighbouring states but consider the central cell's own state separately. The update rule therefore has the form:

$$v_i(t+1) = f(v_i(t), \sum_{j \in \mathcal{N}(i)} v_j(t)) \tag{2.3}$$

Various different neighbourhood definitions may be used with grid-based CA, with the two most common being the Moore neighbourhood of the eight surrounding cells, or the von Neumann neighbourhood of the four cells in the cardinal directions. Numerous studies have been conducted on the effect of using larger or differently defined neighbourhood (eg. the "larger than life" series of CA (Evans, 2003)). However, even the smallest neighbourhood definitions still allow information propagation over long distances, as shown in the study of ECA (Wolfram, 2002).

In Conway's Game of Life, patterns of cell states which maintain coherence over time and/or space are sometimes analogised as organisms. Configurations which cycle through several patterns and remain in the same place indicate that a subspace of the CA has entered an attractor. There are also cycling patterns which move through the grid such as the "glider". However, they are subject to interference from other cell states they encounter, which can destroy, divert or otherwise change the pattern (Beer et al., 2024). Taking advantage of such moving patterns and the interference between them is one way of implementing computation within CA (e.g. Carlini (2020)).

2.3.3 Graph CA

The arrangement of cells is often analogised as the "space" within which organisms (patterns of cell states), can move, grow and interact. It is therefore of interest to observe the effect of different spatial configurations on the dynamics of a system. In addition to square grid, various other configurations have been tried, including grids based on polar coordinates (Schiff, 2011) and Penrose tilings (Owens & Stepney, 2010). These configurations can be thought of as planar graphs, with each cell becoming a vertex in the graph, connected to its neighbours by edges.

Graph Cellular Automata (GCA) take this a step further and allow non-planar graphs. Marr and Huett (2009) uses graphs with random connectivity and uniform degree. The uniform degree is important because each cell uses the same update function or rule and therefore needs the same number of inputs to this function (arity). Whilst grid-based CAs

may use non-totalistic rules by taking advantage of the anisotropy of the grid, this is not possible in a graph with random connectivity because there is no natural and consistent ordering of the neighbours of each cell. For this reason it is generally only possible to use totalistic update rules.[1]

On irregular graphs (ie. with nodes of variable degree), even totalistic rules may not be possible, if the degree varies widely.[2] One way of overcoming this is to use the *proportion* of neighbouring cells in each state as the input to the transition function. This is the approach taken by (Marr & Huett, 2009): they divide the counts of neighbours in each state by the degree of the central node. Since they are concerned with 2-state systems ($v \in \{0, 1\}$), the output of their function is in the set $\{0, 1, +, -\}$ where the symbol $+$ means that the state of the central cell remains unchanged and $-$ means that it flips to the other state. The use of these last two symbols makes their formalism outer-totalistic, since the output of the transition function depends not only on the counts of neighbour states, but also the state of the central cell.

2.3.4 Neural Cellular Automata

The idea of replacing the update rule of a CA with a neural network was introduced by Wulff and Hertz (1992). The update rule is encoded in the weights of a neural network. Tavares et al. (2015) use evolution to adapt these weights (this paper also introduced the term *Neural Cellular Automata* - NCA). Gilpin (2019) represents CA update rules using convolutional neural networks (CNNs), allowing techniques developed for deep learning to be used to train CA rules. Building on this, Mordvintsev et al. (2020, 2022) train the neural networks to produce growing patterns of cells on the CA grid. These patterns grow from a starting ("seed") configuration for a number of timesteps until the pattern is produced, at which point the CA enters a point attractor, so the pattern remains fixed. The growth process can be disrupted by manually changing the state of certain cells. The effect of this disruption can vary - in some cases the CA can regenerate part of a pattern that has been erased, for example.[3] Backpropogation is used to train the weights of the neural network rules which generate these patterns.

Earlier work used genetic programming to evolve artificial circuits for use as the update rule in CA to grow a fixed "French flag" pattern (Miller, 2004). This is a classic problem in developmental biology which seeks to understand how the overall body-plan of an organism (or smaller scale patterning) can develop with no central controller dictating the main spatial axes (Wolpert, 1969).

[1] Unless a system of edge labelling is used, as in GNCA - see below.

[2] Game of Life on a Penrose tiling does use totalistic rules on an irregular graph but the degree of nodes (i.e. number of neighbours of cells) is tightly bounded between 7 and 11 (Owens & Stepney, 2010)

[3] Note that in those papers the terms "growth" and "development" refer to a growing pattern within the fixed space of the CA grid; in the DGCA system introduced in subsequent chapters of this thesis, "growth" means nodes being added to the graph.

2.3.5 Graph Neural Cellular Automata

Grattarola et al. (2021) linked GCA and NCA to create Graph Neural Cellular Automata. These are CAs structured as graphs which use a neural network for their update rule. Nodes in the graph may have variable degree up to some fixed maximum. Whilst the formalism is flexible enough to allow for full anisotropy by the use of edge attributes, it can also be used to generate (outer) totalistic rules. This work also establishes the equivalence between the message-passing step used in Graph Neural Networks (GNNs) with the neighbourhood information transfer used in CA.

The graphs used in GNCA have a fixed size. They mostly also have a fixed topology, although an experiment is shown in which the node state vectors are used as spatial coordinates. The nodes are therefore considered to move on each update step, and the edges are redrawn so that nodes are connected to spatially near neighbours. It is shown that this interpretation allows GNCA to be used to implement the boids model of Reynolds (1987).

Whilst some of the graphs used in GNCA are therefore "dynamic" in the sense that they reconfigure their connections, we do not consider them "developmental" in the sense used in this thesis. We use the word "developmental" to mean that nodes may be added (or removed) and their connections may be reconfigured in precise ways over time as a result of a dynamical process.

2.3.6 Structurally Dynamic CA

Another form of CA which can modify their structure is introduced by Ilachinski and Halpern (1987) as Structurally Dynamic CA (SDCA). This system starts with a two-dimensional grid arrangement of cells. At each timestep, not only is there the usual state update, but also a structural update. This is divided into two parts: a function which checks pairs of nodes which are currently connected and ascertains whether the connection should be removed; and a function which checks *next-nearest neighbours* (nodes which are not connected but have at least one neighbour in common) and decides whether a connection should be added between them. The input to these functions is the state of the two nodes in question and the state of *all* of their neighbours. Totalistic or outer totalistic rules may be used, as well as a new class of "restricted totalistic" rules where the neighbours which are common to both nodes are treated separately from those which are neighbours only of one or other of the nodes. In this case there are three inputs to the function: the aggregate of the states of the two nodes under consideration; the aggregate of the shared neighbour states; and the aggregate of the non-shared neighbour states. The paper uses binary states ($v \in \{0, 1\}$), so the aggregation is simply a sum of states. After running for a number of steps, the original grid structure of the CA is transformed into an irregular graph. Some rules lead to fragmentation of the graph into many disconnected components; some lead to increasingly dense connections; the most interesting are those where the average node degree remains at some stable level, whilst

the graph structure becomes more complex. SDCA are able to create complex structures through the use of only local information. However, the structural update rules are distinct from traditional CA rules in that they work on pairs of cells rather than individual cells. SDCA can be considered an instance of a dynamical system with dynamical structure. Such systems have been examined in detail by Michel (2007) under the name $(DS)^2$, where it is argued they can be used as abstract models of biological development.

2.3.7 Other CA

In addition to the various types of CA mentioned above, there are many other variants. These include:

- Continuous CA, the most famous example being Lenia (Chan, 2019). These use continuous space and state (so are not really "cellular") and replace the update rule with the integral of a convolutional kernel.
- Stochastic CA which use some randomness in the application of update rules. Stochastic ECA were introduced by Baetens et al. (2017). Stochastic CA have been used in modelling tumour growth (Zupanc et al., 2019).
- Lattice Gas CA in which cell states represent the velocities of moving particles. The update rules move the particles and change velocities in the event of collisions. This class of CA can be used to model fluid dynamics, including deriving the Navier–Stokes equations (Wolf-Gladrow, 2000).

2.4 Random Boolean Networks

Random Boolean Networks (RBNs) Kauffman (1969) have some similarities to CAs in that they comprise a network of nodes that update their state at each timestep based on the state of their neighbours. However, unlike CA, RBNs are based on graphs with random connectivity rather than regular lattices. In the original implementation RBNs are still regular in graph-theoretic terms as each node has the same degree (the parameter K in Kauffman's formalism). Again unlike CA, rather than using the same update function at every node, RBNs use a randomly chosen Boolean function at each node to perform the update step. Since every node has the same degree, the update functions in an RBN have a fixed arity, since they all receive input from the same number of connected nodes, just like the update function in a CA. Inputs to each node are taken in a fixed order (set at initialisation), meaning that the update functions do not have to be totalistic, but can be anisotropic.

Kauffman uses RBNs as a model of gene regulatory network (GRN). Each gene (node in the network) can either be "on" or "off" - this is regulated as a function of the gene's inputs.

Since RBNs have a finite state space, they eventually enter an attractor cycle. There may be more than one cycle possible for a given system. Since the system is deterministic, these cycles must be behaviourally isolated; in other words they have separate basins of attraction so are only reachable from separate start states. He notes that the cycles are surprisingly short in length given the huge state spaces involved (e.g. 2^{1000} for a 1000 node system). He finds that the relationship between the number of nodes (genes) and the median attractor cycle length parallels the relationship between the number of genes of an organism and its cell cycle (replication) time. This is used to support his theory that living cells have strong behavioural similarities to randomly connected reaction networks.[4]. Kauffman analogises the different attractor cycles that are possible in an RBN as representing differentiated cell states, or in other words different behaviour modes of the GRN. He further finds a link between the number of nodes / genes and the number of distinct cycles / cell types. By perturbing RBN states (by changing a single node state) he is able to push the system from one attractor into another, which is analogised as the process of cell differentiation. It is noted, however, that $\sim$ 90% of the time the system is robust to such perturbations and returns to its original cycle. In the remaining $\sim$ 10% of cases, it is only possible to move the system to a few other cycles. This observation is compared to the fact that in biology, cells can generally only differentiate into a few other cell types. Note that in RBNs, switching between cycles only takes place as a result of external noise, and cannot happen endogenously. The model does not attempt to explain how cell differentiation is triggered in biological cells.

2.5 Cellular Computing and Biology

Cellular computing is an important concept in the field of ALife. Sipper (1999) defines the three principles of cellular computing: simplicity, vast parallelism and locality. Each individual computational unit or "cell" performs a basic function of the sort that can be defined as a simple rule or a finite state machine. Many of these cells working together in parallel can become computationally powerful. The principle of locality means that each cell only has access to local information, usually from a handful of neighbouring cells, and no cell has an overall view of the system. Since there is no global controller, all system-level behaviours can be said to be emergent properties, since they arise from the local interaction of the many small simple units.

The cellular computing paradigm can unify several disparate techniques under a common framework, including CA, RBNs, ANNs and agent-based modelling (ABM). The framework has also been used with physical substrates, such as biological neurons grown *in vitro* and ensembles of nano-magnets (Heiney et al., 2020). Some of the advantages of cellular com-

[4]This point has implications for origins of life research. Kauffman (1996) argues that proto-cells may have formed as autocatalytic chemical reaction networks enclosed in micelles or liposomes

puting identified by Sipper (1999) are: scalability, since there is no central controller which needs to be connected to all the cells; robustness, due to the parallelism in the system which means that if some cells fail, the computation can often be performed in an alternative way; and the possibility of hierarchical organisation, which can help to decompose a problem.

There are comparisons to be drawn with the emergence of complex coordinated behaviour between biological cells, including morphogenesis. Juxtacrine (chemical) signalling require cells to be in direct contact (like CA), but biology also makes use of long distance signalling (diffusion of chemicals through endocrine system). The latter is thought to be particularly important in setting up chemical gradients which can be used in morphogenesis. However, as work on the French flag problem has shown, large-scale body plans can be created using only local signalling (Miller, 2004; Wolpert, 1969)). The Notch signalling pathway is an example of juxtacrine signalling that is thought to be particularly important for neurogenesis (Luo, 2016). Newer research suggests that in addition to chemical signalling, bio-electrical signals play an important role in morphogenesis, even in non-neural tissues. It is hypothesised that bio-electrical signalling was first used in developmental processes and only later adapted for the nervous system:

> "However, brains did not invent these tricks de novo: they speed-optimized computational properties of slower, "developmental" ionic signaling that cells were using to organize embryogenesis, wound healing, and adaptive physiology long before central nervous systems appeared."
>
> — Sullivan et al. (2016)

2.6 Graph Construction

Having given a brief account of the main types of CA, and a few other examples of spatially discrete dynamical systems, we now move on to a discussion of graph generation algorithms. As stated above, GCA and RBNs use graphs to describe the relationship between nodes. The present work seeks to extend GCA so that they can dynamically reconfigure their structure (including the addition and removal of nodes). This results in a system which can "grow" graphs. In light of this, it is worthwhile to consider other algorithms which can generate graphs. We divide these into two classes: *constructive methods* which create graphs in a single step or through ballistic growth; and *generative methods* which use an iterated process to develop networks over time. The former includes classic algorithms such as the Erdős–Rényi model, whilst the latter includes various generative grammar based approaches. This section discusses the first class of model; the generative methods will be discussed in Section 2.7 along with some relevant developmental models which are not graph-based.

One of the simplest and most widely used methods of constructing random graphs is the Erdős–Rényi model (Erdős & Rényi, 1959). The model has only two parameters, the number of nodes n and the edge probability p, with $p = 0$ meaning there are no edges and

$p = 1$ giving a fully connected graph ($n(n-1)$ edges for an undirected graph, or n^2 edges in the case of a directed graph).

An elaboration of the Erdős—Rényi method is the stochastic block-model approach to graph generation (Holland et al., 1983). This divides the nodes into subgroups, and specifies different edge probabilities within and between each group. This gives slightly more control over the overall structure, and can be used to create graphs with tightly connected regions with fewer edges between them. Such block models can be nested to create hierarchical structuring (Peixoto, 2014b). Block models are used not only to create graphs but also to analyse community structure within given graphs (Peixoto, 2019).

Another elaboration of the Erdős—Rényi model which focusses on creating specific microstructure is the second-order network model of Zhao et al. (2011), which uses additional parameters to modify the probabilities of connections between triplets of nodes. This can be used to vary the occurrence of simple motifs (eg. convergent $A \to B \leftarrow C$ or divergent $A \leftarrow B \to C$ configurations).

Notwithstanding these modifications of the Erdős–Rényi which aim to add high-level structure or specific microstructure, the model often struggles to create realistic degree distributions. Many observed real world graphs such as social networks and citation networks have power-law degree distributions. These are known as scale-free networks. Common algorithm for constructing such networks are the Barabási–Albert model for undirected graphs (Albert & Barabási, 2002) and the Price model for directed graphs (Price, 1965). These models both use a preferential attachment approach, where edges are more likely to be created to nodes which already have a high degree.

Another type of network which is commonly seen in nature is the small-world network (for example in biological neural networks). These are characterised by a short average path length combined with a high clustering coefficient. A popular algorithm for creating such graphs is the Watts and Strogatz (1998) model.

All of the graph construction algorithms mentioned above are probabilistic and make use of a few parameters to create graphs with the required high-level descriptive properties. They cannot be used to create specific graphs (i.e. graphs in a particular isomorphism class), or specific types of microstructure. Whilst algorithms such as preferential attachment have natural analogues (for example in social networks), these models generally appear to be under-parametrised for describing biological morphogenesis. The algorithms in the next section are easier to relate to biological growth, since they unfold over time, responding to the new situation at each timestep.

2.7 Developmental Models

This section covers various developmental algorithms, focussing particularly on their use in graph development.

2.7.1 Rewriting Rules

Rewriting rules such as context-free grammars (CFGs) can be used as a simplified model of biological development. In a CFG system symbols in an initial string are successively replaced by other symbols according to a set of rewriting rules. Certain symbols are designated as terminal and are not rewritten. The process comes to an end when the final string contains only terminal symbols. More generally, it is possible to imagine rewriting systems for a wide variety of symbolic structures (graphs, matrices etc.). These kind of systems are often particularly well-suited to modelling the growth of biological organisms as they can conveniently express both modularity and recursive growth, two things which are typical of biological development. An example of modularity in biological growth is body segmentation in insects, whilst the branching of trees is an example of recursive growth (Dawkins, 2003).

2.7.2 Lindenmayer Systems

Lindenmayer systems (L-systems) are a special form of CFG in which the rewrites are done in parallel rather than by moving along the string linearly (Lindenmayer, 1968). This allows improved modelling of recursive growth since each recursively growing substring can grow at the same time, rather than the first recursive branch having to finish before work is started on the others. This is particularly important in cases where growth does not terminate. Formally, an L-system is defined as the tuple (Σ, Π, ω) where Σ is an alphabet of symbols, Π is the set of production rules, and $\omega \in \Sigma^*$ is the starting string, or "axiom". All symbols in Σ have a rewriting rule associated with them (for example $A \rightarrow BA$) Terminal symbols are simply those symbols which get rewritten as themselves (often for brevity the rewriting rules for terminal symbols are not written out).

Whilst the most basic form of L-systems use context-free rules, versions also exist using context-sensitive rules, which take into account neighbouring symbols. It is therefore possible to view a non-branching context-sensitive L-system as a form of CA with a linear topology, much like ECA, in which the value of each symbol in the string is equivalent to the state of a CA cell. If all the rules result in the replacement of one symbol with exactly one other symbol, then they are effectively identical to CAs since they have a fixed topology. However, if a symbol can be replaced with more than one other symbol (or indeed, can result in the removal of a symbol), then they effectively produce a CA with dynamic structure, where the number of cells can grow or shrink (although the topology always remains linear).

L-systems were originally conceived as a way of modelling the growth of blue-green algae, which grows in linear strands and is therefore easy to represent using strings of symbols. L-systems have since found wide use in simulating growth processes of organic forms in the context of computer graphics (e.g. Prusinkiewicz et al. (2000)). In this case the output of the L-system is usually interpreted as a set of instructions: for example, to a turtle graphics program. Certain symbols may be used to indicate special commands in this program. For

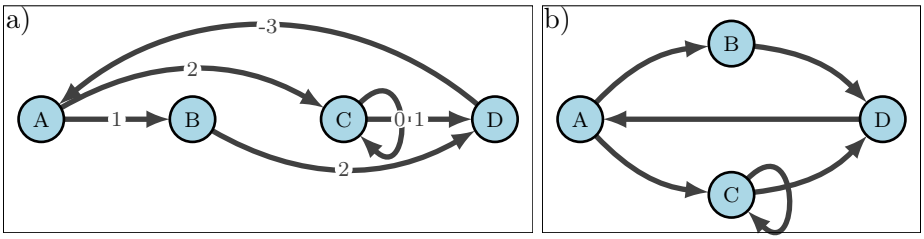


Figure 2.2: Figure recreated from Boers and Sprinkhuizen-Kuyper (2001) showing the graph interpretation of the L-system output string `A12B2C01D-3`. a) shows nodes linearly arranged and the edges labelled with the number of steps forward or back they represent. b) shows a more natural layout for the same graph.

example, brackets may be used to save and restore the turtle state, making it more convenient to draw branching tree-like structures (Floreano & Mattiussi, 2008).

The G2L system of Boers and Sprinkhuizen-Kuyper (2001) interprets L-system strings as graphs, to be used as neural networks. In this system, the alphabet Σ consists of letters and numbers. The letters represent nodes, which are laid out along a line in the order they appear in the string. Any number N following a letter is interpreted as a link from that node to the node N steps to the left or right of that node (depending on the sign of N). Once the links have been made, node positioning (and indeed the node letters) are not important. The process for an example output string is shown in Figure 2.2. Bracketed L-systems can be used to create nested graph structure.

2.7.3 Matrix Rewriting

Kitano (1990) also uses the idea of L-systems for encoding graph development, but applies it to the adjacency matrix. The adjacency matrix starts as a single symbol (like the axiom in a classic L-system), which gets rewritten into a 2×2 matrix of symbols, each of which can also be replaced by a 2×2 matrix of symbols, and so on until the matrix contains only terminal symbols, which must be ones and zeros. Thus rather than operating on a string, the rewriting rules operate on a matrix. An adjacency matrix must always be square, and this property is ensured by constraining every rewriting rule to be of the form:

$$S \rightarrow \begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{2.4}$$

In other words, a single symbol is always replaced with a 2×2 block. It was claimed that this developmental encoding of the connectivity matrix produced faster convergence when applied to encoder-decoder problems, although later analysis disputed this (Siddiqi & Lucas, 1998).

2.7.4 Graph Grammars

A more direct approach to developing graphs is pursued in the field of graph rewriting (Rozenberg, 1997). The replacement rules in these systems directly replace subgraph structures with other subgraphs. An example implementation of graph grammars is the GP2 imperative programming language which uses a rule-based syntax for developing graphs (Plump, 2017). By iteratively applying these rules a graph is developed until it reaches a terminal state.

Tomita et al. (2002) introduce a constrained form of graph grammar which operates on regular planar graphs in which each node has a degree of three. They define three possible structural transformations: replacement of a single node with a triangle of three nodes; removal of a pair of nodes and the link between them; swapping of the connections of two adjacent nodes. All these transformations maintain the condition that each node has three neighbours. They find that this system is able to create complex three dimensional structures.

Kniemeyer et al. (2004, 2008) introduce Relational Growth Grammars which represent both genotype and phenotype using graphs. Genetic mutation and crossover can then be represented within the same graph grammar formalism as phenotypic development. Lobo et al. (2011) combine graph grammars with L-systems, using the L-systems to regulate application of the graph rewriting rules.

Whereas many of the graph grammar systems mentioned here are used as analogues of biological growth, the hypergraph rewriting system of Wolfram (2020) is used as a model of space itself growing. It is claimed that the fundamental laws of physics can be viewed as graph rewriting rules. One feature of graph grammars is that there is often not a unique way of applying the rules at each timestep, and the rules cannot be applied truly in parallel as one subgraph replacement can invalidate other replacements. This is a key difference compared with CAs and L-Systems in which the rules are always applied in parallel. Wolfram takes advantage of the indeterminacy of the rule applications to create a multiway graph of all possible developmental paths. It is claimed that the properties of this graph can be used to derive the theory of special relativity.

2.7.5 Cellular Encoding

Gruau (1994) introduces the concept of cellular encoding for growing ANNs. The idea is that each cell shares a copy of a program, encoded as a grammar tree, which determines its behaviour, including the action of dividing into two cells. Each cell contains a "read-head" which points to the current command in the program tree. Whenever there is command indicating cell division, the program tree divides in two, and the read-heads of the two daughter cells point to the command in the left and right branches respectively. When the read-head of a cell reaches the end of the program, no further development of that cell takes

place. However, there are also loop instructions which may return the read-head to an earlier point in the program.

The cells in this system are not sensitive to their context (adjacent cells) but can nevertheless behave differently because they may follow different branches of the developmental program, creating complex and irregular structure. Unlike a CA, it is therefore not a dynamical system since the update at each timestep does not depend on the current state of the system, but rather follows a predetermined course.

A variant of cellular encoding, in which development is focused on edges rather than nodes is investigated by Luke (2000). Kodjabachian and Meyer (1998) propose a cellular encoding based system in which cells are positioned on a two-dimensional substrate and where the program tree contains commands allowing them to move and grow links to nearby cells.

Aside from the distinctive use of a grammar tree in Gruau's cellular encoding, there have been other attempts to use a "microprogram" embedded within each cell to control cell behaviour and ultimately cause a network to grow. Miller and Thomson (2003) use Cartesian Genetic Programming (CGP) to create a microprogram which is the same in each cell. The input to this microprogram is a description of the cell's position, function and connections; the output is its new connections, new function and whether it should create a new cell. Miller (2021) elaborates on this by using separate microprograms for different parts of the neuron cells: the soma and the dendrites. The biological inspiration for such schemes is clear: neurons in the brain self-organise into a network based on their own rules of behaviour or "microprograms" without any central organising principle. By using information about their local neighbourhood, they are able to create complex structure.

2.8 Genetic Algorithms

This thesis does not investigate Genetic Algorithms (GAs) in their own right, but uses them simply as tools to search over developmental programs. However, there are various GAs which focus particularly on the creation of graph structures which will be summarised here.

Genetic Algorithms (GAs) are an evolutionary search technique which can be used in a wide range of domains, from the design of computer programs and electronic circuits to the creation of artistic images. GAs use a population of candidate solutions, which are usually randomly initialised. At each generation, each candidate is evaluated against the desired outcome using a fitness function. The fittest individuals are used to initialise the next generation using mutation and/or crossover operations. The process continues until one of the solutions reaches the required fitness, or until no further improvement is observed.

In many cases GAs are used with a fixed fitness function. However, a more open-ended approach which is employed in some of the experiments in this thesis is novelty search (Lehman & Stanley, 2011). Here, the fitness of an individual is judged solely with reference

to how dissimilar it is to other previously seen solutions. This is therefore a dynamic fitness function which changes over time and is contingent on other individuals. This has some similarities with natural evolution, in which fitness is not static but changes due to changing environments and ecological niches.

The choice of genetic encoding may have significant consequences for the evolvability of the system. The simplest form is direct encoding, where there is a one-to-one correspondence between the genotype and the phenotype. Such encodings must take care that crossover and mutation operations do not destroy necessary structure in the phenotype. For example, it would be difficult to evolve an image directly by mutating and swapping pixels. For this reason, indirect encodings are often used, where the genome must be *interpreted* to generate the finished product. Indirect encodings can also allow a degree of compression in the genetic representation. A special case of indirect encoding is a developmental encoding, in which the information in the genome must be unfolded in an iterative (often dynamical process). This is the type of encoding most similar to biology. Examples of these three types of genetic encoding are given below.

2.8.1 Direct Encodings

Direct evolution of programs or mathematical expression can be achieved by swapping syntactically complete branches of the abstract syntax tree (AST) of a program or of a mathematical expression. This was introduced as Genetic Programming by Koza (1992). A later elaboration, Grammatical Evolution, guarantees the generation of syntactically correct programmes given a BNF grammar (O'Neill & Ryan, 2003).

The Cartesian Genetic Programming (CGP) introduced by Miller (2003) is another direct encoding approach. The name derives from the fact that nodes in the network are laid out in a two-dimensional grid. It was originally developed for the purpose of evolving electronic circuits, but it can in fact be used to evolve any function which can be represented as a composed graph of sub-functions. An interesting feature of CGP is that not all of the network structure specified in the genome will necessarily be connected to output nodes, and will therefore have no functional impact. The program (phenotype) encoded by a CGP genome is simply the part of the network which connects inputs to outputs. The phenotype can therefore end up smaller than the genotype, but cannot be any bigger. In experiments with large networks it was found that the percentage of redundant nodes was often as high as 95% (Miller & Smith, 2006). This stands in marked contrast to some of the indirect and developmental genetic encodings explored below, in which the mapping from genotype to phenotype often involves a form of decompression, resulting in a phenotype significantly *larger* than the genotype. However, the idea of non-coding regions in the genome does have biological plausibility and may bring advantages for evolvability as it allows "neutral drift." This is the phenomenon of the genome altering (via mutations) with no corresponding

effect on the phenotype. This allows multiple mutations to build up in the genotype before being expressed. Experiments have shown that neutral drift can be very important for CGP evolution.

The Neuro-Evolution of Augmenting Topologies (NEAT) algorithm introduced in Stanley and Miikkulainen (2002) is a direct encoding approach to evolving neural networks. It is a type of TWEANN (topology and weight evolving ANN) which aims to overcome the problem that network crossover operations between two fit individuals can destroy the useful structure in both of them, resulting in an unfit offspring. They refer to this as the "competing conventions" problem, that the same functionality can be represented in topologically distinct ways in a network. They introduce a system of historical markers and speciation which avoids this problem by aligning equivalent parts of parent networks before crossover. The network grows as new nodes are added via mutation. However this is thought of not as a developmental process but as complexification through evolution. Since NEAT uses a direct encoding, the graphs (genomes) acted upon by evolution can be used immediately as neural networks with no interpretation step.

Chatzidimitriou and Mitkas, 2010 use a NEAT-inspired approach to evolve network topologies for Reservoir Computing (see Section 2.9). They directly evolve reservoir connectivity using a matrix representation, but use the idea of historical markings from NEAT to align the matrices before the crossover operation.

2.8.2 Indirect Encodings

Evolutionary approaches have often been used to search for graph structures with good computational properties (ANNs, ESNs etc). However, as the size of these networks is scaled up, the search space can become unmanageably large. One approach is to combine evolution with an indirect encoding. This can result in a genetic representation that is highly compressed compared to the final structure it gives rise to. This reduces the search space, and whilst it constrains structures that can be found, this constraint can result in modularity or repeating motifs, which may be advantageous properties for certain tasks. Modular network structure may improve performance on tasks where the problem domain also contains regularities (overt or hidden) such as image processing (Clune et al., 2011). Furthermore, the evidence from neuroscience suggests that the brain contains many repeated network motifs and structures, such as the cortical minicolumns which are thought by some to be the fundamental unit of computation in the neocortex (Hawkins, 2021; Mountcastle, 1997).

HyperNEAT (Stanley et al., 2009) evolves a small neural network using NEAT, which is then "queried" to discover the connection weight between every pair of nodes in a larger network. The inputs to the small network are the coordinates of each pair of nodes (x_1, y_1, x_2, y_2) and the output is a single value indicating the weight that should be used between that pair

of nodes. This results in ANNs with structural regularities that can usefully mirror geometric properties in the problem domain (for example, the layout of a chess board, or relative pixel locations in an image).

2.8.3 Developmental Encodings

The highly non-linear mapping between genotype and phenotype have been described as a "black box" (Hall, 2003). This mapping is powered by a developmental process and as such, any minor changes in the developmental "rules" may have unpredictable effects on the final organism. In nature, this may enhance evolvability, since small mutations may result in the phenotype "jumping" to a completely different place in the fitness landscape. This also explains the surprising fact that organisms with very different morphology may share a high percentage of their genetic material, with key mutations occurring in the genes which regulate development (Carroll et al., 2011).

Many of the developmental models in Section 2.7 can be combined with evolution to form evo-devo models. In this case, it is the developmental *rules* which are evolved. Producing the phenotype then requires "running" the developmental program. This is true of the G2L system (Boers & Sprinkhuizen-Kuyper, 2001) and the matrix rewriting system (Kitano, 1990) outlined above, where evolutionary search can be used to find developmental rule-sets which produce the desired behaviour. Similarly, Atkinson et al. (2018, 2021) make use of the GP2 graph grammar language in an evolutionary context by evolving the graph rewriting rules. Unlike the systems mentioned above, there is no change in representation from the developmental rules to the finished phenotype: a graph representation is used throughout. There is therefore no "interpretation step" which Atkinson et al. (2018) argue may reduce biases in the evolutionary search process.

Several models use the running of a CA as an analogy of development. Nichele et al. (2017) use NEAT to evolve the transition function of a 2D CA. Running the CA is then used to develop various "flag" target patterns from a starting configuration of a single cell which seeds the pattern. Najarro et al. (2022) use a 3D grid-based CA with real valued states to develop a deep feedforward network for use in reinforcement learning tasks. The CA is run for a fixed number of steps and the final cell values are interpreted as weights in the layers of a feedforward network. For example if the 3D CA grid has size $8 \times 8 \times 20$ then it could represent the weights of 20 fully connected feedforward layers with 8 nodes in each layer. The CA uses a neural update rule, which is evolved so that the networks which are produced (after the development step of the CA running) perform well on the task.

Other evo-devo approaches to growing ANNs with particular properties include Hintze et al. (2020), which uses a gene regulatory network (GRN) to create a gene expression vector which is then translated into RNN weights. Najarro et al. (2023) introduces Neural Development Programs, an elaboration of GNCA which can be used to grow neural networks.

In addition to the usual use of an ANN to update node states, a second ANN determines if a node should replicate. The development rules can be adapted to produce performant ANNs either through evolution or through gradient descent (only in the case of feedforward ANNs).

2.9 Computing with Dynamical Systems

As mentioned, CA and other dynamical systems can be used for computation. It has been widely noted that "criticality" in these systems can enhance computational ability. This means that the system is poised between ordered and chaotic behaviour, in a region sometimes known as the "edge of chaos" (Langton, 1990; Teuscher, 2022). Certain types of natural and artificial systems have the ability to push themselves towards this regime, in a phenomenon known as "self-organised criticality" (Heiney et al., 2019; Kagaya et al., 2022). In the case of biological neural networks, criticality is often indicated by a power-law distribution of neuronal avalanche size and duration (Beggs & Plenz, 2003)

Pontes-Filho et al. (2020) unify many of the discrete dynamical systems mentioned here (CA, RBNs, ESNs) in a single framework and evolve them towards criticality, targeting a power-law distribution of avalanche sizes and durations.

The field of Reservoir Computing (RC) (Jaeger, 2001; Maass et al., 2002) has shown that *any* dynamical system with rich non-linear dynamics and fading memory can be used for computation. Whilst many implementations of RC use physical materials as the dynamical system, the original Echo State Network (ESN) implementation used fixed-weight randomly connected RNNs represented as graphs (Jaeger, 2001). The RC framework has allowed systems exhibiting this rich edge of chaos behaviour to be used as a computational substrate, including some types of CA (Nichele & Molund, 2017; Uragami et al., 2022).

Chapter 3

Research Questions

3.1 Aims

The aim of this thesis is to introduce and evaluate a method of growing networks over time. There are many existing algorithms for creating graphs (both constructive and generative approaches, as discussed in Sections 2.6 & 2.7). The present work aims to contribute to the field in the following areas:

- biological inspiration;
- dynamical development;
- evolvability;
- structural expressiveness;
- multiple layers of abstraction;

These are explained below.

3.1.1 Biological Inspiration

Biological growth development does not have a central controller but rather emerges from the interaction of cell-level actions. Local signalling between cells helps coordinate development. Inspired by this, the present work seeks to introduce a method of growing graphs based on distributed node-level decisions, without a central controller. Only local information should be used and communicated between cells. By adhering to these bio-inspired principles, it is hoped that we can achieve emergent development. As with many emergent phenomena, there may be a highly nonlinear mapping between the micro behaviour (the node-level growth rule) and the macro effects (the resultant graph structure).

3.1.2 Dynamical Development

Constructive approaches to graph generation as discussed in Section 2.6 are generally not dynamical. Even if they are multi-step processes, such as the Barabási–Albert model, the nature of each constructive step is the same. There is no sense in which an external perturbation could push the system into a different basin of attraction, resulting in a qualitatively different graph structure.

Generative approaches, such as the L-systems, graph grammars and cellular encoding described in Section 2.7 *can* be dynamical in that they can change dependent on the current state of the system. However, this is not usually a primary concern when these models are employed. When L-systems are used to grow organic looking trees in a computer graphics setting, for example, the focus is on the final object, not on the developmental process. The present research, by contrast, is concerned not only with the created structures but also with the dynamical process which creates them. This is also true of fields such as Dynamical Systems with Dynamical Structure, mentioned in Section 2.3.6. A feature of dynamical development is that it may enter an attractor, meaning that development comes to a halt of its own accord. Furthermore, if the system is perturbed at any point during development, it may completely alter the onward growth trajectory. Alternatively, the system may be robust to certain perturbations, quickly recovering its previous trajectory. Both phenomena occur in biological development. Sensitivity to perturbations can be used to integrate environmental feedback, whilst robustness can be ensure consistent developmental outcomes.

3.1.3 Deterministic

Many graph generation approaches are stochastic. Parameters may be used to tune the the high level graph statistics: for example, the edge probability in the Erdős–Rényi model or node sampling number in the Barabási–Albert model. However, the specific microstructure is left to chance. Whilst these models are useful for generating ensembles of graphs with similar properties, such as degree distribution, they are insufficiently parametrised to guarantee the creation of graphs in a specific isomorphism class. In many use-cases this is desirable, as the researcher may wish to generate an *example* of a small-world graph, for instance, rather than a specific graph. Graph grammars are also usually not deterministic as there can be multiple non-compatible matches for the left-hand side of subgraph replacement rules, and an arbitrary choice must be made about which rules should be applied.

However, it is easy to imagine situations where the specificity of a deterministic approach would be crucial. For example, this could apply in the creation of self-assembling physical structures, such as buildings, or the unfolding of a computational substrate such as a neural network.

Determinism can also be useful in creating a model of biological development. Whilst biology obviously does make use of randomness, many aspects of biological growth are strongly

determined (leaf shape of an oak tree, number of cervical vertebrae in a mammal etc.). An algorithm which deterministically produces a particular structure can also be viewed as a form of compression, if the description of the algorithm is shorter than a full description of the structure.

3.1.4 Evolvability

Grammar-based approaches to creating graphs (whether using graph-interpretations of L-systems, or graph grammars directly), can be difficult to evolve. The "genome" consists of the rewriting rules used, which are usually expressed symbolically. It is generally not possible to change such rules "smoothly": replacement of one rule by another can lead to a step change in behaviour. Although developmental encodings often lead to a highly nonlinear mapping between genotype and phenotype, this does not preclude evolvability (as evidenced by nature). It is an aim of the present work to achieve good evolvability both in terms of the dynamics of growth and in terms of the final graph structures produced. This requires the developmental rules to be expressive enough to produce a wide variety of behaviour, and also for an evolutionary process using standard mutation and crossover operators to be able to achieve the desired behaviour.

3.1.5 Structural Expressiveness

As already mentioned, it desirable for a graph generation algorithm to be able to create a wide variety of different graphs, not just in terms of high level descriptions but also specific microstructure.

3.2 Hypotheses

We hypothesise that using a Graph CA combined with a development process which can add and remove nodes can achieve the aims listed above. We term this system a Developmental Graph CA (DGCA). CA use local information and local-decision making to produce emergent behaviour, fulfilling the first aim. They also produce dynamical behaviour which can be described in terms of transients and attractors. It is hoped that this will also be true of the developmental behaviour of DGCA. CA are also deterministic (in standard implementations), with the result determined solely by the starting state and the update rule. We further hypothesise that:

1. DGCA with the right representation have high evolvability leading to a wide variety of phenotypes.
2. Evolution of DGCA can create wide variety of growth dynamics and structure
3. Targeted evolution of DGCA can be directed towards different structural expression.

3.3 Research Questions

This thesis seeks to answer the following research questions:

1. Can we implement a version of Graph CA with dynamical structure?
2. Can we evolve the dynamics of development *and* the graph structures which result?
3. Can the developmental dynamics produce analogues of self-reproduction and other biological phenomena?
4. How do the growth dynamics impact the structures that are produced? Can multiple different structures be produced by a single growth rule?
5. Can we use an evo-devo process to produce a useful computational substrate?

3.4 Methodology

In order to design a model with the properties described above and investigate the research questions, the following approach is taken.

- Introduce a basic DGCA model and identify different types of dynamical behaviour it produces, with a focus on whether development can "naturally" halt by entering an attractor. This work is presented in Chapter 4.
- Investigate the evolvability of these developmental dynamics. This work is presented in Chapter 5.
- Investigate the structural expressiveness of the model, focussing on graph microstructure rather than high-level descriptive statistics such as degree distribution. A more sophisticated model with greater structural expressiveness is introduced in Chapter 6, along with a proof that it can be used to create *any* graph isomorphism class (provided it is sufficiently parametrised).
- Investigate the evolvability of the developmental program in terms of the structures it creates. Can the growth rules be evolved to create specific graph microstructure? This work is presented in Chapter 7.
- Investigate the biologically analogous behaviour produced by the system, such as self-reproduction. A formalism for describing these higher-level dynamics is introduced in Chapter 8.
- Applying an analogy of cell division and differentiation, investigate whether the dynamics of the system can be harnessed to create multiple graphs with different structures, using a single developmental rule. This work is presented in Chapter 9.

- Investigate whether the evo-devo process can be used to create good computational substrates, under the Reservoir Computing paradigm. This work is presented in Chapter 10.

3.5 Structure of the Technical Chapters

The original work of the technical chapters of the thesis is outlined above. Note that Chapters 4 to 9 are arranged in pairs, while Chapter 10 is standalone:

- Chapters 4 & 5 are concerned with the dynamical behaviour of the system, with the first being more theoretical (introducing the system and its behaviour) and the second being more experimental (on the evolvability of the behaviour).
- Chapters 6 & 7 are concerned with the structures the system can produce. Once again, the first is more theoretical, introducing the more advanced model which is necessary in order to be able to produce any graph structure, and the second is more experimental, evolving the growth rule to produce specific structures.
- Chapters 8 & 9 again form a theoretical / experimental pair, concerned with a higher-level abstraction of the system's behaviour.
- Chapter 10 is a standalone chapter which can be considered supplementary. Whereas the main body of the thesis is concerned with the capabilities and behaviour of DGCA *per se*, Chapter 10 is concerned with its use for computation.

Chapter 4

Developmental Graph Cellular Automata Behaviour

4.1 Introduction

This chapter introduces the basic model of Developmental Graph Cellular Automata (DGCA), which encodes developmental rules for graphs in the weights of a small "internal" neural network which determines each cell's behaviour. Exploratory work is then conducted to assess the different types of behaviour which are produced by the system. Particular attention is paid to cases where the system enters an attractor and so development comes to a halt.

The chapter begins with a discussion of how developmental systems come to a halt, both in Artificial Life models and in natural biology. This recapitulates the main research question for this chapter, namely whether the developmental system introduced here can "naturally" reach an end point of development, even though the state space is unbounded. The model of DGCA is then introduced, with a worked example to illustrate one update step. Comparisons are drawn with conventional and neural CAs, and the biological inspiration from signal transduction networks is highlighted. An experiment is then conducted, using random search over developmental rules to evaluate the frequency of five different classes of growth behaviour. These results show that the system enters attractors surprisingly frequently, inviting analogies with endogenously terminating biological growth processes.

4.2 Halting Growth

One question that immediately arises when implementing a developmental process is how to decide when development has finished. Many models of developmental processes do not naturally terminate but are rather run for a fixed number of timesteps (eg. Hintze et al. (2020) and Najarro et al. (2023)). However, the system introduced here has the property that growth often (but not always) comes to a halt of its own accord, as an attractor state

is reached.

Taking inspiration from nature, it is noted that in the simplest groupings of single-celled organisms, such as bacterial biofilms, growth is limited only by availability of nutrients. The same is largely true of colonial organisms such as corals. Plant growth is primarily limited by availability of nutrients and physical constraints such as the strength required to stay upright. Nevertheless, individual plant organs such as leaves, flowers, and seeds reach fixed sizes and stop growing. In the animal kingdom growth is limited both for organs and for the whole organism: in most cases it comes to a halt when an organism reaches maturity. It is clear that the limits of growth are in some sense genetically encoded, since organism body size is to some extent heritable. However, if we think of the process of development as the iterative decoding of genetic information (see Section 2.8.3), then how does this process "know" when to come to a halt?

One possible model is that of individual cells reaching terminal states, beyond which they do not divide, rather like the terminal symbols in an L-System (see Section 2.7.2). The Cellular Encoding model of Gruau et al. (1996) also implements a the "distributed decision" to stop growing, via the termination of the individual cells' growth programs (see Section 2.7.5).

Another approach to terminating growth is via global signalling. Although in the systems under discussion there is no central controller, it is possible for chemical signals to be produced by one or more cells, which can then diffuse throughout the system and prevent further cell division globally. This notion of the diffusion of morphogens is used in models such as that of Wolpert (1969).

However, neither the purely local cell-based termination of growth, nor the global signalling approach seem to capture the full complexities of growth halting in biological organisms, although they may be part of the mechanism. In a review paper on "Mechanisms Limiting Body Growth in Mammals", Lui and Baron (2011) write:

> "Although the mechanisms responsible for limiting organ growth appear to be local, rather than systemic, they may not be cell autonomous but rather may involve local interactions among cells, such as paracrine signals."

This points to growth termination arising as an emergent effect from the interaction of simple system components. Supporting this view is the fact that growth can sometimes restart as a response to injury, in the form of repair or regeneration. It seems unlikely that either the fully local approach ("terminal symbols") or the fully global approach (system-wide morphogens) to growth termination would be able to restart growth as a response to injury. A study on regeneration in Planaria worms concluded that local cell signalling was at least partly responsible (Wenemoser & Reddien, 2010).

In biology therefore, it seems that the halting (and possible restarting) of growth is largely due to a combination of cell-level decisions and local signalling between cells. This

points to CA being a useful modelling paradigm, since they use exactly this mechanism. In the terminology of dynamical systems, when a CA reaches a state in which no more changes take place, or cycles between a finite number of states, it has reached an attractor. Point or cyclic attractors in CA seem a natural metaphor for growth termination. They also offer the possibility of restarting growth, by perturbing the CA out of its attractor. Some perturbations may lead the overall state of the system to return to the same attractor after some transient steps. Others may push the system into a different basin of attraction so that it comes to rest in a completely different state. This second phenomenon brings to mind cases of biological metamorphosis where some stimulus (exogenous or endogenous) can cause an organism to completely restructure itself, as with holometabolous insects, or to restructure an organ, as with the brain restructuring that takes place in some birds in preparation for seasonal migration (Yaskin, 2011).

In many standard grid or graph based CA, there are a finite number of cells and cell states, and therefore a finite overall state space. Any dynamical system with a finite state space will eventually end up back in the same state; if it is deterministic, this will mean it has entered an attractor. In this class of CA it is therefore trivially true that an attractor will eventually be reached[1], making it an uninteresting model for biological growth termination. In biology, the state space is infinite, in the sense that there is no hard limit on the number of cells an organism can contain (beyond the bio-physical limits of self-supporting structure and nutrient provision). Biological cells are likewise not limited to a finite number of "states". The DGCA system introduced here does use a finite number of cell states, but cells may be added or removed with no limit, meaning that the overall state space is infinite. This allows DGCA to be used in investigation of biological growth. Not all DGCA instances will enter an attractor: some will grow infinitely like the simpler classes of organism described above. Note that in this model there is no notion of nutrients or energy being required for growth. Whilst this is biologically unrealistic, it allows the research to focus on the question of endogenously driven termination of development, as an emergent effect of the system dynamics.

4.3 The DGCA Model

This section introduces the DGCA model and shows a worked example of a single update step.

4.3.1 Directed Graphs

A design decision was taken to use directed rather than undirected graphs in this system. This was partly due to the fact that the inspiration for this work came from neurogenesis.

[1] Admittedly, finite CA may have enormous state spaces, so reaching an attractor *within a given time* is still noteworthy.

Neural networks are directed graphs in the sense that electrical spikes flow in one direction through neurons. The choice to use directed graphs was also motivated by the possibility of using the developed graphs as RNNs (see Chapter 10). However, it is important to note that during the development process, the edge directions do *not* indicate the flow of information into node transition functions, but rather are used to provide limited anisotropy in the node neighbourhood: the incoming and outgoing edges define two “types" of neighbours (not to be confused with the states of the neighbours).

To formalise this, consider a set of vertices V, and a set of edges $E = V \times V$ that are all the ordered pairs of nodes, with the first in the pair being the source node of the edge and the second the target node. Each vertex has a state, drawn from a finite set of states Σ (which in the graph theory literature would be called non-unique labels). There is a mapping σ from nodes to states:

$$\sigma : V \to \Sigma \tag{4.1}$$

A single graph g is then represented by the tuple

$$g = (V_g, E_g, \sigma_g) \tag{4.2}$$

where $V_g \subseteq V$, $E_g \subseteq (V_g \times V_g)$, $\sigma_g \subseteq \sigma$, and the domain of σ_g is V_g. All the graphs produced by a particular run of the system use node states from the same set Σ.

In the remainder of the thesis the terms vertex and node are used interchangeably. The term *node state* indicates the state of a particular node at a point in time (eg. the state of node u in graph g is $\sigma_g(u) \in \Sigma$).

4.3.2 Requirements

The system presented here is based on irregular networks in which nodes can have different degrees. RBNs use graphs in which every node has the same degree (Kauffman, 1969). This means that the update functions in a RBN have a fixed arity, since they all receive input from the same number of connected nodes, just like the update function in a CA. One of the requirements of the present system is that it should have a single update function that is used at all nodes, but that each node must be able to have variable degree. Since we are dealing with growing networks, a single node may change its degree between timesteps: it may gain or lose connections to other nodes. The differences between CAs, RBNs, GCAs, context-sensitive L-systems and the system presented here are summarised in Table 4.1.

With an irregular and changing graph, there is no obvious way to order the inputs. In a CA the regular grid provides a natural way to order the inputs: for example in a two-dimensional CA using a von Neumann neighbourhood, the inputs can be supplied to the update function in the order `[North, East, South, West, Centre]`. Even in an RBN, where there is not a grid topology to provide a natural order to the inputs at each node,

System	Connectivity	Update	Structure
Cellular Automaton	lattice	homogeneous	fixed
Random Boolean Network	regular graph	heterogeneous	fixed
Graph Cellular Automaton	regular graph	homogeneous	fixed
Elementary Cellular Automaton	linear	homogeneous	fixed
Context-sensitive L-System	linear	homogeneous	dynamic
DGCA	irregular graph	homogeneous	dynamic

Table 4.1: Examples of spatially discrete dynamical systems with different types of connectivity, update function (homogeneous, hom, and heterogeneous, het) and structure.

an arbitrary order can be defined at initialisation. Since the topology remains fixed, this ordering can be used throughout. However, if one or more connections may be added or removed at each timestep, any arbitrary ordering of inputs to the transition function at each node would have to be constantly redefined. This means that the update function used must be isotropic. The varying node degree means that it must be (outer) totalistic.

4.3.3 Preprocessing Step

Consider a system of N nodes where each node can be in one of S states (i.e. $|\Sigma| = S$). To represent the state of a node, we use a vector of length S with one-hot encoding. For example, in a 4-state system, a node in the second state would be represented by the vector $[0, 1, 0, 0]$.

Stacking N state vectors, one per node, gives the state matrix $\mathbf{S} \in \{0, 1\}^{N \times S}$. This representation of the nodes' states allows easy calculation of the number of neighbours in each state that each node has: the matrix product of the graph adjacency matrix $\mathbf{A}$ and state matrix $\mathbf{S}$ gives a matrix $\mathbf{C} \in \mathbb{N}^{N \times S}$ of the counts of each node's neighbours in each state:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{S} \tag{4.3}$$

In the case of a directed graph (as used here), we can separately calculate the count of outgoing nodes in each state as above, and incoming nodes in each state using the transpose of the adjacency matrix:

$$\mathbf{C_{out}} = \mathbf{A} \cdot \mathbf{S} \tag{4.4}$$

$$\mathbf{C_{in}} = \mathbf{A}^T \cdot \mathbf{S} \tag{4.5}$$

The $\mathbf{C}$ matrices count only neighbour states and do not give separate information about each node's *own* state. If each node has a self-loop (equivalent to having ones on the diagonal of the adjacency matrix) this would ensure that the node's own state is counted, in the manner of a totalistic CA rule. However, it would not distinguish between the node's own state and the counts of its neighbours, so could not be used to represent outer-totalistic

rules.

To allow this finer distinction, we borrow an idea from Graph Neural Networks (GNNs) (Scarselli et al., 2009) which are used for machine learning on data structured as graphs. They perform a convolution operation to combine the information at each node with that of its neighbours. To do this they use the graph Laplacian matrix. The graph Laplacian $\mathbf{L}$ is defined as the difference between a diagonalised degree matrix $\mathbf{D}$ and the adjacency matrix $\mathbf{A}$:

$$\mathbf{D}_{ij} = \begin{cases} \deg(v_i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \tag{4.6}$$

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{4.7}$$

We use the graph Laplacian instead of the adjacency matrix when counting neighbouring node states by taking the matrix product of $\mathbf{L}$ and $\mathbf{S}$:

$$\mathbf{F} = \mathbf{L} \cdot \mathbf{S} \tag{4.8}$$

This gives us a matrix $\mathbf{F} \in \mathbb{Z}^{N \times S}$ of the filtered values of neighbouring nodes. Using the Laplacian essentially weights each node's own state by its degree, and weights connected node states by -1.

We extend this approach to directed graphs by defining an "out Laplacian" as the difference between the diagonalised out-degree matrix $\mathbf{D_{out}}$ and the adjacency matrix, and an "in Laplacian" as the difference between the diagonalised in-degree matrix $\mathbf{D_{in}}$ and the transpose of the adjacency matrix:

$$\mathbf{L_{out}} = \mathbf{D_{out}} - \mathbf{A} \tag{4.9}$$

$$\mathbf{L_{in}} = \mathbf{D_{in}} - \mathbf{A}^T \tag{4.10}$$

Taking the product of each of these with the matrix $\mathbf{S}$, of the node states, gives us the filtered values of outgoing and incoming nodes $\mathbf{F_{out}}$ and $\mathbf{F_{in}}$:

$$\mathbf{F_{out}} = \mathbf{L_{out}} \cdot \mathbf{S} \tag{4.11}$$

$$\mathbf{F_{in}} = \mathbf{L_{in}} \cdot \mathbf{S} \tag{4.12}$$

Together, $\mathbf{F_{out}}$ and $\mathbf{F_{in}}$ provide a considerable amount of information about each node's neighbourhood, and they remain of a fixed size no matter how each node's number of connections varies. This makes them a suitable input for our transition function.

Graph Example An example of this bi-directional Laplacian filtering of neighbourhood information is shown for the graph in Figure 4.1. The corresponding adjacency matrix $\mathbf{A}$, state matrix $\mathbf{S}$, degree matrices, "out Laplacian", "in Laplacian" and filtered states are:

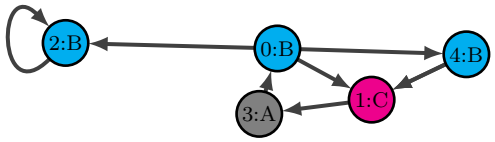


Figure 4.1: An example network. Nodes are numbered 0–4 and each is in the the state A, B or C (also indicated by the node colour). The adjacency matrix, matrix of node states, and graph Laplacians for this network are given in equations 4.13 and 4.15.

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{4.13}$$

$$\mathbf{D_{out}} = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D_{in}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.14}$$

$$\mathbf{L_{out}} = \begin{bmatrix} 3 & -1 & -1 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{L_{in}} = \begin{bmatrix} 1 & 0 & 0 & -1 & 0 \\ -1 & 2 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.15}$$

$$\mathbf{F_{out}} = \begin{bmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & -1 & 0 \\ 0 & 1 & -1 \end{bmatrix} \quad \mathbf{F_{in}} = \begin{bmatrix} -1 & 1 & 0 \\ 0 & -2 & 2 \\ 0 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \tag{4.16}$$

4.3.4 Single-Layer Perceptron

We use a single-layer perceptron (SLP) to define our state transition function. The input layer consists of a vector of information about each node plus a bias term, and the output layer consists of a vector of the same length as the number of possible “actions" that may be taken by the node. The index of the maximum value in the output vector indicates the action to be taken. The actions consist of changing the node’s state or duplicating or removing the node. The length of the output vector is therefore $S + 2$. When a node is duplicated, its row and column in the graph’s adjacency matrix is repeated, meaning the new node gets all

the same incoming and outgoing connections as its parent.

An SLP can be implemented using a matrix-vector multiplication in which the real-valued matrix ($\mathbf{W}$) represents the connection weights. Rather than doing this once per node, we do it simultaneously for all nodes by stacking the input vectors for all nodes in the graph.

We define a matrix $\mathbf{H}$ as a horizontal concatenation of matrices $\mathbf{F_{out}}$ and $\mathbf{F_{in}}$ together with a vector of ones for the bias term. For ease of reading, the matrix sizes are shown as subscripts.

$$\mathbf{H}_{(N\times(2S+1))} = [\mathbf{F_{in}}_{(N\times S)}, \mathbf{F_{out}}_{(N\times S)}, \mathbf{1}_{(N\times 1)}] \tag{4.17}$$

Each row of this matrix serves as the input to the SLP for one node: each row gives information about a node's neighbourhood (as discussed above) together with a bias term which is always 1. For the example graph in Figure 4.1, the matrix $\mathbf{H}$ is:

$$\mathbf{H} = \begin{bmatrix} -1 & 1 & 0 & 0 & 1 & -1 & 1 \\ 0 & -2 & 2 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \tag{4.18}$$

To run the SLP for all nodes simultaneously we multiply $\mathbf{W}$ by $\mathbf{H}$:

$$\mathbf{O}_{(N\times(S+2))} = \mathbf{H}_{(N\times(2S+1))} \cdot \mathbf{W}_{((2S+1)\times(S+2))} \tag{4.19}$$

This produces an output matrix $\mathbf{O}$ with size $N \times (S+2)$. The index of the maximum value in each row of this matrix indicates the action which that node should take.

Continuing the example, consider the weight matrix:

$$\mathbf{W} = \begin{bmatrix} -0.2 & -0.7 & -0.3 & 0.1 & -0.8 \\ -0.1 & -0.1 & 0.4 & 0.4 & -0.3 \\ 0.3 & -0.8 & 0.0 & 0.5 & -1.0 \\ 0.9 & -0.4 & -0.6 & 0.9 & 0.9 \\ 0.9 & -0.6 & 1.0 & -0.4 & 0.5 \\ 0.1 & 0.7 & 0.8 & 0.7 & 0.3 \\ 0.2 & 0.3 & -0.6 & 0.1 & 0.7 \end{bmatrix} \tag{4.20}$$

Multiplying this by the $\mathbf{H}$ matrix in Equation 4.18 gives the output matrix (rounded to one

decimal place):

$$\mathbf{O} = \begin{bmatrix} 1.1 & -0.4 & 0.3 & -0.7 & 1.4 \\ 0.2 & 0.0 & 0.0 & 0.1 & -1.3 \\ 0.2 & 0.3 & -0.6 & 0.1 & 0.7 \\ -0.3 & 0.6 & -2.5 & 1.0 & 1.3 \\ 1.0 & -1.0 & -0.4 & -1.0 & 0.9 \end{bmatrix} \tag{4.21}$$

Taking the index of the maximum value in each row (arg max) gives $[5, 1, 5, 5, 1]$. This is the final output of the transition function and indicates which action each of the five nodes should take. Values in the range 1–3 indicate that the node should change to the corresponding state (A–C); the value 4 indicates that the node should be removed; the value 5 indicates that the node should be duplicated. Thus, in the example, nodes 1 and 4 change to state A; nodes 0, 2 and 3 are duplicated.

Here, when a node is duplicated, the original node retains its original state, whilst the copy takes the state indicated by the index of the next highest value in that row of $\mathbf{O}$ (eg. node 0 would be duplicated; one copy would retain the original state B, the other would take state A). The resulting graph after one development step is shown in Figure 4.2. The nodes labelled 5, 6 and 7 are duplicates of nodes 0, 2 and 3 respectively, and have the same in/out edges from/to other nodes as their parents. In terms of the adjacency matrix, assigning the offspring nodes the same edges as their parents can be achieved simply by duplicating the relevant rows and columns of the adjacency matrix, as indicated with the coloured shading below:

$$\mathbf{A}_{t+1} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \left(\begin{array}{cccccc|ccc} 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \end{array} \tag{4.22}$$

For example, the 0^{th} row and column (indicating the outgoing and incoming edges of node 0) are duplicated to create the 5^{th} row and column. The lower-right quadrant is filled with zeros since there are no connections between newly created nodes.

This example has not shown the removal of nodes, but that is is easily achieved by removing the relevant row and column in the adjacency matrix. This removes the node from the graph, along with all of its incident edges.

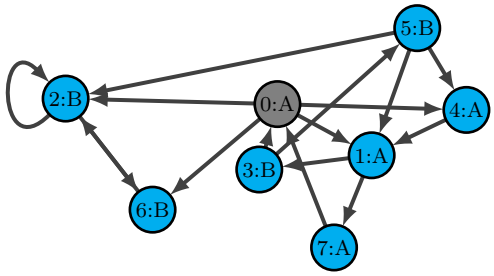


Figure 4.2: The graph in Figure 4.1 after one development step, using the SLP weights matrix in Equation 4.20 as explained in the text.

4.3.5 Comparison with CA rules

The behaviour of the system, in terms of changes of node states and addition/removal of nodes, is entirely determined by the weights of the SLP ($\mathbf{W}$). These weights therefore perform the same function as the update rules of a conventional CA. In most CAs however, the rules are specified as a set of discrete node state changes. For example, in ECA an state change is specified for every possible configuration of the neighbourhood of three cells, of which there are 8 (2^3). An ECA ruleset therefore consists of 8 rules each of which specifies a new state for the central node. The ruleset can be described as belonging to $\{0,1\}^8$. By contrast in DGCA, the update function is parametrised by a real valued matrix of SLP weights: $\mathbf{W} \in \mathbb{R}^{(5\times4)}$ when the number of node states S is 2.

This makes the DGCA update rule less interpretable, in common with many systems based on ANNs, in which it is difficult to say what the weights of the system represent. On the other hand, it may make the system more evolvable, since it is possible to adjust the update rules by small increments, allowing interpolation between different behaviour. This is one of the main advantages of ANNs, that the weights can be incrementally adjusted, whether by gradient descent or evolution, to achieve the desired outcome.

Furthermore, using an SLP rather than an MLP does allow some interpretability. We can think of each column of $\mathbf{W}$, being matched against the node's neighbourhood information vector (a row of $\mathbf{H}$) - see Equation 4.19. The column which is the closest match will have the highest dot product with the information vector and will therefore lead to the highest value in that row of $\mathbf{O}$. The arg max of $\mathbf{O}$ rows is then used to choose the new state (or action) of the node. In a sense therefore, each node's information vector is undergoes a fuzzy match with the columns of $\mathbf{W}$ (by using the dot product as a similarity measure) to decide what to do next.[2] This view allows the columns of $\mathbf{W}$ to be interpreted similarly to the individual rules of an ECA.

[2]This is analogous to the attention mechanism in transformer ANN architectures, where the similarity of "query" vectors to "key" vectors (assessed using the dot product or cosine similarity) is used to select "value" vectors.

4.3.6 Comparison with (G)NCA

The procedure described above has much in common with the way neural cellular automata (NCA) work (Mordvintsev et al., 2020; Tavares et al., 2015; Wulff & Hertz, 1992), with a few important differences. NCA are fixed-topology grid-based CA in which the update rule is replaced by a multi-layer perceptron (MLP) neural network. Since the topology and the neighbourhood of the CA are fixed, the input to the MLP can simply be the raw vector of the neighbourhood cell states. Alternatively, in Mordvintsev et al. (2020), a series of two-dimensional convolution kernels are applied to the cell neighbourhoods as a preprocessing step. The MLP has a single output which can be used directly as the next state of the cell (either a scalar value or a vector if cell states are represented as a vector). This is in contrast to the system described here, which cannot use a raw vector of neighbourhood states since there may be varying numbers of neighbours, and so requires the preprocessing step using bi-directional Laplacian filtering. The present system must not only choose a new state for each node, but also whether or not the node should be duplicated or removed, so the output of the SLP is interpreted as a choice of action for the node to take, rather than being directly used as a new node state.

GNCA (Grattarola et al., 2021) are designed to choose the next state for each node, and do not have the ability to add or remove nodes. However, they did conduct an experiment in which each node's state vector represented its spatial position, which in turn determined its connectivity, leading to graphs with dynamic topology but a fixed number of nodes.

(G)NCAs use MLPs rather than SLPs to update cell states as this allows a greater diversity of update rules to be represented. Many of the most interesting rules in conventional CAs, such as the ECA rules 110 and 30 in Wolfram's cataloguing scheme (Wolfram, 2002), are not linearly separable (Tavares et al., 2015). This means that they cannot be represented by a SLP. MLPs with even a single hidden layer can be fitted to functions which are not linearly separable (as shown by the Universal Approximation Theorem of Cybenko (1989)). Since part of the purpose of NCA is to discover interesting rules which create high-complexity patterns, it makes sense to use MLPs there. The main aim of the present work is to introduce the model of DGCA as simply as possible, hence the use of SLPs. Future work may experiment with the use of MLPs to power the update behaviour at each timestep. However, as will be seen, even SLP based update functions can produce rich and complex behaviour.

4.3.7 Comparison with STNs

Signal transduction networks (STNs) are chemical networks within cells which allow them to react to complex environmental signals. These may include chemical signals produced by the organism itself as part of the endocrine system. The closest biological analogy with the neighbourhood signalling used in CA (including DGCA) is juxtacrine signalling, in which

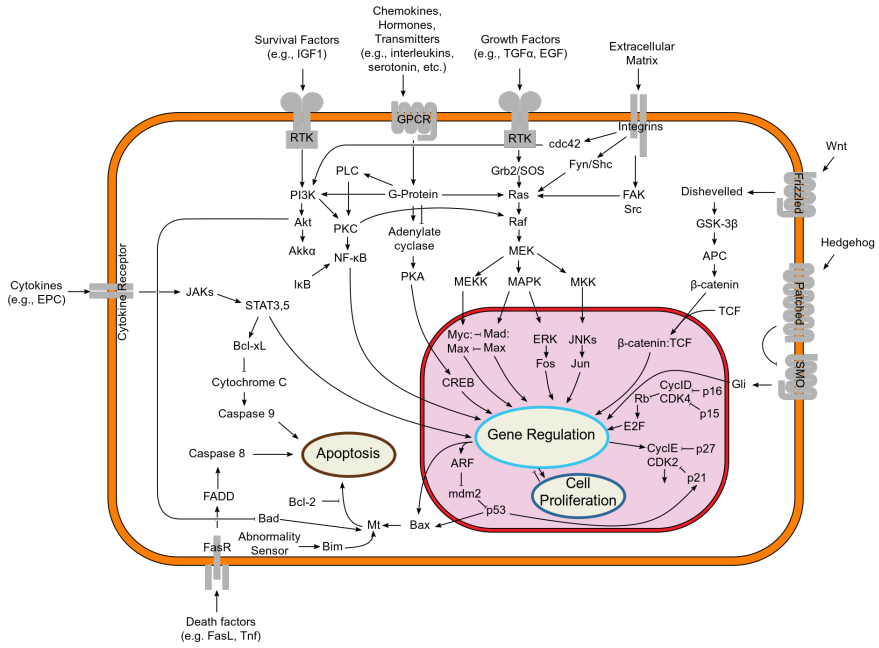


Figure 4.3: Major signal transduction pathways in a mammalian cell. The cell receives signals from its environment, including neighbouring cells (juxtacrine signalling). After processing through a chemical network, these can result in signals for the cell to divide or die (apoptosis).
This image is from: https://commons.wikimedia.org/w/index.php?curid=12081090 (Signal transduction v1.png, CC BY-SA 3.0)

cells which are physically in contact with one another may transfer chemical signals. Once signals are received they may are combined and processed, potentially leading to cell state changes or actions. As pointed out by Bray (2009), these signal transduction networks can be very complex and implement non-linear responses, memory and learning - things that are more usually associated with neural networks. An illustration of the major mammalian signal transduction pathways in shown in Figure 4.3. As indicated in the diagram, the STN may lead to cell behaviour such as division and apoptosis (death). This provides an analogy for DGCA in which neighbourhood signals are processed by a small "internal" neural network and can lead to actions including division and removal, as well as changes of state.

4.4 Experiment

The aim of this chapter is to perform an initial investigation to explore and characterise some of the different behaviours produced by DGCA. For this initial investigation, we use random search over developmental rules, as represented by the SLP weights, $\mathbf{W}$.

4.4.1 Detecting Attractors

As noted in Section 4.2, conventional CA with a finite number of cells have a finite state space and therefore ultimately always reach an attractor, provided their update rules are

deterministic.

In the case of a growing graph, the state space can get bigger at each timestep, meaning that it is not inevitable that the system reaches an attractor: the state space can grow faster than it can be explored. Subjectively speaking, runaway growth of the system is not particularly interesting behaviour. For example, a doubling of the number of nodes at each timestep is a low complexity behaviour. The type of behaviour we are interested in exploring is *growth that comes to a halt*. Such behaviour could be an analogue of the self-regulating growth seen in living organisms: growing to a mature state.

It is not possible to know in advance whether growth will stop for a given rule-set. However, it is possible to be sure that growth *has* stopped in a particular case. This occurs if the system reaches an attractor. Even if the number of nodes remains constant for several timesteps, if the system has not reached an attractor, growth may resume again at any point. The system may only partially explore the state space for a given number of nodes before moving into a different state space with a different number of nodes. However, once an attractor has been reached the system will remain in that attractor forever (since it is a closed, deterministic system).

To test whether the present system has reached an attractor, we check whether the graph is isomorphic to a previous graph *given* the node states. Unlike a conventional CA, we cannot simply look at the vector of node states, but must also check that those states exist at isomorphically equivalent nodes. A formal definition of this vertex-label conditioned isomorphism is given below (adapted from Hsieh et al. (2006)).

Definition 1 (Isomorphism of vertex-labelled directed graphs) *Graph* $g = (V_g, E_g, \sigma_g)$ *is isomorphic to graph* $h = (V_h, E_h, \sigma_h)$ *iff there exists a bijective function* $\phi : V_g \to V_h$ *such that:*

1. $\forall u \in V_g, \sigma_g(u) = \sigma_h(\phi(u))$
2. $\forall (u, v) \in E_g \leftrightarrow (\phi(u), \phi(v)) \in E_g$

For brevity, in the remainder of this thesis we refer to "isomorphism classes" rather than "vertex-labelled isomorphism classes". We also use the term *graph state* to indicate the same concept: if two graphs belong to the same isomorphism class they have the same graph state (this should not be confused with *node state* meaning the state, or "label" $\sigma_g(u)$ of an individual node within a graph).

After every development step, it is necessary to check the current graph state against all previously seen graph states to determine whether the system has entered an attractor. However, since graph isomorphism is a computationally expensive problem, it is useful to have a heuristic for checking whether a graph state *might* be isomorphic with one that has been previously seen, before performing a full isomorphism check. A convenient algorithm

for this heuristic check is the Weisfeiler and Leman (1968) graph hash[3]. This iteratively aggregates node labels over the node neighbourhood and hashes the results, eventually producing a single hash value for the graph. If two graphs have different hash values then they are definitely non-isomorphic. If the values are the same then they may be isomorphic. The algorithm has more chance of distinguishing graphs if it is run for more iterations. However there are certain sets of non-isomorphic graphs which the Weisfeiler–Leman algorithm is never able to distinguish (Cai et al., 1992). For these reasons, we use the fast Weisfeiler–Leman algorithm as a preliminary check to identify any previous graph states which *may* be isomorphic to the current graph state. If there are any candidates we then apply a full graph isomorphism check. This process is shown in Algorithm 1.

Algorithm 1 DGCA Run

$f \leftarrow$ parameterised DGCA update function
$g \leftarrow$ seed graph
$graphs \leftarrow []$
$hashes \leftarrow []$
$max\_steps \leftarrow 200$
$max\_nodes \leftarrow 512$
for $i = 0$ to max_steps **do**
    $h \leftarrow$ `WL_hash`(g) ▷ fast Weisfeiler–Leman hash
    if $h \in hashes$ **then** ▷ graph state may have been seen before
        for all i such that $hashes[i] = h$ **do**
            $iso \leftarrow$ `isomorphism_check`$(g, graphs[i])$ ▷ expensive isomorphism check
            if iso **then**
                return AttractorFound
            end if
        end for
    end if
    $graphs \leftarrow [graphs + g]$
    $hashes \leftarrow [hashes + h]$
    $g \leftarrow f(g)$ ▷ Run an update step
    if $\#(g) = 0$ **then** ▷ Check size of graph
        return ZeroNodesReached
    end if
    if $\#(g) > max\_nodes$ **then**
        return MaxNodesReached
    end if
end for
return MaxStepsReached

4.4.2 Five classes of growth behaviour

We identify five classes of behaviour the system can display in terms of its growth.

[3]Note that the Weisfeiler–Leman algorithm is very similar to the neighbourhood information aggregation step of GNNs (Grohe, 2021), which is part of the inspiration for the DGCA system.

- **Runaway Growth** occurs when the number of nodes in the graph increases rapidly and shows no sign of stopping. This can be exponential if all nodes are duplicated at each timestep, or at a sub-exponential rate if a subset of nodes is duplicated. Here we use an arbitrary cut-off of network size to categorise runaway growth. Of course, it may be that a graph would stop growing if it were run for longer, so this categorisation is contingent on the cut-off point.

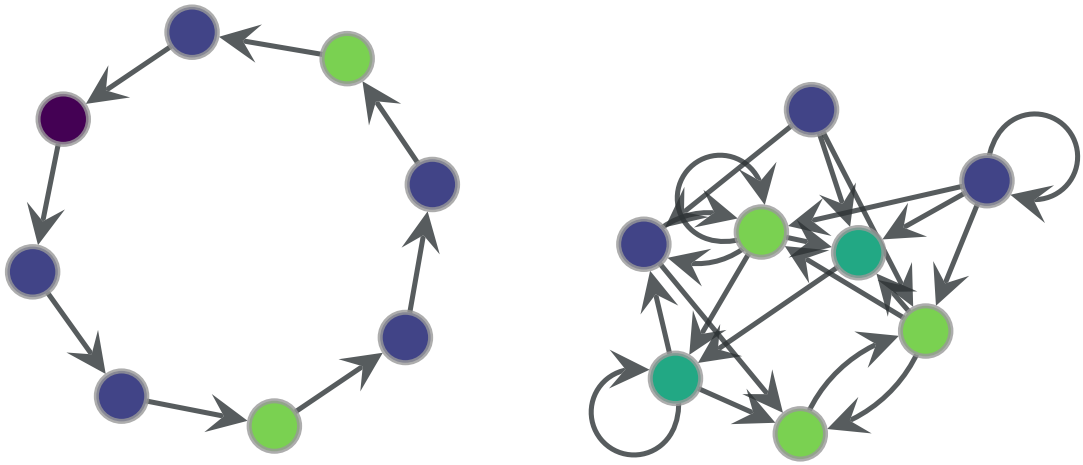
- **Death** occurs when the number of nodes in the network shrinks to zero. This can occur if nodes are removed at every timestep. Once there are zero nodes in the network it is impossible for new nodes to be produced, since there are no nodes to duplicate.

- **No Growth** means that the graph remains the same size as the seed graph. This does not preclude the network structure changing: one node could be added and one removed at each timestep, for example, meaning the network would retain the same size but might change connectivity. To be included in this class, the graph must be in an attractor: this confirms it will not subsequently change size.

- **Halting Growth** is the class of behaviour we are most interested in. The graph changes size from its original seed (it can get bigger or smaller) but eventually reaches an attractor, meaning growth has finished. In the case of a point attractor, the graph remains at a fixed size. A cyclic attractor can cycle between node state configurations in a fixed size network, or can cycle between states in different sized networks. Although the size of the graph in the latter case is not static, we still include it in the halting growth class of behaviour as the network size fluctuates regularly and runaway growth or death is now impossible.

- **Slow Growth** is how we categorise graphs for which we have not been able to determine another class of behaviour after some arbitrary number of timesteps. The graph has not grown beyond the threshold which would put it in the runaway growth category, nor has it reached an attractor. This categorisation is contingent on the cut-off number of timesteps.

Three of these classes represent the system reaching an attractor. Reaching a zero node state (“death") can be thought of as a point attractor. “Halting growth" is defined as reaching an attractor which has more than zero nodes. The “no growth" case is a special case of halting behaviour: the system ends up in an attractor with a fixed number of nodes which is the same as the original number of nodes in the seed network, and during the transient phase (before it has reached the attractor) the number of nodes also remains fixed. However, we treat it separately since such static behaviour is the only behaviour possible with a conventional finite-space CA, and is less interesting in the present investigation of growth. Note that the different return statements in Algorithm 1 correspond



(a) Unidirectional Ring (b) Random Graph (example)

Figure 4.4: The two types seed graphs used in experiments. The ring graph always has the same structure but node states (illustrated by colours) are randomly assigned for each trial. The random graph is initialised randomly for each trial. It is constructed as an Erdős–Rényi graph with $n = 8$, $p = 0.23$. Node states are randomly assigned.

to the behaviour classes (`MaxNodesReached` = *runaway growth*; `ZeroNodesReached` = *death*; `MaxStepsReached` = *slow growth*; `AttractorFound` may indicate *halting growth* or *no growth*, depending on whether the network has changed size at all during the run).

4.4.3 Results

We run an experiment to ascertain how frequently each of the five classes of behaviour outlined above are observed in systems with 2–8 states. For each number of states we create 1000 seed graphs with randomly initialised weight matrices in their SLPs (which effectively define the update rules for the system).

For the initial experiment, each seed graph has the same structure: a ring of 8 nodes with edges connecting them in one direction, as shown in Figure 4.4a. The node states are assigned randomly. In this experiment, the same seed graph structure is used in each trial as we are interested in whether different SLP weights can result in significantly different behaviour. As with many discrete dynamical systems, behaviour is determined both by the update function and the starting point (seed graph), and we focus on the first of these variables, holding the seed graph structure constant.

However, in order to make sure that the results obtained are robust and are not an unusual consequence of starting with this particular seed graph structure, we conduct a second experiment in which every trial uses a random seed graph, constructed as an Erdős–Rényi graph with $n = 8$, $p = 0.23$. Once again, node states are randomly assigned. An example of one of the random seed graphs used is shown in Figure 4.4b

For each trial (with random SLP weights), we run the system for 200 timesteps. Runaway

growth is defined as the graph growing to greater than 512 nodes within this time. In most cases, within the 200 timesteps, the system will either exceed this threshold or will enter an attractor corresponding to one of three classes of behaviour: death, static or halting growth. In rare cases (approx 1% of trials), the system shows neither runaway growth nor reaches an attractor within the 200 timesteps, in which case we class the behaviour as “slow growth". The results are shown in Figure 4.5. There is also one more class shown in the charts, namely “Unknown” - this applies when we were unable to establish whether the system had entered an attractor or not because the graph isomorphism check took too long. As mentioned above (Section 4.4.1), graph isomorphism is a computationally expensive process, so we used a timeout to abandon this check if it was taking too long.

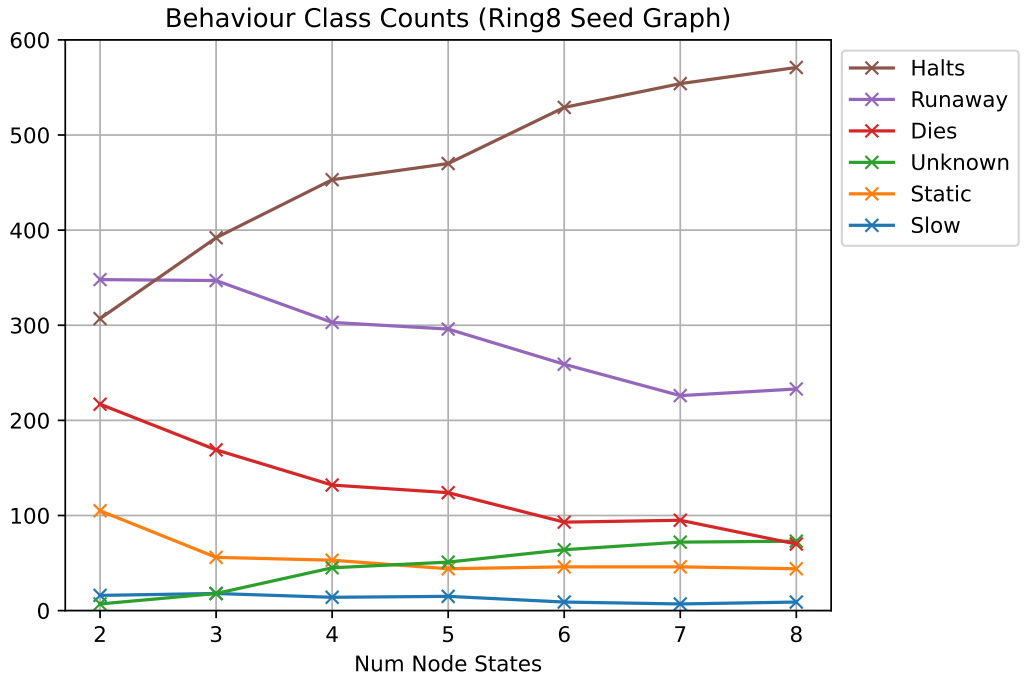
For a 2-state system, the runaway behaviour is the most common and is observed in 348 out of the 1000 trials. The halting growth behaviour is the next most common with 307 occurrences. This alone is a surprising result, given that the state space is unbounded. Despite this, the system still manages to find attractors where growth halts.

As the number of states in the system is increased, the runaway growth becomes less common and the halting growth becomes prevalent. By the time we get to a 6-state system, over half of the trials result in halting growth.

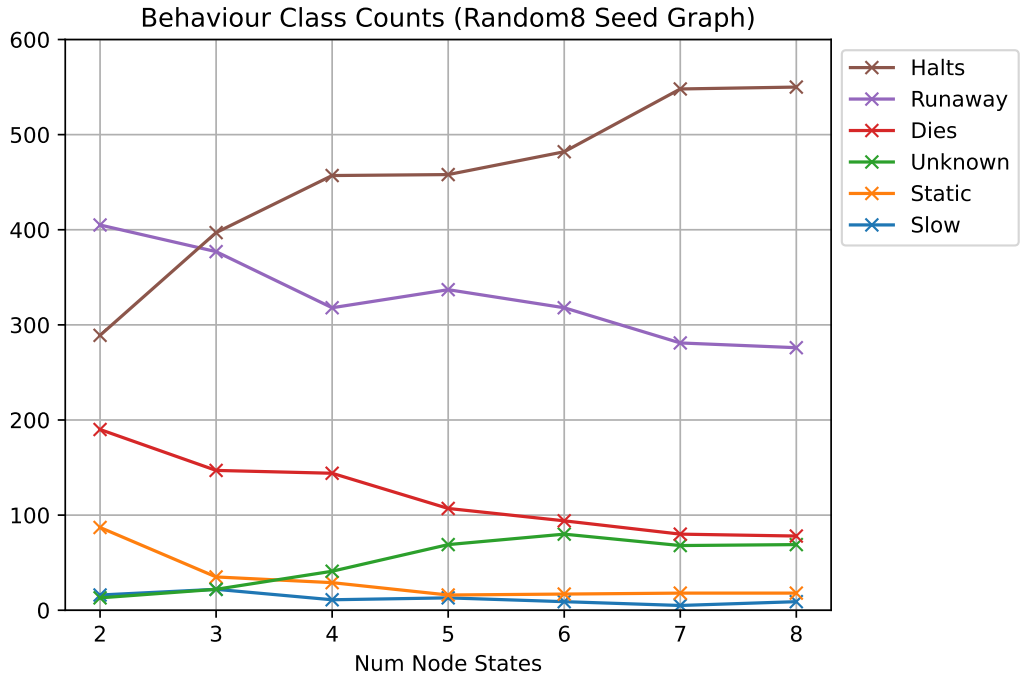
It is not immediately obvious why a system with more states would be more likely to find an attractor within the 200 timesteps used in the experiment. Indeed, for a given network size, the state space is larger if there are more possible states that each node can be in. For example, for a particular 100 node graph structure with two node states the number of possible overall states of the system is $2^{100} \approx 10^{30}$, whereas with the same graph and five possible node states, the number of possibilities is $5^{100} \approx 10^{70}$. One might therefore expect the system to take longer to reach an attractor when the number of possible nodes states is higher, since the state space for a given network size is order of magnitudes larger.

On the other hand, due to the way the transition rule SLPs are used, the node duplication and removal operations are less likely to occur when there are more states in the system. These actions are only two of the possibilities that the SLP may choose, while the other actions are changing to one of S states. All else being equal, there is a higher chance of a randomly initialised SLP choosing a simple “change state" operation when there are more states to choose from. This means that we might expect the systems containing more states to change size more rarely than those with more states. This in turn makes it more likely that an attractor will be found, since the system has more time to explore the graph state space for a fixed size and topology.

It therefore seems likely that the results shown above, with the halting behaviour increasing as the number of node states in the system increases, is an unintended consequence of the design decisions about how the SLP selects the node action. This is reflected upon further when an improved version of the system is introduced in Chapter 6.

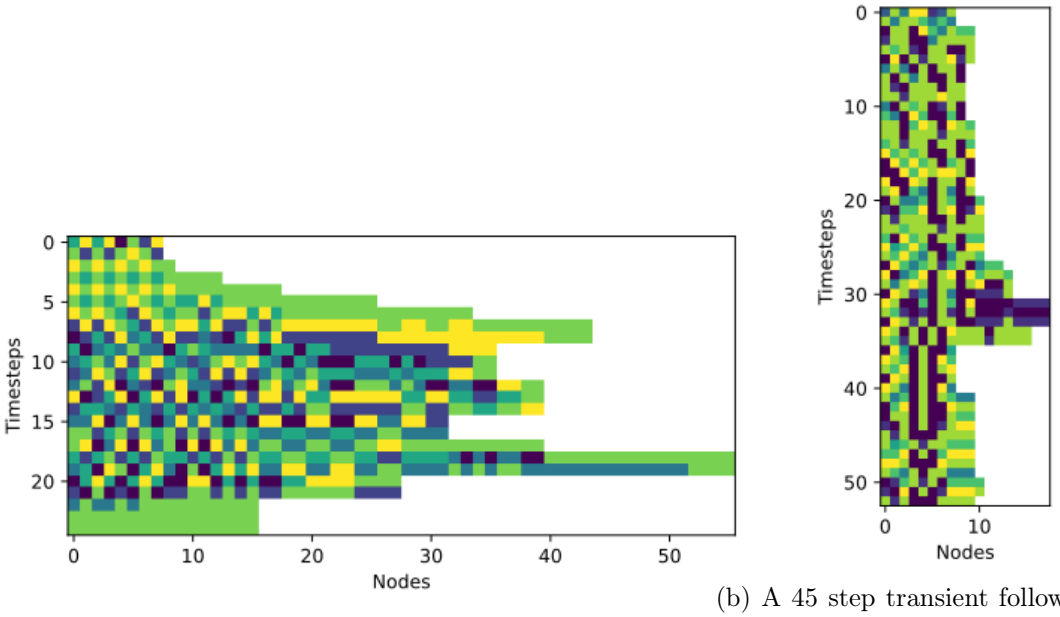


(a) Using a ring of eight nodes as the seed graph



(b) Using a random eight node graph as the seed graph

Figure 4.5: Counts of the different behaviour classes after 1000 trials with randomly initialised growth rules (**W** matrix), using systems with 2 to 8 node states. As the number of node states in the system increases, runaway growth behaviour is seen less, and halting growth behaviour is seen more.



(a) A 23 step transient followed by a point attractor

(b) A 45 step transient followed by a 7 step attractor

Figure 4.6: Example space-time diagrams for two 6-state systems in which growth halts. Timesteps proceed down the y-axis. At each timestep, the state of every node is concatenated into a vector, shown as a line containing six different possible colours. This line gets longer as the graph grows. It can be seen that the graph grows and shrinks over time before reaching a stable state. In (a), the final (bottom) two lines are the same, indicating that the system has reached a point attractor. In(b), the system has a transient length of 45 steps followed by a 7-step attractor cycle (the line at timestep 52 is the same as that at timestep 46).

4.4.4 Space-Time Diagrams

A common way of visualising the behaviour of ECAs is to plot space-time diagrams that represent the state of all cells in the system at each timestep by a row of coloured squares (e.g. see Figure 2.1). These are stacked on top of each other to show how the system evolves over time. This method is not directly applicable to the present system as the nodes are not ordered, and because the number of nodes can grow and shrink over time. However, by using a fixed arbitrary ordering of the nodes, and varying row sizes, this type of diagram can be adapted for use with the present system, as shown in Figure 4.6. Newly created nodes are added at the right-hand end of the line, and when a node is removed, the rest of the line is shifted left.

The diagram of development of a 6-state system in Figure 4.6a shows the complexity of the behaviour which is possible with this type of system. The graph grows rapidly from its initial eight nodes, displaying a wide variety of node states. Its size then fluctuates until it reaches its final size of 16 nodes (all of which are in the same state). After this no more changes take place: it has reached a point attractor.

A different kind of behaviour is shown in Figure 4.6b. The graph size remains fairly small throughout. After a transient of 45 steps, a 7-step attractor cycle is found. Further

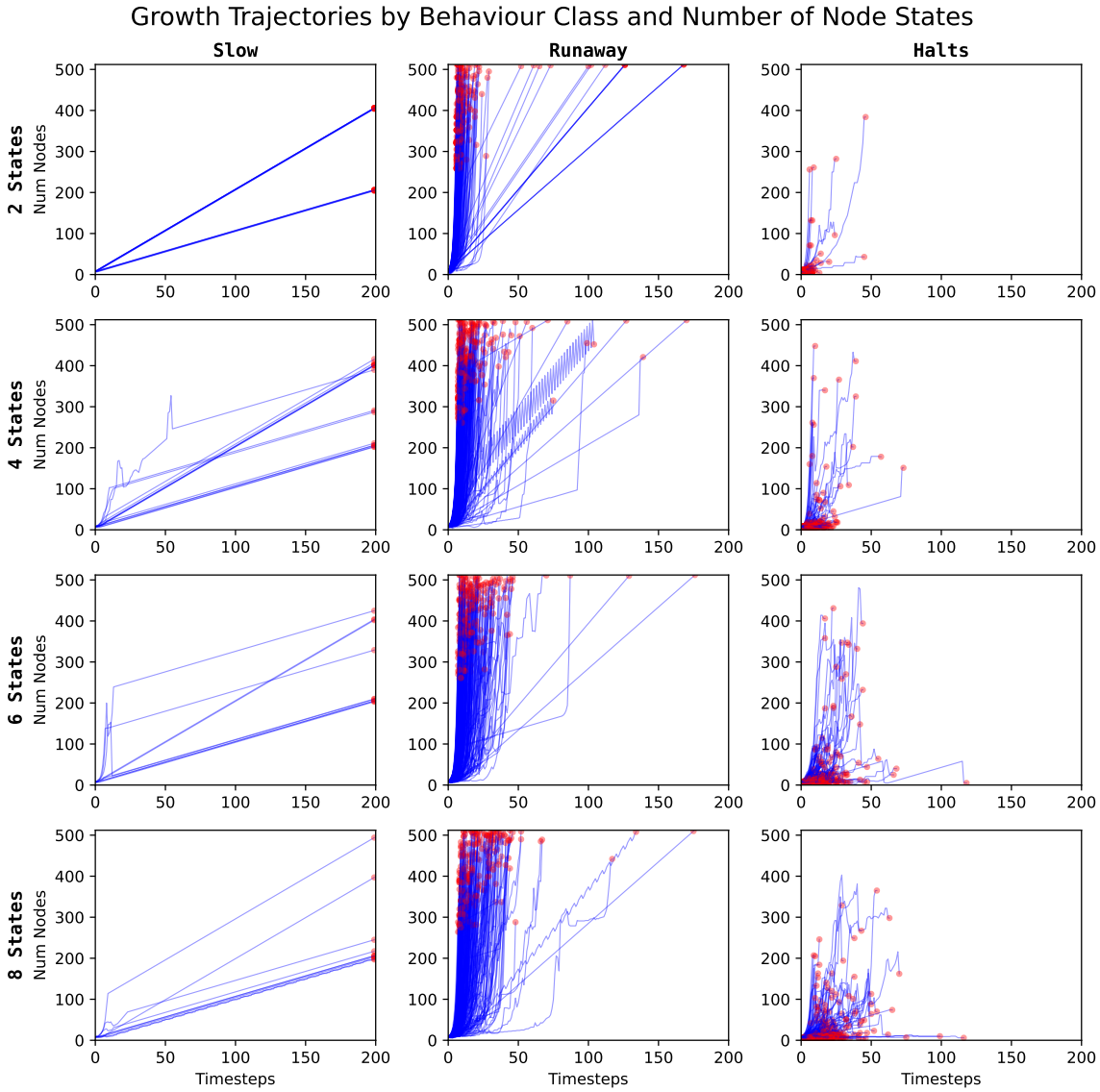


Figure 4.7: The growth trajectories for the slow growth, runaway growth and halting growth behaviour classes, shown for systems with 2, 4, 6 & 8 node states. Blue lines show the number of nodes in the graph over time for each trial. The red dots show the point at which a trial was terminated. For the slow growth class, this was when it reached the maximum number of timesteps (200). For the runaway growth class, trials were terminated one step before they would have gone over the maximum number of nodes (512). For the halting growth class, trials terminated at the first repeated graph state (ie. when they were in an attractor). For an enlarged example of the halting growth class see Fig. 4.8.

space-time diagrams will be shown in subsequent chapters.

4.4.5 Growth Trajectories

The growth trajectories of individual trials for the different behaviour classes and different numbers of node states are shown in 4.7. These trajectories are shown for the eight node

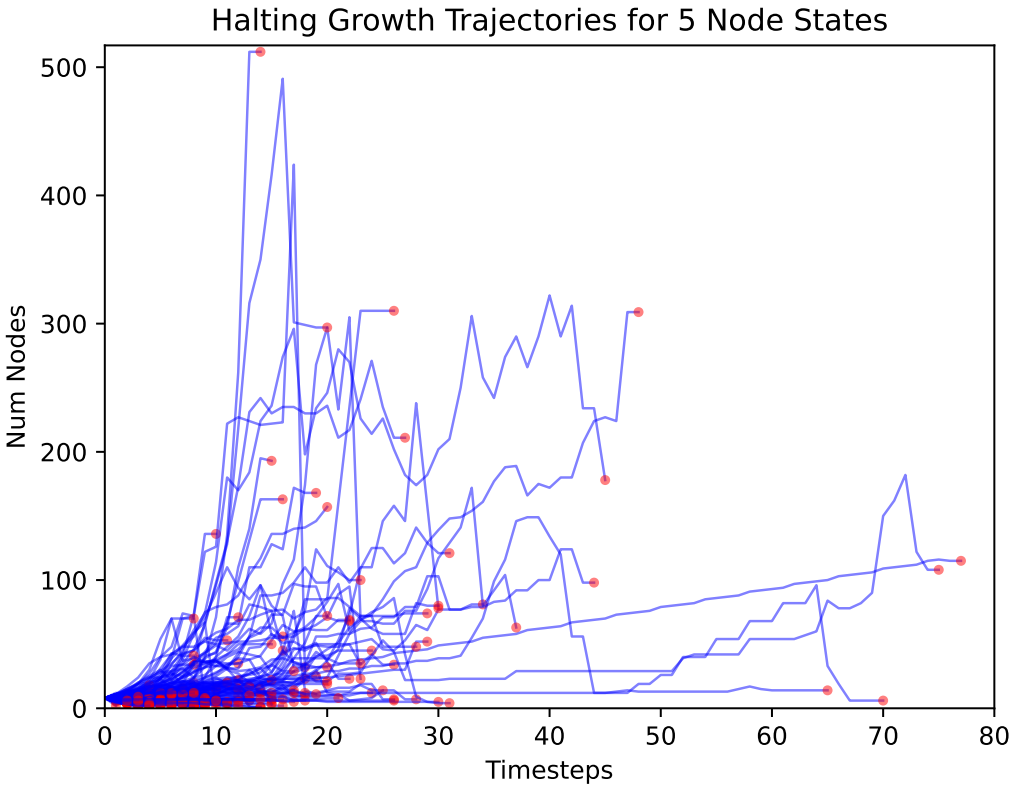


Figure 4.8: The growth trajectories for the 470 runs of the 5 node state system which resulted in halting growth (ie. where an attractor was found). Note that the red dots indicate the termination point of the trial at the first repeated graph state. For cyclic attractors, this therefore means that trials were terminated after one pass around the attractor cycle.

ring seed graph (as shown in Figure 4.4a). See Figure A.1 in Appendix A for the equivalent plots for the eight node random seed graph - these do not show major differences with the results displayed here. Only the slow growth, runaway growth and halting growth behaviour classes are shown. The "static" and "death" classes are not plotted since they show either a fized number of nodes, or a (usually rapid) decline to zero nodes.

The slow growth class (in which graphs grew for 200 timesteps without either reaching an attractor or exceeding the size limit of 512) generally show linear growth rates, although there are a few more interesting trajectories where the growth rate changes after a number of timesteps. There also seems to be a greater diversity of growth rates in the systems with more node states.

The runaway growth class (in which graphs grew to more than 512 nodes within the 200 timesteps), show either (faster) linear or exponential growth rates. There are also some examples of "sawtooth" growth where the number of nodes oscillates whilst moving upwards.

Trajectories included in these two classes may have reached an attractor if they had been allowed to run for longer or if the maximum size cutoff had been higher. However, it seems likely that those with linear or exponential growth rates would have continued on that course

without reaching an attractor, even on a longer run. Intuitively it seems more likely that those with more irregular growth trajectories may have reached attractors if they had been run for longer, since most of the growth trajectories in the halting growth class are irregular.

The third column of Figure 4.7 shows the growth trajectories of trials which were in the halting growth class, ie. which found an attractor within the time and size limits. It is immediately clear from the shape of these trajectories that these graphs showed richer dynamics, including frequent changes of growth rates, and unpredictably fluctuating sizes. Apart from the fact that there are more trials which ended up in this class for systems with more node states (as shown in the results in Figure 4.5), there is no obvious characterisation to be made about the effect on increasing the number of node states on this class of behaviour. An enlarged version of the halting growth trajectories for systems with five node states is shown in Figure 4.8 (note that this is not featured in the subplots of Figure 4.7, which only show systems with 2, 4, 6, and 8 node states).

Unsurprisingly, these growth trajectories do not have much in common with observed biological growth rates, where the total number of cells may grow rapidly before followed by a period of slowing growth. An example of this in the case of neurogenesis is shown in Postel et al. (2019). However, the complexity of the growth trajectories shown in the halting growth class give an indication of the rich dynamics possible with DGCA.

4.5 Conclusion

This chapter has introduced the basic DGCA model, an extension of GCA which allows the structure of the graph to develop over time. The system displays not only the dynamics of nodes changing state on a given graph topology, but also the dynamics of the structure of the graph itself changing. The experiment shown uses random search to examine the behaviour of different rule-sets (as represented by SLP weights). Five different classes of behaviour are defined, three of which involve the system reaching an attractor state. One of these (halting growth) appears to be the most “interesting" as it involves the seed graph altering its structure for a finite number of timesteps before coming to rest in an attractor. This class of behaviour invites comparisons with organic morphogenesis in which structures develop from a seed state but in which growth “naturally" comes to a halt. The experiment shows that, despite the state space for these growing graphs being unbounded, coming to rest in an attractor is quite common, and furthermore the “interesting" behaviour of halting growth is the most common attractor type. Systems with more possible node states reach this type of attractor more often; this may be a consequence of the way the system is set up, with structural alterations having a lower probability in systems with more states. The growth trajectories in the halting growth class are indicative of rich dynamics, with graph sizes changing irregularly rather than in a simple linear or exponential fashion.

Chapter 5

DGCA Dynamics

5.1 Introduction

Chapter 4 introduced a model of *Developmental Graph Cellular Automata* (DGCA) that are inspired by the rich global behaviour that can be generated from local interactions in CAs, as well as the dynamically growing structures that can be modelled using L-Systems and other developmental models. Combining the two ideas points towards dynamically growing graph structures in which each node's behaviour is determined by purely local information.

Development halts when the system enters an attractor, when the graph is isomorphic (conditioned on the node states) with a graph that has been previously seen in the course of development. Because it is a closed, deterministic system, it is impossible to escape such an attractor once it has been reached. Attractors can either be point attractors, in which neither the graph structure nor the node states change from one timestep to the next, or cyclic attractors, in which the graph cycles through a number of topologies and/or configurations of node states.

This chapter explores the dynamics of DGCA growth in more detail, focusing on the "halting growth" behaviour class identified in the last chapter. Instances of this behaviour involve a transient phase followed by an attractor cycle phase. These are used as dimensions of a behaviour space, which is then explored using novelty search. This pushes the system to exhibit more extreme or unusual dynamics. Several examples of the interesting behaviour found by the novelty search are then explored in more detail. Some of these are evocative of biological process such as self-reproduction.

5.2 Experiment

Of the five behaviour classes identified in Chapter 4, the halting growth behaviour appeared to be the most "interesting" in that the variety of growth trajectories suggested rich dynamics. Furthermore it represents a developmental process that comes to a halt *endogenously* without

either manual intervention (eg. halting growth after a given number of timesteps) or the use of parameters extraneous to the developmental process itself, such as an energy model. This behaviour invites comparisons with the self-regulating growth of some natural organisms, in which growth halts once maturity is reached.

Here, we conduct an experiment to see if more novel halting growth behaviours can be discovered using an evolutionary search than using random search. This is both a proof-of-concept that the system is amenable to evolution, and also illustrates the richness of possible behaviours of the system.

5.2.1 Novelty Search

Novelty search is an open-ended evolutionary search paradigm introduced by Lehman and Stanley (2011). All evolutionary search processes require a fitness function, and in most cases this involves an assessment of how well the individual (i.e. candidate solution) achieves some predefined goal. In novelty search the fitness function instead measures how different the individual is from previously assessed individuals. The objective is therefore not fixed at the start, and the "fitness" of an individual changes dynamically as the search proceeds. This can enable more complex behaviour to emerge over time. It is argued that this is often an easier way to achieve such complex behaviour than specifying it in a static fitness function from the start, which can lead to the search process becoming trapped in local fitness optima.

Novelty search has analogies with biological evolution, in which fitness functions are also not fixed but are often contingent on the behaviour of other individuals, giving rise to the concept of ecological niches. This dynamical and open-ended search process seems to give rise to the complexity that is seen in the natural world.

Novelty search is particularly appropriate for the exploratory work here, in which the aim is to discover what behaviour the system is capable of, but without any fixed objectives in terms of dynamics or graph structures.

5.2.2 Behaviour Space

In setting up a novelty search, one must define the behaviour space within which to assess novelty. Since we are presently interested in exploring the dynamics of the system, we define the behaviour space over the dimensions of transient length and attractor cycle length. These are commonly used metrics for analysing the behaviour of dynamical systems (see for example Wuensche et al. (1992)).

Both of these measures are unbounded: both the transient length and the attractor cycle length (if an attractor is found at all) could be arbitrarily large, since the graph may be arbitrarily large. To make the experiment tractable we set fixed limits of network size and timesteps allowed for development (both to 256). Individuals that do not find an attractor within these time and space limits are not discarded, but left in the results. Since this is a

Behaviour class	Transient len.	Attractor len.	max size
Death	$\leq max$	1	$\leq max$
Runaway	$\leq max$	—	$= max$
Slow	$= max$	—	$\leq max$
Static	$\leq max$	$1...max - T$	$= init$
Halting	$\leq max$	$1...max - T$	$\leq max$

Table 5.1: The different classes of behaviour and their corresponding ranges of possible values in behaviour space. The attractor cycle length can be up to the maximum number of timesteps ($max = 256$ here) less the number of steps that have already been used in the transient (T). In the "static" behaviour, the size of the graph does not change so the number of nodes remains the same as its initial value ($init$). The halting growth behaviour is least constrained in the behaviour values it can take, so novelty search should discover more of this type of behaviour.

frequently occurring behaviour, novelty search should naturally seek to move away from it and focus on systems that do find an attractor.

We include a third dimension to the behaviour space: the maximum size (number of nodes) that the graph reaches over the course of its development. Since the graphs can grow and shrink over time, we used the maximum size rather than the final size. This means we have usable numbers for individuals that do not find an attractor. Furthermore in the case of graphs where all nodes are removed (the "death" behaviour), recording the maximum size allows us to distinguish between graphs which grow to a large size before they dwindle to nothing, and those that only shrink.

Some examples of how the three features of our behaviour space can be used to identify different behaviour classes are shown in Table 5.1. We are most interested in the "halting growth" behaviour class, and this is where novelty search should push the system since there is most freedom in all three dimensions.

5.2.3 Microbial Genetic Algorithm

In order to implement novelty search, we use the Microbial Genetic Algorithm (MGA) (Harvey, 2011). This is a minimal implementation of a genetic algorithm which uses a steady-state population rather than a generational model. At each timestep, two individuals are randomly chosen from the population and their fitness is assessed. As novelty is the fitness criterion, a score is calculated as the mean distance of the individual to its nearest neighbours in the three dimensional behaviour space defined above. A larger distance gives a higher fitness.

The fitter individual remains, whilst the less fit individual is replaced by the "offspring" of the two. The MGA model is inspired by bacterial conjugation, in which genetic material is transferred horizontally between bacteria. Interpreted like this, the MGA model does not involve death and reproduction, but continual transfer of genetic material from the stronger to the weaker members of a steady-state population.

For the genetic crossover operation, the half of the weights in the SLP that constitute the update function of each individual are replaced by weights from the other parent. This is done

on a column-by-column basis: a randomly selected 50% of the columns in the weight matrix of the offspring are taken from each parent. As mentioned in Section 4.3.5, each column of the weights matrix can be interpreted as an individual "rule", selecting a particular outcome based on an assessment of the node's neighbourhood information vector.

After the crossover operation is performed, a mutation is applied, randomising 5% of the weights in the matrix. This relatively small degree of mutation was chosen through preliminary experiments (not reported here), and found to enable the discovery of new behaviour whilst not completely destroying the behaviour inherited from the parents.

5.2.4 Seed Graph Selection

The purpose of the novelty search experiment is to find growth rules (as encoded in the SLP weights) which produce novel behaviour. To make sure that new behaviour arises from the evolution of the rule rather than from having a different starting point, we use the same seed graph in each trial.

One feature of the present implementation of DGCA is how the structure of the seed graph influences the structure of the developed graph. For instance, a seed graph without cyclic connections can never develop into a graph *with* cyclic connections. This is because whenever a node is created, its incoming and outgoing edges are copied from its parent node: if there are no cycles in the graph at one timestep, they cannot be created at the next. This property of the system is discussed further in Chapter 6, which also introduces a more flexible method of node division.

Preliminary experiments indicated that using a very simple seed graph limited the range of dynamical behaviours. For this reason, the relatively complex seed graph shown in Figure 5.1 was selected. It is a ring of eight nodes, with unidirectional edges, and with every node having a self connection. The node states in the seed graph are set to a repeating sequence of the number of states in the system being used.

5.2.5 Experiment parameters

In our experiment, we implement novelty search using the MGA with a population size of 30, and 1,000 evaluations. Fitness is calculated as the average Euclidean distance from the five nearest neighbour points in the behaviour space. For the random search, we also use 1,000 evaluations. We run the random search and the novelty search once each for systems with 2 states to 8 states.

5.3 Results

The charts in Figure 5.2 compare random search with novelty search for systems with 6, 7, and 8 possible node states. They plot individuals in the behaviour space of transient length

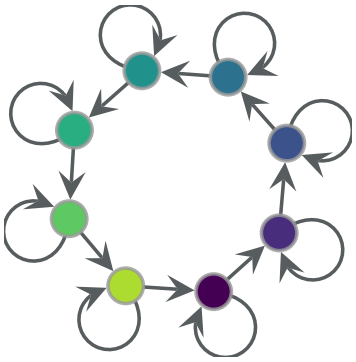


Figure 5.1: The seed graph used in the random search vs. novelty search experiment. The topology is a unidirectional ring of eight nodes, each of which has a self-loop edge. Node colours and numbers represent the state of the node. Node states in the seed graph are assigned sequentially using the number of states in the system. The graph shown is for an eight state system, so every node has a different state. In a four state system, the states would be repeated around the ring as: 1-2-3-4-1-2-3-4.

System States	Random Search	Novelty Search
2 State	18	20
3 State	29	33
4 State	40	42
5 State	34	39
6 State	44	45
7 State	30	54
8 State	50	57

Table 5.2: The number of 'voxels' that are populated when the behaviour space is quantised into 8 intervals along each of the three dimensions (transient length, attractor cycle length, and maximum size reached). This gives an overall total of 512 (8^3) voxels in the behaviour space. In all cases, novelty search achieves a greater spread of behaviour (populates more voxels).

and attractor cycle length, with colour used to represent the maximum number of nodes a graph reaches over the course of its development.

In all cases, novelty search seems to be able to explore more of the behaviour space. To quantify the greater spread of behaviour found with novelty search compared with random search, we quantize the behaviour space into 8 regions along each of the three dimensions and count the number of three-dimensional 'voxels' (out of a total of $8^3 = 512$) that are populated (this approach follows the CHARC method used in Dale et al. (2019)). The results for systems with 2 to 8 node states are shown in Table 5.2. In all cases, more of the behaviour space voxels are populated when novelty search is used, although the degree of improvement varies. It should be noted that there is a certain portion of the space that its unreachable, where the transient length plus the attractor cycle length is greater than the cutoff we have set for the number of iterations of the system (256 steps). This is highlighted on the charts in Figure 5.2 with a red line at the upper right indicating this unreachable portion of the space.

Exploring more of the behaviour space is the whole purpose of novelty search; here it

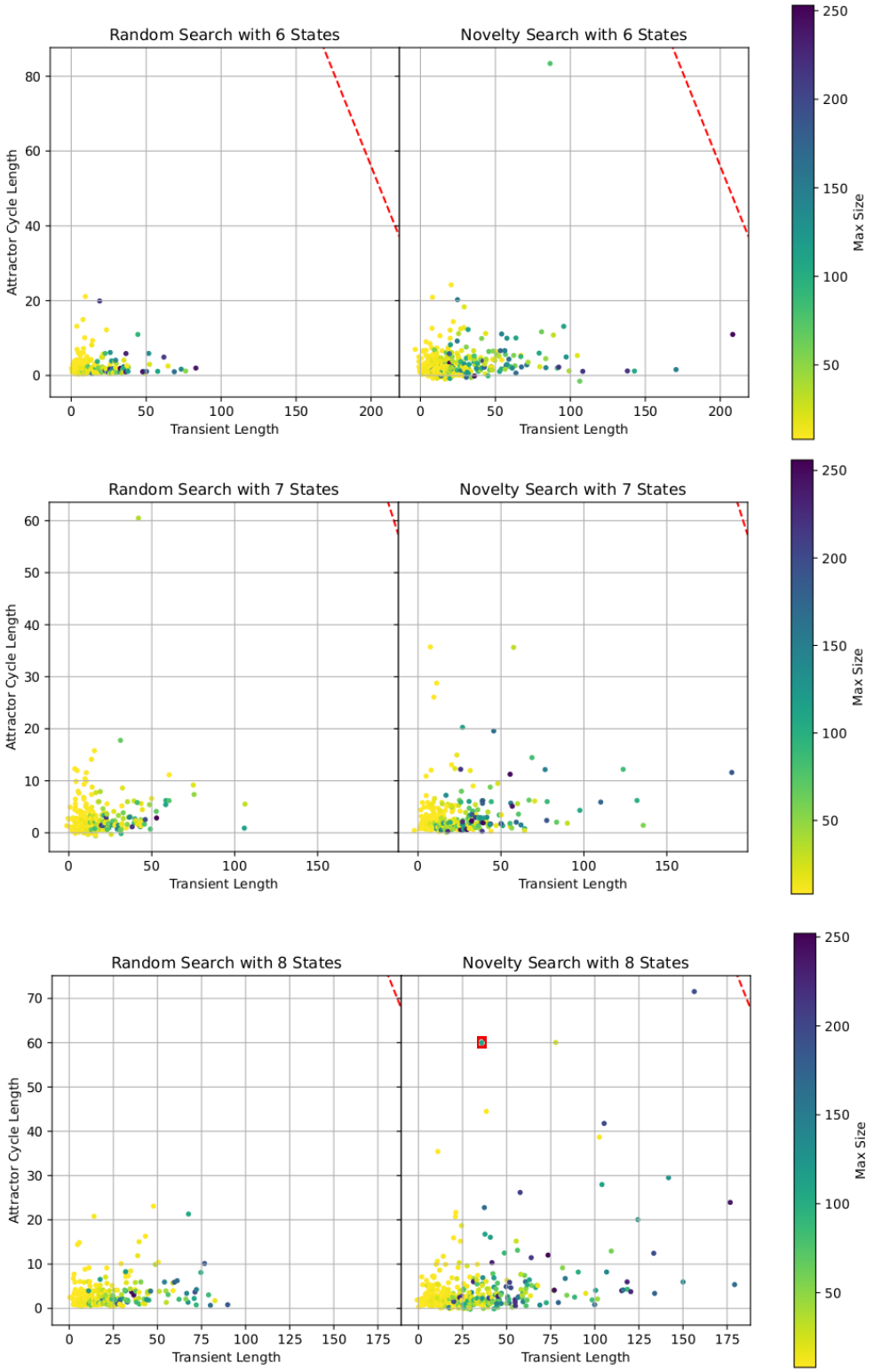


Figure 5.2: Charts showing random vs novelty search for 6, 7 and 8-state systems. Each point on the charts represents an individual's developmental behaviour in terms of transient length, attractor cycle length and maximum size reached. Only graphs which reached an attractor are plotted. Graphs in which all nodes were removed are also not plotted. The red dashed line in the upper right corner shows the limit of the explorable space, since development was arbitrarily cut off after 256 timesteps. The individual highlighted with a red box in the 8 state novelty search chart is shown in detail in Figure 5.3

also shows that the system can be evolved towards desired behaviour by using the mutation and crossover operators on the weights matrices of the SLPs which constitute the transition rules. This is an encouraging result as it may mean that goal directed evolution can be used to create systems which develop in particular ways. Such goals might include not only the attractor behaviour explored here, but also structural properties of the developed graphs themselves. This will be investigated in subsequent chapters.

Some of the behaviour found by novelty search is surprisingly complex, with long transients and attractor cycles, and rapid growth to a large size which nevertheless comes to a halt. Some individual examples of interesting behaviour are selected for further discussion below.

5.4 Interesting Behaviour

In the remainder of this chapter, we select some individual examples of "interesting" behaviour to examine in more detail. This provides anecdotal (rather than quantitative) evidence for the richness and variety of behaviour which can be produced by DGCA. This is similar to the "zoos" of interesting behaviour that have been compiled for Game of Life and other CA rules.

In the present system, both the seed graph and the "rule" (as encoded in the SLP) can be explored. This gives a huge parameter space to explore. This section does not aim to explore this space systematically or comprehensively, but rather to pick out some examples which indicate the range of behaviour the system can produce. In particular, we highlight a class of behaviour that involves the graph splitting separate components.

5.4.1 Graphs Dividing

Since nodes can be removed as well as added during the development process, it is possible for the graph to split into two or more components. In the present system it is not possible for disconnected components to join together again. Unlike some models of neural development (for example, Miller (2021)) there is no underlying spatial model by which nodes can "move towards" each other and connect. Nodes only "know about" their connected neighbours; once a graph has split, there is no possibility of the nodes in one component forming a connection with the nodes in the other. This mirrors some biological processes such as cell division or indeed reproduction of larger organisms.

When a graph divides, some of the components may enter an attractor, whilst others carry on developing. This behaviour is illustrated in Figure 5.3. The individual illustrated was found during the novelty search, and is highlighted in Figure 5.2. The overall graph has a transient of 35 steps and an attractor cycle of 60 steps. However, it results in three separate components. Looking at the space-time diagram one can see that each of the separate

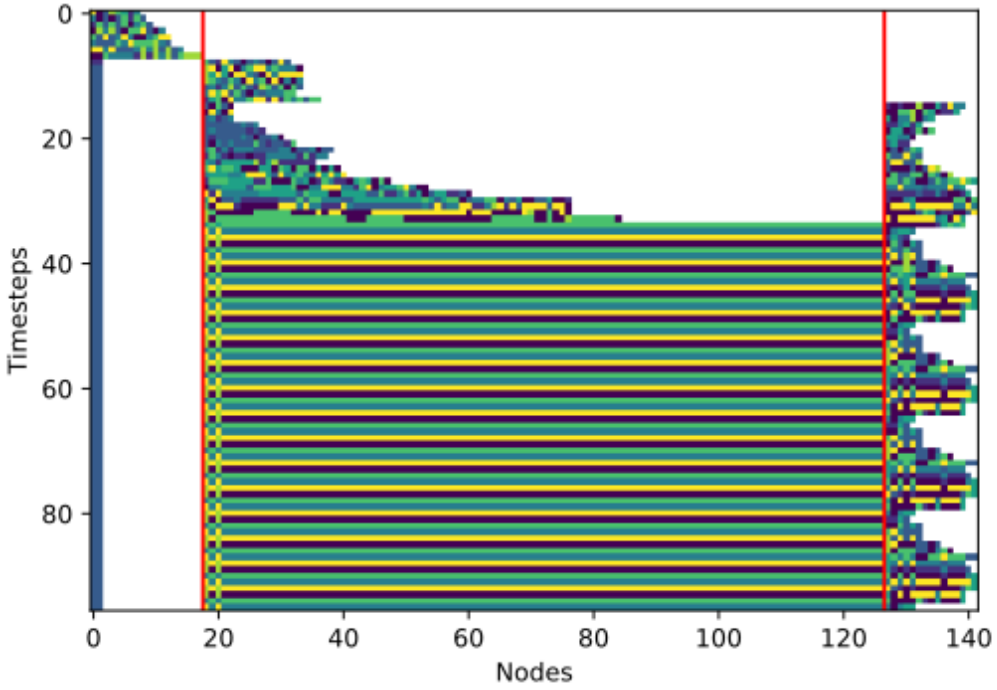


Figure 5.3: A space-time diagram of the individual highlighted in the 8-state novelty search in Figure 5.2. When nodes are added they are appended to the right hand end of the line, and when they are removed the gap is closed and the line shrinks. The timesteps proceed down the y-axis. When the graph divides into separate components, a red line is drawn and the second component is drawn to the right of it. In this case the graph divides twice: the initial seed graph of 8 nodes grows for 7 timesteps before splitting into two approximately equal sized components, one of which rapidly shrinks, one of which continues to grow. A third component is split off at about timestep 15.

components has a much shorter attractor cycle: the component on the left ends in a point attractor with two nodes; the middle component has a four step cyclic attractor with 109 nodes; the right component has a 15 step atrractor cycle which fluctuates in size between 4 and 15 nodes. Since 4 and 15 have no common factors, the two cycles of the middle and right components create an overall attractor cycle at the system level of 60 steps. The novelty search used above defines attractors only at the whole-system level, but this example makes it clear that individual graph components can have their own attractor behaviour.

The example in Figure 5.4 illustrates a case in which the graph splits in two, with one component quickly entering a point attractor, whilst the other continues to grow in a fluctuating manner. This individual is not plotted in the novelty search charts in Figure 5.2 because a system-level attractor is not found within the arbitrary size limit: this individual is classed as having runway growth. It is possible that a system-level attractor would be found if it was run for longer. A complementary visualisation of this behaviour is shown in Figure 5.5, which shows the size of each graph component over time. This example illustrates that even when an overall attractor is not found, individual graph components may enter an attractor. Although the classification system used here places this into the "uninteresting" class of runaway growth, the behaviour shown is relatively complex.

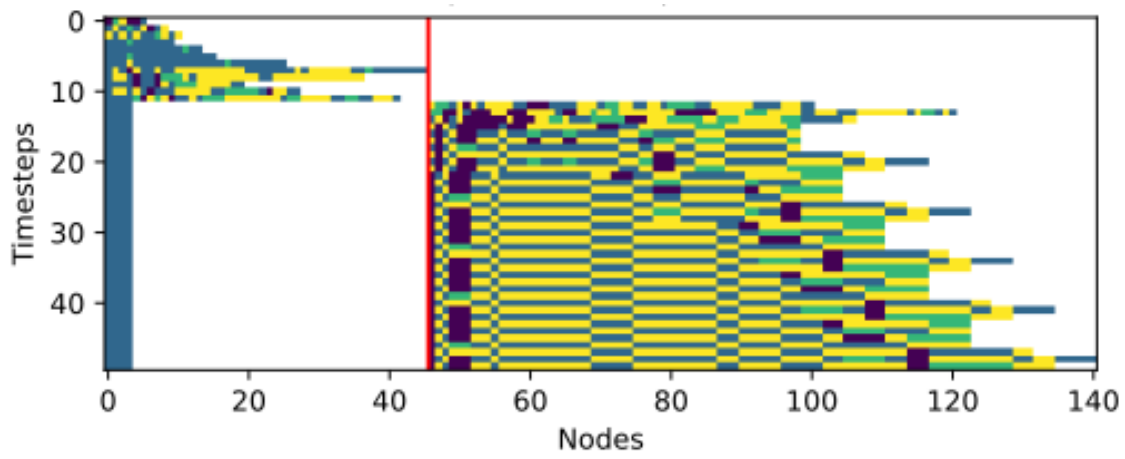


Figure 5.4: A space-time diagram for a four-state system in which the graph splits into two components at the 12th timestep of development. The component shown on the left settles into a four node point attractor, whilst the component on the right continues to grow with a fluctuating rhythm. The diagram is arbitrarily cut off at 50 timesteps; it is possible that the right hand component would settle into an attractor if the system was run for longer.

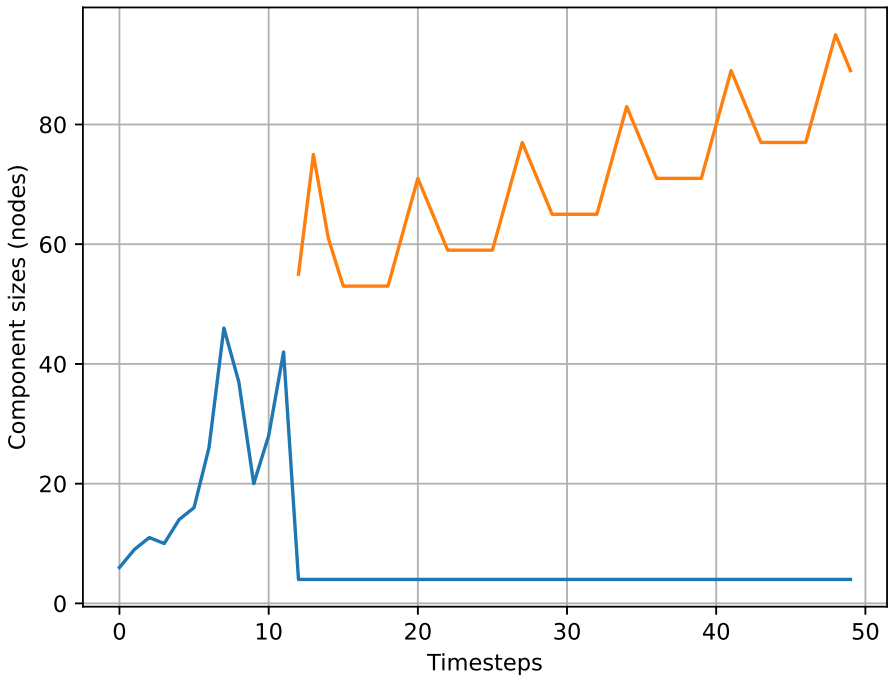


Figure 5.5: A different visualisation of the system shown in Figure 5.4, showing the sizes of the two graph components. The diagram is arbitrarily cut off at 50 timesteps.

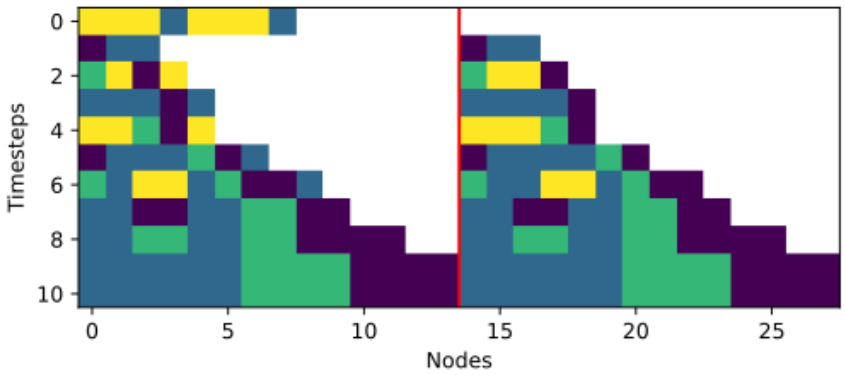


Figure 5.6: A space-time diagram for a four-state system in which the eight node seed graph immediately splits into two identical three node graphs due to the removal of two nodes. Each component traces the same course of development for eight more steps before reaching a point attractor. Note that the ordering of nodes in each half is arbitrary and differs somewhat (so the colours are not in the same order), but the two graphs are indeed isomorphic.

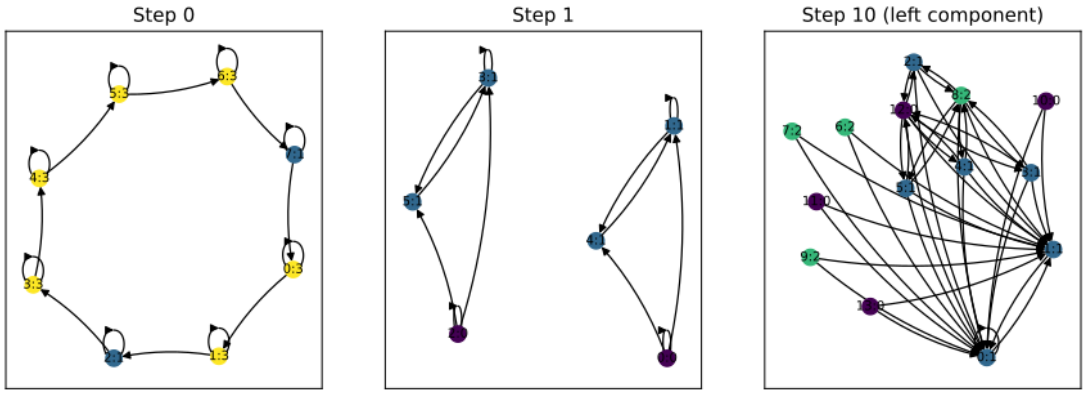


Figure 5.7: The actual graph developed in the space-time diagram in Figure 5.6, shown at step 0 (the seed graph), step 1 (when it splits into two identical components), and step 10 (when each component has reached a point attractor; only one component is shown).

Chapter 8 explores the behaviour of the separate components of graphs that have split, by continuing their development independently. This includes behaviour where a seed graph grows to a certain point and then part of it splits off to create a new seed graph identical to the original. Such self-replicating behaviour has clear biological parallels. Something approaching this behaviour is shown in Figure 5.6. Here the seed splits into two identical potions, each of which grow for several more steps before reaching point attractors. This is not true self-replication as seeds do not continue to be produced, but rather a one-off "twin development" as the initial seed has split into identical portions. This system is further illustrated in Figure 5.7, which shows the seed graph splitting into two components in the first step of development, and the final graph that each component ends up as, in a point attractor.

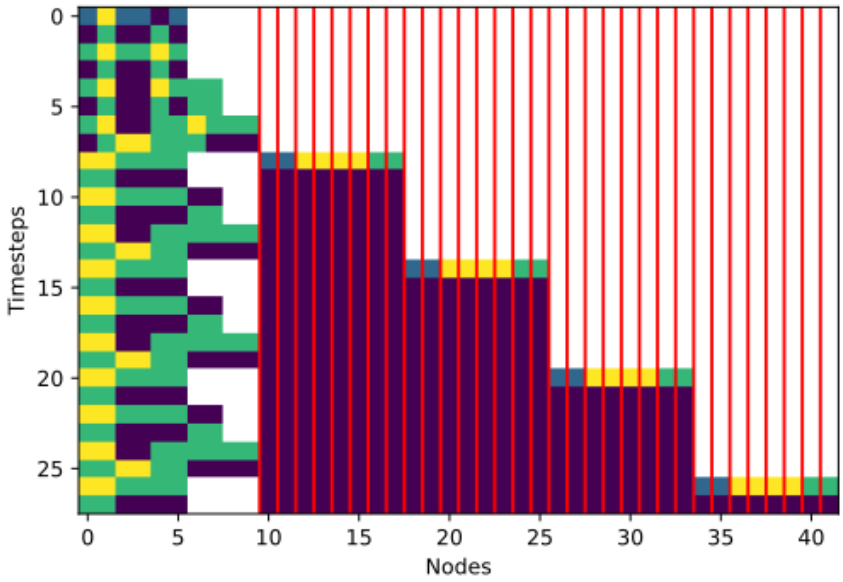


Figure 5.8: A space-time diagram for a four-state system with interesting periodic behaviour. The largest component of the graph (shown on the left) quickly settles into a six step cycle. However, at the beginning of each cycle, it throws off eight singleton nodes, each of which changes state once before remaining static. As before, a red line is drawn between disconnected graph components. The seed graph in this example is a six-node Kautz graph rather than the eight node ring used in the novelty search experiment. This behaviour was found during an earlier random search.

5.4.2 Periodic Detachment

Whilst the experiment reported in this chapter does not find the self-reproducing behaviour described above, several examples have been found in which elements are detached from the main body of the graph in a periodic cycle. An example is shown in Figure 5.8. The system as a whole has not found an attractor, since the overall number of nodes continues to increase. However, if one disregards the singleton nodes that are cast off, the main graph component has a six-step attractor cycle. This behaviour is reminiscent of a "glider gun" in Game of Life (Gardner, 2001), although unlike a glider gun, the emitted elements remain inert. Figure 5.9 shows the main graph component when it is at the smallest size in its cycle (six nodes).

5.5 Conclusion

This chapter has shown that the growth dynamics of DGCA are evolvable by using crossover and mutation operations on the weight matrices of the SLPs that define the update rules of the system. This was demonstrated using novelty search rather than goal-based evolution, pushing the system into more extreme areas of a behaviour space defined by the transient length, attractor cycle length and graph size. The experiment also demonstrates that the system is capable of a wide range of complex behaviour.

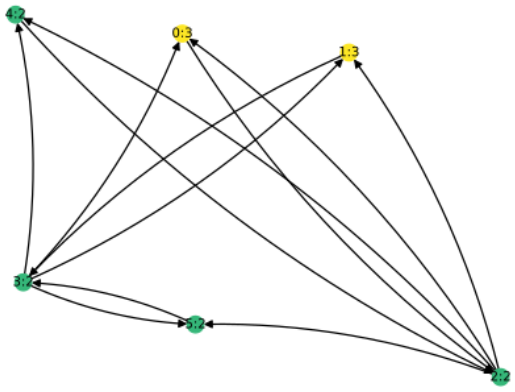


Figure 5.9: The main graph component from the system shown in Figure 5.8, shown at the point in its cycle where it has six nodes. Like Gosper's glider gun in Game of Life, this is a relatively simple structure which emits elements at a certain point in its cycle.

The selected examples discussed highlight that some of the observed complexity comes from the different behaviour of disconnected graph components. Even if the whole system appears to have a very long attractor cycle, or indeed if it does not find an attractor at all, individual components can have their own shorter attractor cycles. This observation leads to the the investigations of Chapters 8 & 9.

Chapter 6

DGCA Formalism

6.1 Introduction

The DGCA system as outlined in Section 4.3 shows interesting dynamical behaviour but has various shortcomings in terms of the range of graph topologies it is capable of producing. This chapter reviews these deficiencies and introduces a new version of the model which is more flexible and expressive. The system introduced in Chapter 4 is be referred to as DGCA v1, and the system introduced in this Chapter (and used in the rest of this thesis), as DGCA v2.

6.1.1 Motivation

A possible use-case for DGCA is as a graph generation algorithm. As noted in Section 2.6, there are many existing graph construction algorithms such as the Erdős and Rényi (1959) random graph model and the and preferential attachment model of Albert and Barabási (2002)), which specify various hyperparameters for graph generation such as edge probabilities, but involve randomness in creating the actual structure. DGCA, by contrast, is fully deterministic, driven by the parameters of the update function.

The mapping between the parameter space (the neural network weights) and graph topologies is highly complex, since it involves a growth process which takes place over multiple timesteps. Nevertheless, it is desirable that the full range of the parameter space should map onto as much as possible of the space of graph topologies. Ideally, it should be possible for the system to create *any* graph, provided the model is suitably parametrised.

As previously mentioned, it is not only the parameters of the neural network which determine what the final graph will look like, but also the starting graph (referred to as the "seed graph"). DGCA has this in common with conventional CA (and indeed most dynamical systems), in which both the update rule and the starting state contribute to the final state (or the state after a fixed number of timesteps). If we would like the system to be able to reach any graph in the universe of all directed graphs, we must first specify an allowable

set of seed graphs. Clearly if we allowed any seed graph, then the question of whether the system can generate any graph would become trivial: it could simply use the target graph as the seed graph and apply zero developmental steps. A fairer test for the expressiveness of the system would be to ask which final graphs are reachable from a very limited set of seed graphs, which should themselves be as small and simple as possible. Taking this to the extreme, it would be desirable if as many graphs as possible could be reached from the very simplest seed graph, consisting of a single node with no edges. This is the seed graph which will be used throughout this chapter and in most of the rest of the thesis. By contrast, a relatively complex seed graph of eight nodes was used for the experiments in Chapters 4 & 5 because of the limitations of DGCA v1 in creating novel structure. The reasons for this are explored in Section 6.2.

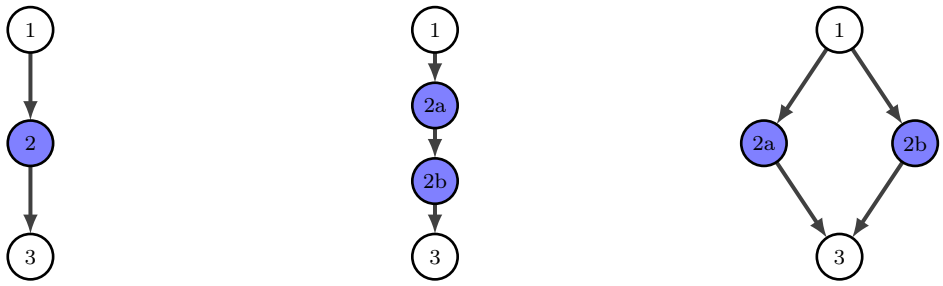
6.1.2 Chapter outline

In this chapter, Section 6.2 explains in detail why certain classes of graph structure cannot be created with DGCA v1. Section 6.3 outlines the design principles and requirements of an improved model, before Section 6.4 introduces the model itself, DGCA v2. This section includes implementation details showing how update steps are performed using matrix multiplication and Kronecker tensor products, making the system fast to run on modern hardware. Examples of the capabilities of DGCA v2 are shown in Section 6.5. Finally, Section 6.6 presents a proof that the model is in fact capable of creating *any* graph starting from a single-node seed graph.

6.1.3 A note on terminology

Some terminology which was introduced in Chapter 4 is recapitulated here for clarity.

- **Possible node states**: The set of possible states a node in the system can be in, Σ. This is a hyperparameter of the system, and we refer to systems with different numbers of possible node states. In most of the rest of the thesis, three possible node states will be used, (variously represented as $\Sigma = \{0, 1, 2\}$, $\Sigma = \{A, B, C\}$, or using the one-hot representation: $\Sigma = \{[0, 0, 1], [0, 1, 0], [1, 0, 0]\}$).
- **Node state**: the state of an individual node, which is selected from from a finite set of states: $\sigma_g(u) \in \Sigma$, where u is a node in graph g and σ_g is mapping from nodes to states.
- **Graph state**: we use this term interchangeably with **system state** to mean the overall state of the graph, comprising both its topology and the states of all the individual nodes. Two graphs which are isomorphic *conditioned on* node states are said to have the same graph state. This is formalised in Definition 1 of vertex-labelled isomorphism classes.



(a) The original graph (b) Sequential division (c) Parallel division

Figure 6.1: The two cell division types used in the Cellular Encoding model of Gruau and Whitley (1993). (b) shows sequential division in which the first child cell($2a$) gets the parent cell's incoming edges, and the second ($2b$) gets the parent's outgoing edges and a new edge is created from the first to the second. (c) shows parallel division, in which both child cells inherit both the incoming and the outgoing edges of the parent cell, and no edges are created between the two child cells.

- **(Graph) state space**: The space of all possible graph states. Note that for a fixed topology this is a finite space which comprises all of the possible assignments of node states to nodes. For a fixed size (number of nodes) graph, this is a much larger but still finite space comprising all of the isomorphism classes of graphs of that size, conditioned on assignment of node states. In both cases, the larger the graph, the larger the state space. If we allow graphs of any size the state space is unbounded.

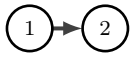
6.2 Shortcomings of DGCA v1

6.2.1 Node division

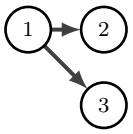
The limited abilites of DGCA v1 to create novel structure stem from the way it implements node division. When a node divides in DGCA v1, the two child nodes inherit the incoming and outgoing edges of the parent node. This corresponds to a "parallel split" in the Cellular Encoding system of Gruau et al. (1996) (see Figure 6.1c). In terms of the adjacency matrix, it involves simply copying the row and column of the parent node and appending them to the end of the matrix (see the example in Equation 4.22).

If the seed graph is already relatively complex, the combination of parallel division and node removal can still create a wide range of graph structure. However, if we start with a very simple seed graph, there are severe constraints on what kind of graphs we can reach using DGCA v1. It is for this reason that relatively large and complicated seed graphs were used in Chapters 4 & 5. Although the experiments in those chapters were focused on the dynamics of the developmental process rather than the structure of the graphs produced, it was empirically found that a wider range of dynamics were possible when using a more complex seed graph as this allowed more of the space of possible graphs to be explored.

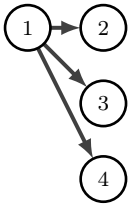
An illustration of the point that DGCA v1 can only produce certain topologies from a simple seed graph is given in Figure 6.2. It is easy to see that the structure that is produced



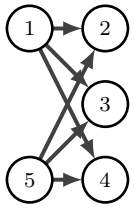
(a) A simple seed graph



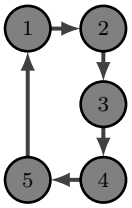
(b) Node 2 divides into 2 & 3



(c) Nodes 3 divides into 3 & 4



(d) Nodes 1 divides into 1 & 5



(e) It would *not* be possible to develop this graph from the seed in (a)

Figure 6.2: Using DGCA v1, a seed graph of two nodes with a single edge makes it easy to create something that looks like a two-layer feedforward network, but it would not be possible to create a unidirectional ring as shown in (e). There is no way for the edges $2 \rightarrow 3$, $3 \rightarrow 4$, or $5 \rightarrow 1$ to be created.

in Figure 6.2d was in a sense *already present* in the structure of the seed graph (Fig. 6.2a): the two nodes connected by a unidirectional edge became two *layers* of nodes connected by unidirectional fan-out / fan-in edges. Similarly, if we had started with a chain of three nodes, it would be trivial to develop a three-layer feedforward network. However, it would be impossible to go from a unidirectional chain of any number of nodes to a ring of nodes as shown in Figure 6.2e.

As mentioned it is still possible to develop a wide range of structures using DGCA v1 provided we use quite a large and complex seed. There must already be enough variety of structure in the seed graph to produce the required variety in the final graph. However, the system is inadequate for creating novel structural forms during the developmental process. In particular, if a very simple seed graph is used, the space of possible topologies for the final graph is significantly constrained.

The reason for this shortcoming is that when a node divides, all of its incoming and outgoing edges *must* be cloned in the child. Furthermore, it is impossible for edges to be created *between* child nodes (eg. between nodes 2 and 3 in Fig. 6.2b), or between a parent and a child node (ie. the sequential division of Figure 6.1b).

These constraints make it difficult for repeated structure to be produced. As illustrated in Figure 6.3, even if all nodes of a particular subgraph are duplicated in DGCA v1, this does not result in the duplication of that subgraph structure, and in fact leads to quite a different structure (Fig. 6.3b). DGCA v2 allows more detailed specification of which edges should be copied or changed. Amongst other things, this allows a subgraph or motif to be reproduced as a whole, as shown in Figure 6.3c. This is desirable from the point of view of biological

realism, as many biological networks such as neural networks tend to have repeated, modular structure.

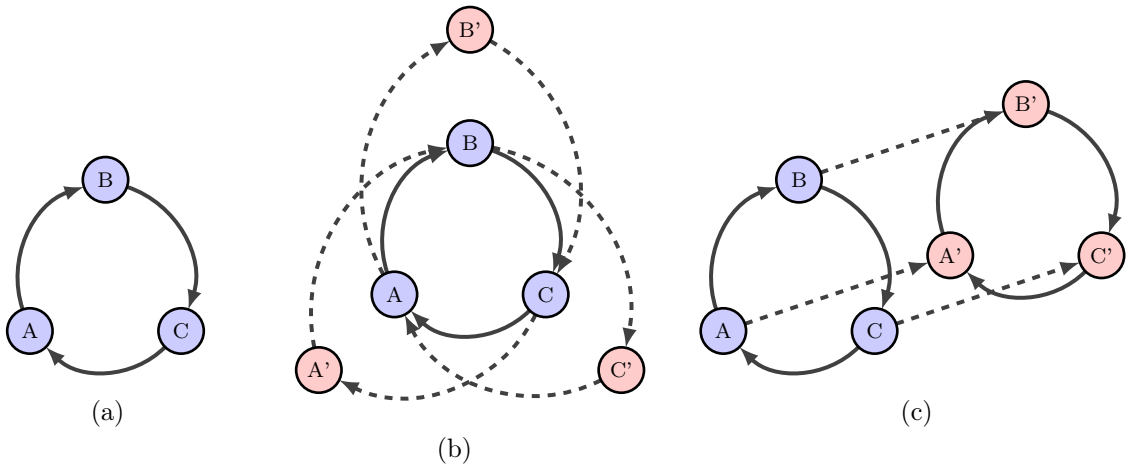


Figure 6.3: The original graph is shown in (a). In (b) all three nodes divide and clone their parents' edges, which does not result in the ring motif being replicated (DGCA v1 approach). In (c) the ring motif is reproduced as a whole. Both outcomes are possible using DGCA v2, but only the first is possible in DGCA v1.

6.2.2 Combination of State and Action decisions

DGCA v1 uses a single SLP to control node state changes and node "actions" (addition or removal of a node). This has the side effect of meaning that the addition or removal of a node is less likely in a system with more possible node states Since the SLP is used to choose just one option (using arg max) out of changing to one of the possible states *or* adding or removing a node, it is *a priori* less likely that one of the structural alteration outcomes will be chosen when there are more states to choose from.

The actual likelihood of each option being chosen depends on the particular SLP weights, but if we average over the behaviour of an ensemble of systems with random SLP weights, then we will find that node actions are chosen less frequently in a system with more node states. This in turn means there will be fewer structural alterations to the graph. Fewer structural alterations in turn mean that the system is more likely to enter an attractor. In the limit, with no structural alterations being made, the system would inevitably enter an attractor if it was run for long enough, because the state space would be finite. Each time the graph's structure changes, it is effectively pushed into a different state space. When nodes are added, the state space gets larger. It is for this reason that making fewer structural alterations corresponds to a higher likelihood of entering an attractor.

This intuition is confirmed by the experimental results in Section 4.4.3. The system is more likely to enter an attractor (the "halting growth" behaviour class defined in Chapter 4)

when more node states are used. Whilst it may be desirable to be able to tune the likelihood of structural alterations being made, it is inelegant for this tuning to be done via the number of node states. For a fixed graph size and topology, having more possible nodes states obviously means a larger state space of graph configurations. If no structural alterations were made at all, this means that the fixed size graph in a system with fewer node states would (all else being equal) enter an attractor more quickly. In DGCA v1 there are therefore two forces working against each other - a system with fewer node states would enter an attractor more quickly *if* no structural alterations were made, but having fewer node states in fact makes it more likely that structural alterations *will* be made. The effect of having different numbers of node states in a system is therefore difficult to interpret and adds biases to the system in an opaque manner.

A better design is for the choice of "action" (addition or removal of nodes) to be independent from the choice of node state change. This makes it possible to tune the probability of nodes being added or removed independently from the node state change decision. This tuning is done entirely through the SLP parameters. This means the number of possible node states no longer biases the system in favour or against making structural alterations to the graph. We implement this approach in DGCA v2.

6.3 Design Principles

Any model is inevitably the result of some arbitrary design decisions: what level of detail should be included in the model; how many degrees of freedom it should have; at what level of abstraction it should operate. There is often a balance to be struck between expressiveness and simplicity of a model. Generally the aim is to maximise expressiveness whilst minimising model complexity: in the words of Leibniz, using "the simplest means to create the richest effects" (Johns, 2023). The same criterion is used in the natural sciences when creating a model of an observed phenomenon: a better model is one which explains more, more economically (Deutsch, 1998).

Although the present work is *inspired* by biology, it does not attempt to model the actual mechanisms of biological development. In common with much of Artificial Life research, it aims to create *ab initio* a completely abstract system which exhibits some *life-like* properties, as set out in Section 1.3. As such, the conventional modelling relation which exists in the natural sciences does not apply here. We cannot test the explanatory power of the model against some observed ground-truth.

However, the "quality" of the model can still be evaluated in terms of internal consistency and expressive power. With this in mind, we can state several guiding principles which should be used for devising the model. These include:

1. All "decisions" about nodes' changes of state or choices of action should be taken locally

at each node, using only local information. In other works there should be no global controller. This is a *sine qua non* of emergent behaviour - global behaviour cannot be described as "emergent" if it is the result of global control.

2. As a model based purely on graphs, only information present in the graph should be used. There is no spatial embedding of the graph. The distance between nodes can only be expressed by the path length between them in the graph. The local neighbourhood of a node therefore consists only of nodes with which it shares an edge.

3. When a new node is created, in an analogy of cell division, it shares the same *potential* local neighbourhood as its parent. Since there is no spatial embedding we cannot say that a new node is created adjacent to its parent in space, as in some models such as Avida (Adami, 1998). However, we can say that it should be adjacent to its parent node in some meaningful sense within the graph. We therefore say that it may form connections with its parent and with nodes that are local to its parent. However, it is not obligatory for it to make all (or any) of these connections, so the local neighbourhood of the parent is only the *potential* neighbourhood of the child nodes.

4. Another consequence of the lack of spatial embedding is that if two subgraphs become entirely detached from each other, there is no way for them ever to re-join or interact in any way.

These guiding principals all flow from the desire to implement a purely graph-based system using only information local to nodes.

6.4 Model

This section introduces a family of models which we collectively refer to as DGCA v2. Several different particular implementations are possible, including the size of the neural networks used and the precise timing of node state and action updates.

Two slightly different implementations of DGCA v2 are used in Chapters 7 & 8, which are further explained in those chapters. Not all of the possible features in this section will be used in subsequent chapters, but they are described here for completeness.

6.4.1 Graph Definition

To summarise Section 4.3, a single graph in the DGCA system is represented by the tuple:

$$g = (V_g, E_g, \sigma_g) \tag{6.1}$$

where V_g is a set of vertices (or nodes),

$$E_g \subseteq V_g \times V_g$$

is a set of directed edges, and σ_g is a mapping from vertices in V_g to a finite set of states Σ. Every graph produced by a run of the system has vertices with states from the same set Σ. The size of this set of possible node states is s:

$$s = |\Sigma| \tag{6.2}$$

where $|\ldots|$ indicates cardinality. Individual states within this set are denoted using subscripts $1...s$. Thus we may say $\sigma_g(3) = \Sigma_1$ to mean that vertex number 3 in graph g is in the first state.

The number of vertices in a particular graph is n:

$$n = |V_g| \tag{6.3}$$

Moving to the practical implementation of the system, we describe a graph fully using two matrices, the adjacency matrix $\mathbf{A}$ with dimensions $n \times n$, and the state matrix $\mathbf{S}$ with dimensions $n \times s$. The adjacency matrix is simply a binary matrix indicating the presence of (directed) edges between nodes:

$$\mathbf{A}_{i,j} = \begin{cases} 1, & \text{if } (i,j) \in E_g \\ 0, & \text{otherwise} \end{cases} \tag{6.4}$$

The state matrix uses a one-hot encoding for each row to show which of the possible states that node is in:

$$\mathbf{S}_{a,b} = \begin{cases} 1, & \text{if } \sigma_g(a) = \Sigma_b \\ 0, & \text{otherwise} \end{cases} \tag{6.5}$$

The formal mathematical description of a graph in Equation 6.1 is therefore equivalent to the practical definition we use based on these matrices:

$$g = (V_g, E_g, \sigma_g) \Leftrightarrow (\mathbf{A}, \mathbf{S}) \tag{6.6}$$

Together, the $\mathbf{A}$ and $\mathbf{S}$ matrices fully define a graph in the system. For the example graph

shown in Figure 6.4 the **A** and **S** matrices can be defined as follows:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{6.7}$$

The ordering of rows and columns in the adjacency matrix follows the node numbering in Figure 6.4 (i.e. node 1 is represented by the first row/column in the matrix). The same goes for the ordering if the rows in the state matrix. The one-hot encoding within the rows follows the lexicographic ordering of node states (i.e. states a, b and c are represented by rows $[1, 0, 0]$, $[0, 1, 0]$, and $[0, 0, 1]$ respectively. In general the ordering of the rows and columns of **A** and the rows in **S** is arbitrary and does not make a difference to the graph that is represented.

6.4.2 Update function

The basic form of the update function f is:

$$f : G \to G \tag{6.8}$$

where G is the set of all possible graphs (following the definition of graphs in the DGCA system shown in Equation 6.1). More concretely, since we are defining the graphs using the adjacency and state matrices, the update function has the form:

$$(\mathbf{A}_{t+1}, \mathbf{S}_{t+1}) = f(\mathbf{A}_t, \mathbf{S}_t) \tag{6.9}$$

where the time subscripts indicate that one application of the update function performs one development step.

6.4.3 Neighbourhood Information Aggregation

The vector of each node's local information forms the input to the various neural networks which implement the update function. The first part of an update must therefore involve aggregating this local information. This process was briefly covered in Section 4.3.3, but is described more fully here. Neighbourhood information aggregation is done simultaneously for all nodes. This simply involves counting the number of neighbouring nodes in each state. In the following description we will use the term "subject node" to describe the node whose neighbourhood information is being gathered. There are various different approaches to neighbourhood information aggregation depending on:

- how / whether we wish to include the subject node's own state;

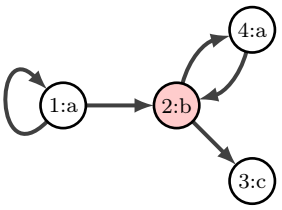


Figure 6.4: An example graph. Each node is referred to by its number. Each node also has a state from $\{a, b, c\}$. For example node 1 is in state a. Node 2 in particular (highlighted in red) is referred to as the "subject node" in the discussion of neighbourhood information aggregation below.

- whether we wish to include nodes connected by outgoing *and* incoming edges in the neighbourhood;
- the radius of the neighbourhood (number of steps from the central node).

The most basic option is to count the number of incoming nodes of the subject node in each state. The value of this at a particular node v, c_v^{in} is given by:

$$c_v^{in} = \sum_{u \in \mathcal{N}_{in}(v)} \sigma(u) \tag{6.10}$$

Switching to the matrix notation which is used in the rest of the chapter, this can be calculated for all nodes as:

$$\mathbf{C_{in}} = \mathbf{A}^T \cdot \mathbf{S} \tag{6.11}$$

The size of $\mathbf{C_{in}}$ is $n \times s$, with each row representing the counts of incoming nodes in each state. For the example graph in Figure 6.4:

$$\mathbf{C_{in}} = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{6.12}$$

Node 2 (highlighted in red) has two incoming neighbours in state a (nodes 1 and 4).

We can optionally also count the number of outgoing neighbours in each state.[1] We refer to these counts as $\mathbf{C_{out}}$:

$$\mathbf{C_{out}} = \mathbf{A} \cdot \mathbf{S} \tag{6.13}$$

In the example graph:

$$\mathbf{C_{out}} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{6.14}$$

[1] As mentioned in Chapter 4, edge direction is not intended to represent the flow of information in DGCA, but rather provides a level of anisotropy by treating outgoing and incoming neighbours differently.

This time the highlighted node has one outgoing neighbour in state a and one in state c (nodes 4 and 3) as indicated on the second row of the matrix.

If we wish to count *both* the incoming and outgoing neighbour state counts (treating them separately), we can simply horizontally concatenate $\mathbf{C_{in}}$ and $\mathbf{C_{out}}$. If we also wish to provide the subject node's own state to the update functions, we can also concatenate $\mathbf{S}$. We refer to the full matrix provided to the update function simply as $\mathbf{C}$. Each row in $\mathbf{C}$ is the vector of information about one node. In the default implementation, we use the triple concatenation described above, so:

$$\mathbf{C}_{(n\times 3s)} = \left[\mathbf{S}_{(n\times s)}, \mathbf{C_{in}}_{(n\times s)}, \mathbf{C_{out}}_{(n\times s)}\right] \tag{6.15}$$

Here, the matrix dimensions are given as subscripts, and $[\dots,\dots]$ indicates horizontal concatenation of matrices. For the example graph,

$$\mathbf{C} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{6.16}$$

Not all of $\mathbf{S}$, $\mathbf{C_{in}}$, $\mathbf{C_{out}}$ need to be included in $\mathbf{C}$. Different variants of the model may be generated by using only some of these. For example we could implement a system where only the subject node's state is taken into account ($\mathbf{C} = \mathbf{S}$), or one where the incoming and outgoing neighbours are considered but the subject node's state is not ($\mathbf{C} = [\mathbf{C_{in}}, \mathbf{C_{out}}]$), or any other combination of information. Using the terminology of CA, including $\mathbf{S}$ leads to outer-totalistic rules, while omitting it leads to totalistic rules. It is also possible to increase the neighbourhood radius by taking powers of the adjacency matrix. For example, the number of nodes in each state which are two (outgoing) steps away from the subject node could be calculated as $\mathbf{A}^2 \cdot \mathbf{S}$. This possibility is not explored in the present work.

Since the rows of $\mathbf{C}$ will be used as inputs to the neural network, it is desirable to normalise them, since otherwise the values may become very large (eg. for a node with high degree connected to many other nodes in the same state). We do this normalisation by dividing by the row maximum (unless this is zero, in which case the row is left unchanged):

$$\tilde{\mathbf{C}}_{ij} = \begin{cases} \frac{\mathbf{C}_{ij}}{\max_k \mathbf{C}_{ik}}, & \text{if } \max_k \mathbf{C}_{ik} > 0 \\ 0, & \text{if } \max_k \mathbf{C}_{ik} = 0 \end{cases} \tag{6.17}$$

Abstracting away from the exact implementation chosen, we will refer to the neighbourhood information aggregation step as the function Agg:

$$\mathbf{C} = \mathrm{Agg}(\mathbf{A}, \mathbf{S}) \tag{6.18}$$

The neighbourhood aggregation step is also used in Graph Neural Networks (GNNs), where various different approaches exist. One class of GNNs uses spectral filtering of graphs using the graph Laplacian to combine node level information (eg. Bruna et al. (2013) and Defferrard et al. (2016)). A simplification of this approach was used in DGCA v1 as introduced in Section 4.3.3, using the graph Laplacian to weight neighbourhood information. The motivation for this was to find an efficient way of combining the subject node's own state with that of its neighbourhood, so that outer-totalistic rules could be implemented using a smaller vector of values ($2s$ values for each node rather than the $3s$ of Eq. 6.15). This in turn means a smaller input layer for the SLP, resulting in fewer weights and a smaller parameter space. However, concatenating the subject node's own state, as shown in Equation 6.15 provides a more complete description of the neighbourhood and allows the neural network to distinguish better between different situations.

As a trivial example of this, note that the Laplacian approach used in DGCA v1 weights the subject node's state by its degree and the connected nodes' states by -1 (see Eq. 4.12). Therefore in a two state system with the subject node in state $[0, 1]$ (using one-hot encoding), and two neighbour nodes in state $[0, 1]$, the information would be combined as $(2 \times [0, 1]) - [0, 1] - [0, 1] = [0, 0]$. This would fail to distinguish the same situation but with three neighbour nodes in the same state ($(3 \times [0, 1]) - [0, 1] - [0, 1] - [0, 1] = [0, 0]$). On the other hand, using the concatenation of the subject node's state with the sum of the neighbours' states would give $[0, 1, 0, 2]$ in the first case and $[0, 1, 0, 3]$ in the second. (Note that in this simple example edge direction is disregarded). This increased descriptiveness comes at the cost of needing a larger input layer for the neural network and therefore more parameters. However, as explained in the next section, the number of parameters is still relatively small.

6.4.4 Multilayer perceptrons

The output from the neighbourhood information aggregation step (namely the matrix $\mathbf{C}$) is then passed into two multilayer perceptrons (MLP). In the implementation of DGCA v1, one single-layer perceptron (SLP) was used. Here we use two separate MLPs, one for node state update and one for the node action choice. This is to avoid the biases introduced by using a single network, described in Section 6.2.2. In this section we refer to MLPs rather than SLPs simply to make the model general. The number and size of the hidden layers of the MLPs are hyperparameters of a particular implementation, and of course it is possible to have no hidden layers (ie. SLPs). It is hypothesised that using MLPs with at least one hidden layer may improve the expressiveness of the model, due the the Universal Approximation Theorem (UAT) which states that an MLP with a single hidden layer (containing enough units) can approximate any function (Cybenko, 1989). These comes at the cost of the system having more parameters (MLP weights and biases) which increases the parameter search space so may make evolutionary search more difficult.

We use the standard definition for a multiplayer perceptron:

$$\mathbf{h}^0 = \mathbf{x}, \quad \mathbf{h}^l = \sigma\left(\mathbf{W}^l \mathbf{h}^{l-1} + \mathbf{b}^l\right) \quad \text{for } l = 1, 2, \ldots, L \tag{6.19}$$

where L is the number of layers; $\mathbf{W}^l$ and $\mathbf{b}^l$ are the weights and biases at layer l respectively; σ is the activation function (assumed to be a sigmoid function such as the hyperbolic tangent in the default implementation); $\mathbf{x}$ is the initial input vector; $\mathbf{h}^l$ is the output vector from layer l (so $\mathbf{h}^L$ is the final output).

In this formulation, $\mathbf{x}$ would be the neighbourhood information vector for a single node. In fact we can run the MLPs for all nodes at once using the matrix $\mathbf{C}$. Hence the node state and action choice MLPs respectively are defined as:

$$\mathbf{H_S}^0 = \mathbf{C}, \quad \mathbf{H_S}^l = \sigma\left(\mathbf{H_S}^{l-1} \cdot \mathbf{W_S}^l + \mathbf{b_S}^l\right) \quad \text{for } l = 1, 2, \ldots, L_S \tag{6.20}$$

$$\mathbf{H_A}^0 = \mathbf{C}, \quad \mathbf{H_A}^l = \sigma\left(\mathbf{H_A}^{l-1} \cdot \mathbf{W_A}^l + \mathbf{b_A}^l\right) \quad \text{for } l = 1, 2, \ldots, L_A \tag{6.21}$$

where the same notation is used except that the output from each layer is now a matrix $\mathbf{H}^l$. The A and S subscripts indicate that there are different weights and biases in each MLP. For example $\mathbf{W_S}^2$ indicates the weights used in the second layer of the node state update MLP. The two MLPs may also have different numbers of layers (L_S and L_A respectively).

Abstracting away from the detailed implementation of the MLPs, we can regard them as instances of a function MLP parameterised by θ_A and θ_S respectively (ie. all the weights and biases in Equations 6.20 & 6.21):

$$\mathbf{Y_S} = \text{MLP}(\mathbf{C}, \theta_S) \tag{6.22}$$

$$\mathbf{Y_A} = \text{MLP}(\mathbf{C}, \theta_A) \tag{6.23}$$

Here we call the output of the MLPs $\mathbf{Y_A}$ and $\mathbf{Y_S}$ respectively (equivalent to the final layer activations $\mathbf{H_S}^{L_S}$ and $\mathbf{H_A}^{L_A}$ in Equations 6.20 & 6.21). The dimensions of $\mathbf{Y_S}$ must be $n \times s$, and those of $\mathbf{Y_A}$ must be $n \times 15$. This is so that the they can be used in the next part of the process, explained below.

This size constraint dictates the size of the weights matrix in the final layer of the MLPs. Likewise, the dimensions of $\mathbf{C}$, the output of the neighbourhood information aggregation operation dictate the dimensions of the first layer of the MLP. For example, in the simplest case where both MLPs have zero hidden layers (so are in fact SLPs), and if $\mathbf{C}$ has dimensions $n \times 3s$ (as in Equation 6.15), then $\mathbf{W_S^1}$ must have dimensions $3s \times s$, and $\mathbf{W_A^1}$ must have dimensions $3s \times 15$. The bias vectors $\mathbf{b_S}^1$ and $\mathbf{b_A}^1$, would have sizes s and 15 respectively. In this SLP case, the total number of parameters would therefore be:

$$|\theta_s| + |\theta_a| = (3s^2 + s) + (45s + 15) \tag{6.24}$$

In a system with three possible node states ($s = 3$) this would mean a total of 180 parameters. This is a relatively small number of parameters, given that the model can be used to grow graphs of arbitrary size, with specific microstructure, as shown in Chapter 7.

6.4.5 Node State Update

Since the output of the state update MLP, $\mathbf{Y_S}$ is already the same size as $\mathbf{S}$, namely $n \times s$, it can be used directly as a replacement for the state matrix. All that needs to be done is that each row should be made one-hot. We do this by setting the maximum value of each row to one, and the other values in the row to be zero. We define the OneHot function as follows:

$$\text{OneHot}(\mathbf{X})_{ij} = \begin{cases} 1, & \text{if } \mathbf{X}_{ij} == \max_k \mathbf{X}_{ik} \\ 0, & \text{otherwise} \end{cases} \tag{6.25}$$

where $\mathbf{X}_{ij}$ indicates the entry in the i^{th} row and j^{th} column of $\mathbf{X}$, and $\max_k \mathbf{X}_{ik}$ indicates the maximum value of the i^{th} row of the matrix.

Using $\mathbf{S}'$ to indicate the updated state matrix, we can write:

$$\mathbf{S}' = \text{OneHot}(\mathbf{Y_S}) \tag{6.26}$$

Putting all the steps together we can write:

$$\mathbf{S}' = \text{OneHot}(\text{MLP}(\text{Agg}(\mathbf{A}, \mathbf{S}), \theta_S)) \tag{6.27}$$

The size of $\mathbf{S}'$ is the same as that of $\mathbf{S}$, $n \times s$. However it is possible that all the nodes in the graph will be duplicated, which will be explained below. We adopt the convention that the original nodes are assigned the updated states and any newly created nodes are assigned the original states their parents had. For this reason we create the new matrix $\mathbf{S}^+$ by vertically concatenating the new and original state matrices:

$$\mathbf{S}^+ = \begin{bmatrix} \mathbf{S}' \\ \mathbf{S} \end{bmatrix} \tag{6.28}$$

$\mathbf{S}^+$ has dimensions $2n \times s$ and represents the new state matrix *if* all the nodes choose to divide. In fact, it is likely that not all nodes do divide on an update step, so $\mathbf{S}^+$ will be reduced after the node action choices are calculated.

6.4.6 Action Choice

The first three entries in the action MLP output vectors ($\mathbf{Y_A}$ as defined in Equation 6.23) are used as an indication of the choice of "action" for each node. This can be removal from the graph, no action, or division. The choice is indicated by using the OneHot function over

the first three entries in each row of $\mathbf{Y_A}$:

$$\mathbf{E}_i = \text{OneHot}(\mathbf{Y_A}_{i,1:3}) \tag{6.29}$$

The first column of the resultant matrix $\mathbf{E}$ indicates whether a node should be removed (1) or not (0). The presence of a 1 in the second column indicates whether that node should do nothing, and the presence of a 1 in the third column indicates whether that node should divide. These columns are defined as the vectors $\mathbf{r}$, $\mathbf{o}$ and $\mathbf{d}$ respectively:

$$\mathbf{r}_i = \mathbf{E}_{i,1} \tag{6.30}$$

$$\mathbf{o}_i = \mathbf{E}_{i,2} \tag{6.31}$$

$$\mathbf{d}_i = \mathbf{E}_{i,3} \tag{6.32}$$

Note that because of the application of the OneHot function in Equation 6.29, only one of these actions may be chosen per node.

In Equation 6.28, the $\mathbf{S}^+$ matrix of node states *if* every node chose to divide was defined. This can now be reduced to retain only the rows corresponding to those original nodes which were not removed, and those new nodes which did in fact divide:

$$\mathbf{S}^- = \begin{bmatrix} \{\mathbf{S'}_{i,*} \mid \mathbf{r}_i = 0\} \\ \{\mathbf{S}_{i,*} \mid \mathbf{d}_i = 1\} \end{bmatrix} \tag{6.33}$$

6.4.7 Graph Structure Update

Restructuring the adjacency matrix is more complicated and is where most of the innovation in DGCA v2 comes into play. As mentioned, the output vectors of the action choice MLP, $\mathbf{Y_A}$ always have 15 entries. The first three entries determine the *action* that the node should take as described above. If the chosen action is removal or stasis, we do not need to look at the remaining entries. However, if the node chooses to divide, the remaining entries specify which edges the newly created node should have.

To preserve localism, a new node is not allowed to connect to remote nodes. The set of nodes that it "knows about" includes the parent node itself, and any nodes that are directly connected to the parent node (with either edge direction). However, if these were the only nodes that an child node could connect to, it would not be possible for subgraph structures to be recreated, as described in Figure 6.3. Any offspring nodes created at time t could only connect to nodes that were already in existence at time $t-1$ (their "parents' generation").

To create the potential for whole subgraphs to be duplicated, we add to the set of valid edges connections with "cousin" nodes: other offspring nodes that are created at the *same timestep*, where there is a connection between the parents (who are therefore "siblings"). This is shown in Figure 6.5. Nodes have no way of predicting whether their sibling nodes will

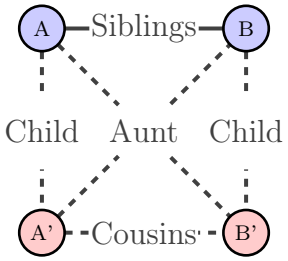


Figure 6.5: Node `A` and `B` are connected at time t (so are "siblings"). They each divide at time $t+1$, creating child nodes `A'` and `B'`. These nodes can have connections to their parents, the neighbours of their parents ("aunts") and any children of their aunt also created at time $t+1$ ("cousins").

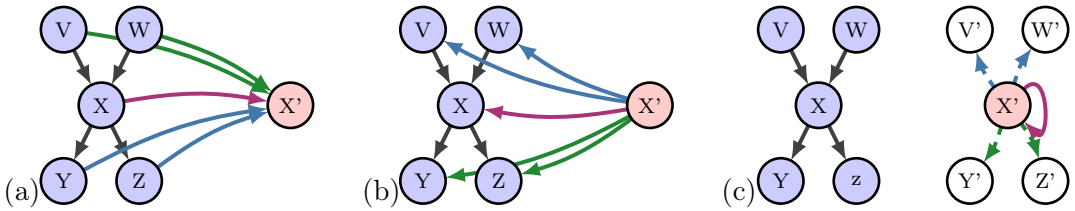


Figure 6.6: All valid edges choices for newly created node `X'`, offspring of node `X`. (a) The three sets of possible incoming edges from existing nodes. (b) The three sets of possible outgoing edges to existing nodes. (c) The three sets of possible outgoing edges to new nodes created in the same timestep (if they exist: edges to "cousin" nodes are dashed to show that they are only "potential edges"). Red edges are direct connections to/from the parent or the node itself; green edges are equivalent to what the parent node had; blue edges are the reverse of what the parent had.

divide, so when cousin edges are specified, they are only *potential* connections. For example, in Figure 6.5, offspring node `A'` can specify that it wants a connection to offspring node `B'`, but this is instantiated only *if* `B'` exists. We judge that the creation of an edge between `A'` and `B'` does *not* break the localism rule, meaning that it does not constitute a connection to a "remote" node. It is as if the two parent nodes `A` and `B` have been cloned along with their edge `A`–`B`.

Figure 6.6 shows all the valid edges for a newly created node. It can have an edge from or to its parent node (red edges in (a) and (b) respectively). It can have the same incoming or outgoing edges as its parent (green edges in (a) and (b)). It could also have the reverse of its parent's incoming and outgoing edges (blue edges in (a) and (b). Furthermore, it could have edges to other newly created nodes, as shown in (c): to itself (red), to the children of its parent's outgoing nodes (ie. cousins, shown in green), or to the children of its parent's incoming nodes (blue). Equivalent sets of edges are shown in the Equation 6.34 table, where E_g is the set of edges in the graph before the structural update step, x is the parent node, and x' is the duplicated node. The same colour coding is used to indicate different groups of edges. Allowing new nodes to have some subset of these edges preserves localism whilst allowing complex network structure (including repeated subgraphs) to emerge.

	From Existing	To Existing	To New Nodes
None	$\emptyset$	$\emptyset$	$\emptyset$
Self	$\{(x, x')\}$	$\{(x', x)\}$	$\{(x', x')\}$
Equivalent	$\{(w, x') \mid (w, x) \in E_g\}$	$\{(x', y) \mid (x, y) \in E_g\}$	$\{(x', y') \mid (x, y) \in E_g\}$
Reversed	$\{(y, x') \mid (x, y) \in E_g\}$	$\{(x', w) \mid (w, x) \in E_g\}$	$\{(x', w') \mid (w, x) \in E_g\}$

(6.34)

In order to prevent the graph becoming too dense (since biological networks tend to be very sparse), we allow each new node to have only *one* set of edges (red, green or blue) from each of the three categories: edges from existing nodes, edges to existing nodes, edges to newly created (cousin) nodes. In other words, one entry from each column of the Equation 6.34 table. Whichever set of edges is chosen within each category, the new node gets *all* of those edges. A fourth possibility is to have no edges in a category, shown in the top row of the table.

The last 12 entries in the Action SLP output vector are correspondingly divided into three groups of four. The largest value within each group indicates which set of edges from that group the new node should have, as shown in the following equations. Equation 6.35 defines $\mathbf{F}$ as the application of the OneHot function to the first group (columns 4 to 7) for each row of the matrix. Equations 6.36 & 6.37 show the same for the second and third groups. The matrices $\mathbf{F}$, $\mathbf{B}$ and $\mathbf{N}$ all have size $n \times 4$, with each row indicating whether that node should have the **Self**, **Equivalent**, **Reversed** or **None** set of edges defined in Figure 6.6 and Equation 6.34, for the edges **F**orward from the exiting nodes to the new nodes, **B**ackwards from the new to the existing, and to other **N**ew nodes.

$$\mathbf{F}_i = \text{OneHot}(\mathbf{Y^A}_{(i,4:7)}) \tag{6.35}$$

$$\mathbf{B}_i = \text{OneHot}(\mathbf{Y^A}_{(i,8:11)}) \tag{6.36}$$

$$\mathbf{N}_i = \text{OneHot}(\mathbf{Y^A}_{(i,12:15)}) \tag{6.37}$$

Looking at the columns of these indicates (with ones and zeros) which set of edges a new node should have in each category:

	From Existing	To Existing	To New Nodes
None	$\mathbf{k_{fo}} = \mathbf{F}_{(*,1)}$	$\mathbf{k_{bo}} = \mathbf{B}_{(*,1)}$	$\mathbf{k_{no}} = \mathbf{N}_{(*,1)}$
Self	$\mathbf{k_{fi}} = \mathbf{F}_{(*,2)}$	$\mathbf{k_{bi}} = \mathbf{B}_{(*,2)}$	$\mathbf{k_{ni}} = \mathbf{N}_{(*,2)}$
Equivalent	$\mathbf{k_{fa}} = \mathbf{F}_{(*,3)}$	$\mathbf{k_{ba}} = \mathbf{B}_{(*,3)}$	$\mathbf{k_{na}} = \mathbf{N}_{(*,3)}$
Reversed	$\mathbf{k_{ft}} = \mathbf{F}_{(*,4)}$	$\mathbf{k_{bt}} = \mathbf{B}_{(*,4)}$	$\mathbf{k_{nt}} = \mathbf{N}_{(*,4)}$

(6.38)

6.4.8 Implementation

DGCA v2 can be run entirely using matrix multiplication, along with Kronecker tensor products to perform the graph restructuring step. This makes the system fast to run, even for large graphs. For simplicity, it is easier to begin by imagining that every node chooses to divide, and create the matrix $\mathbf{A}^+$ with dimensions $2n \times 2n$. This can later be shrunk to $\mathbf{A}^-$ to reflect the fact that not all nodes divided, and some of the original nodes were removed. We start by defining four 2×2 matrices which are used to indicate each of the four quadrants of $\mathbf{A}^+$, with a single entry of 1.

$$\mathbf{Q_m} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \ \mathbf{Q_f} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \ \mathbf{Q_b} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \ \mathbf{Q_n} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{6.39}$$

$\mathbf{Q_m}$ indicates the upper left quadrant, which contains the original adjacency matrix; $\mathbf{Q_n}$ indicates the lower right quadrant, which contains the new nodes; $\mathbf{Q_f}$ indicates the upper right quadrant, which contains the links forward from the original nodes to the new nodes; $\mathbf{Q_b}$ indicates the lower left quadrant, which contains the links backward from the new nodes to the original nodes. We then define

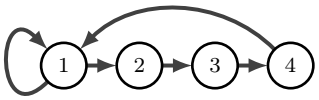
$$\begin{aligned} \mathbf{A}^+ = &\ \mathbf{Q_m} \otimes \mathbf{A} \\ &+ \mathbf{Q_f} \otimes (\text{diag}(\mathbf{k_{fi}}) + \mathbf{A} \cdot \text{diag}(\mathbf{k_{fa}}) + \mathbf{A}^\mathsf{T} \cdot \text{diag}(\mathbf{k_{ft}})) \\ &+ \mathbf{Q_b} \otimes (\text{diag}(\mathbf{k_{bi}}) + \text{diag}(\mathbf{k_{ba}}) \cdot \mathbf{A} + \text{diag}(\mathbf{k_{bt}}) \cdot \mathbf{A}^\mathsf{T}) \\ &+ \mathbf{Q_n} \otimes (\text{diag}(\mathbf{k_{ni}}) + \text{diag}(\mathbf{k_{na}}) \cdot \mathbf{A} + \text{diag}(\mathbf{k_{nt}}) \cdot \mathbf{A}^\mathsf{T}) \end{aligned} \tag{6.40}$$

where $\mathbf{Q_m} \otimes \mathbf{A}$ indicates that the original adjacency matrix, $\mathbf{A}$ (size $n \times n$) is placed into the upper left quadrant of a $2n \times 2n$ matrix, with the other three quadrants filled with zeros. The nine different $\mathbf{k}$ vectors indicate how new nodes should be connected, and are defined in Equation 6.38. For example, $\mathbf{k_{ni}}$ indicates (by 1 or 0) whether each new node should have a self-connection. These vectors are diagonalised and multiplied by the relevant matrices: the identity matrix $\mathbf{I}$ (omitted) to create connections from/to parent nodes; the original adjacency matrix $\mathbf{A}$ to create the same connections that the parent node had; the transposed adjacency matrix $\mathbf{A}^\mathsf{T}$ to create the reverse of the parent connections. This is done three times to create the three categories of edges (from existing, to existing, to new), with the results placed into the relevant quadrants via tensor products with the $\mathbf{Q}$ matrices.

Since not all the nodes may in fact have chosen to divide, and since some of the existing nodes may have chosen to be removed from the graph, the final step is to reduce $\mathbf{A}^+$ to only those nodes that should be kept. We can define the set of indices of the nodes which should be kept as:

$$\text{keep} = \{i \mid [1 - \mathbf{r}, \mathbf{d}]_i = 1\} \tag{6.41}$$

This concatenates two action indication vectors: the inverse of the removal indicator $\mathbf{r}$



$$\mathbf{A}_0 = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{6.43}$$

Figure 6.7: The starting graph (at $t = 0$) and its adjacency matrix.

(meaning the original node is kept), and the division indicator $\mathbf{d}$. It constructs the set of original node indices which survive, and new node indices where division was the chosen action. $\mathbf{A}^-$ is simply then those rows and columns of $\mathbf{A}^+$ whose indices are in the *keep* set:

$$\mathbf{A}^- = (\mathbf{A}^+{}_{ij})_{i,j \in \text{keep}} \tag{6.42}$$

A minimal Python implementation of one graph update step using this process is given in Appendix B.

6.5 Examples

This section shows a worked example to clarify the above explanation. It also shows an example of how DGCA v2 is able to produce long-distance connections.

6.5.1 Graph Restructuring

For this example we start with a graph of a unidirectional ring of four nodes, one of which has a self-loop, as shown in Figure 6.7, together with its adjacency matrix.

We use the uppercase bold letters to denote the all-zeros matrix ($\mathbf{O}$), the identity matrix ($\mathbf{I}$), the original adjacency matrix ($\mathbf{A}$) and the transpose of the original adjacency matrix ($\mathbf{A}^T$ which we abbreviate to $\mathbf{T}$). Note that all of these matrices have the same size as the original adjacency matrix $\mathbf{A}$. We denote the nth row or column of these matrices using the corresponding lowercase bold letter with the subscript $n, *$ for rows or $*, n$ for columns. For instance $\mathbf{i}_{3,*}$ means the third row of the identity matrix (ie. a vector with the third entry set to 1 and the rest to zero). These definitions are illustrated below. The colours are used for clarification of subsequent equations.

$$\mathbf{O} = \begin{array}{c} \\ \mathbf{o}_{1,*} \\ \mathbf{o}_{2,*} \\ \mathbf{o}_{3,*} \\ \mathbf{o}_{4,*} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{o}_{*,1} & \mathbf{o}_{*,2} & \mathbf{o}_{*,3} & \mathbf{o}_{*,4} \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{I} = \begin{array}{c} \\ \mathbf{i}_{1,*} \\ \mathbf{i}_{2,*} \\ \mathbf{i}_{3,*} \\ \mathbf{i}_{4,*} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{i}_{*,1} & \mathbf{i}_{*,2} & \mathbf{i}_{*,3} & \mathbf{i}_{*,4} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

$$\mathbf{A} = \begin{array}{c} \\ \mathbf{a}_{1,*} \\ \mathbf{a}_{2,*} \\ \mathbf{a}_{3,*} \\ \mathbf{a}_{4,*} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{a}_{*,1} & \mathbf{a}_{*,2} & \mathbf{a}_{*,3} & \mathbf{a}_{*,4} \end{array} \\ \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array} \quad \mathbf{T} = \mathbf{A}^\top = \begin{array}{c} \\ \mathbf{t}_{1,*} \\ \mathbf{t}_{2,*} \\ \mathbf{t}_{3,*} \\ \mathbf{t}_{4,*} \end{array} \begin{array}{c} \begin{array}{cccc} \mathbf{t}_{*,1} & \mathbf{t}_{*,2} & \mathbf{t}_{*,3} & \mathbf{t}_{*,4} \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{6.44}$$

We now consider the output of the action MLP, $\mathbf{Y_A}$. In this example, we imagine that it has values as follows:

$$\mathbf{Y_A} = \left(\begin{array}{cccc|cccc|cccc|cccc} .2 & .1 & \mathbf{.4} & & \mathbf{.9} & .6 & .3 & .7 & .6 & \mathbf{.7} & .5 & .5 & .2 & .6 & \mathbf{.7} & .5 \\ .1 & .1 & \mathbf{.2} & & .5 & .5 & .6 & \mathbf{.8} & .4 & .3 & .6 & \mathbf{.8} & .3 & \mathbf{.4} & .1 & .2 \\ \mathbf{.7} & .3 & .5 & & .2 & .2 & \mathbf{.4} & .3 & .2 & .7 & \mathbf{.9} & .3 & .4 & \mathbf{.7} & .6 & .1 \\ .4 & \mathbf{.5} & .2 & & .7 & \mathbf{.8} & .1 & .2 & \mathbf{.5} & .3 & .3 & .4 & .2 & .4 & .3 & \mathbf{.6} \end{array}\right) \tag{6.45}$$

Using Equations 6.29, 6.35, 6.36 & 6.37, which apply the OneHot function along rows in the groups of columns delimited by vertical lines above, we find:

$$\mathbf{E} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \mathbf{F} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{N} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.46}$$

This allows us to construct the nine $\mathbf{k}$ vectors in Equation 6.38, by reading down the columns of $\mathbf{F}$, $\mathbf{B}$ & $\mathbf{N}$. These vectors specify how each of the hypothetical new nodes should be connected:

	From Existing	To Existing	To New Nodes
None	$\mathbf{k_{fo}} = [1,0,0,0]$	$\mathbf{k_{bo}} = [0,0,0,1]$	$\mathbf{k_{no}} = [0,0,0,0]$
Self	$\mathbf{k_{fi}} = [0,0,0,1]$	$\mathbf{k_{bi}} = [1,0,0,0]$	$\mathbf{k_{ni}} = [0,1,1,0]$
Incoming	$\mathbf{k_{fa}} = [0,0,1,0]$	$\mathbf{k_{ba}} = [0,0,1,0]$	$\mathbf{k_{na}} = [1,0,0,0]$
Outgoing	$\mathbf{k_{ft}} = [0,1,0,0]$	$\mathbf{k_{bt}} = [0,1,0,0]$	$\mathbf{k_{nt}} = [0,0,0,1]$

(6.47)

The $\mathbf{A}^+$ matrix is calculated using Equation 6.40. The first part of this $(\mathbf{Q_m} \otimes \mathbf{A})$, places the original adjacency matrix into the upper left quadrant of the expanded matrix. The second

part constructs a submatrix which is placed into the upper right quadrant (connections **f**orward). The diagonalised $\mathbf{k_{fo}}$, $\mathbf{k_{fi}}$, $\mathbf{k_{fa}}$ & $\mathbf{k_{ft}}$ vectors are essentially used to select *columns* from the **O**, **I**, **A** & **T** matrices defined in Equation 6.44. A full expression of this would be:

$$\mathbf{O} \cdot \text{diag}(\mathbf{k_{fo}}) + \mathbf{I} \cdot \text{diag}(\mathbf{k_{fi}}) + \mathbf{A} \cdot \text{diag}(\mathbf{k_{fa}}) + \mathbf{T} \cdot \text{diag}(\mathbf{k_{ft}}) \tag{6.48}$$

Note that because of the use of the OneHot function in the construction of **F**, for each position in the four $\mathbf{k_f}$ vectors, exactly one of them will have the value 1, meaning that only one column will be selected from **O**, **I**, **A** & **T** in each position. This submatrix is placed into the upper right quadrant using $\mathbf{Q_f} \otimes \_$.

The same process is repeated for the lower left and lower right quadrants. This time, however, the $\mathbf{k_b}$ and $\mathbf{k_n}$ vectors are used to select *rows* rather than columns of **O**, **I**, **A** & **T**. This is because it was decided that each new node should have control over its incoming connections from exiting nodes (*columns* in the upper right quadrant), its outgoing connections to the existing nodes (*rows* in the lower left quadrant), and outgoing connections to other new nodes (*rows* in the lower right quadrant). Selecting rows rather than columns is achieved by reversing the terms in the matrix multiplications with the diagonalised **k** vectors. For example, the full expression for constructing the lower left quadrant is:

$$\text{diag}(\mathbf{k_{bo}}) \cdot \mathbf{O} + \text{diag}(\mathbf{k_{bi}}) \cdot \mathbf{I} + \text{diag}(\mathbf{k_{ba}}) \cdot \mathbf{A} + \text{diag}(\mathbf{k_{bt}}) \cdot \mathbf{T} \tag{6.49}$$

Once constructed, these submatrices are placed into the lower left and lower right quadrants using the tensor products $\mathbf{Q_b} \otimes \_$ and $\mathbf{Q_n} \otimes \_$ respectively.

Note that in Equation 6.40 the expressions in 6.48 & 6.49 are simplified, leaving out the **O** term (which always results in a column/row) of zeros) and skipping the matrix multiplication by the identity matrix **I** which obviously has no effect. These redundant terms are shown here in order to clarify the operation of selecting rows/columns from the **O**, **I**, **A** & **T** matrices. This operation is also indicated by the use of coloured shading in the following depiction of the fully expanded matrix $\mathbf{A}^+$:

$$\mathbf{A}^+ = \begin{array}{c} \\ \mathbf{i}_{1,*} \\ \mathbf{t}_{2,*} \\ \mathbf{a}_{3,*} \\ \mathbf{o}_{4,*} \end{array} \begin{array}{c} \begin{array}{cccc|cccc} \mathbf{o}_{*,1} & \mathbf{t}_{*,2} & \mathbf{a}_{*,3} & \mathbf{i}_{*,4} \end{array} \\ \left(\begin{array}{cccc|cccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \end{array} \begin{array}{c} \\ \mathbf{a}_{1,*} \\ \mathbf{i}_{2,*} \\ \mathbf{i}_{3,*} \\ \mathbf{t}_{4,*} \end{array} \tag{6.50}$$

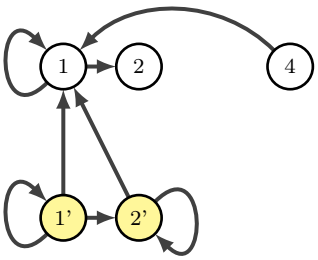


Figure 6.8: The graph from Fig. 6.7 after the restructuring update step. The adjacency matrix for this graph is shown in Eq. 6.54.

This *would* be the new adjacency matrix *if* every node divided and all of the original nodes were kept. In fact the "action" of each node is defined in matrix $\mathbf{E}$ (see Equation 6.46). Reading down the three columns of $\mathbf{E}$ gives us the three vectors $\mathbf{r}$, $\mathbf{o}$ & $\mathbf{d}$, which respectively indicate **r**emoval of the node, n**o** action, and **d**ivision of the node.

$$\mathbf{r} = [0, 0, 1, 0] \tag{6.51}$$

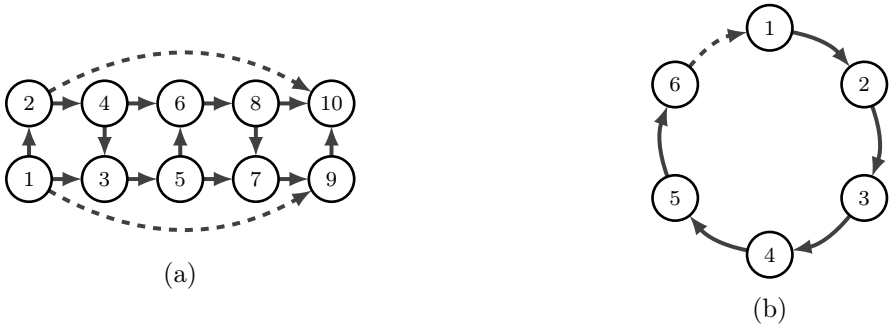
$$\mathbf{o} = [0, 0, 0, 1] \tag{6.52}$$

$$\mathbf{d} = [1, 1, 0, 0] \tag{6.53}$$

In this example, nodes 1 and 2 are to divide, node 3 is to be removed, and node 4 is to remain as it is (neither divide nor be removed). To remove node 3 we delete the third row and column of $\mathbf{A}^+$. We must also delete the seventh row and column ($3'$, which represents the potential child node of node 3). These deletions are represented using the red wavy lines in Equation 6.54. Node 4 survives but does not divide so we also remove the eighth row and column ($4'$). This deletion is represented using the blue wavy lines. Nodes 1 and 2 survive and divide, so the rows columns $1, 2, 1', 2'$ are all kept. The reduced matrix $\mathbf{A}^-$ is shown in Equation 6.54. The restructured graph represented by adjacency matrix $\mathbf{A}^-$ is shown in Figure 6.8.

$$\mathbf{A}^- = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 1' \\ 2' \\ 3' \\ 4' \end{array} \begin{array}{c} \begin{array}{cccccccc} 1 & 2 & 3 & 4 & 1' & 2' & 3' & 4' \end{array} \\ \left(\begin{array}{cccc|cccc} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \end{array} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.54}$$

This two-step process of:



(a) (b)

Figure 6.9: Two example graphs which appear difficult to create with the DGCA system owning to long-distance (ie. non-local) connections, shown as dashed lines.

1. full expansion of $\mathbf{A}$ to $\mathbf{A}^+$ using the Kronecker product formula in Equation 6.40,

2. removal of rows and columns of $\mathbf{A}^+$ to form $\mathbf{A}^-$

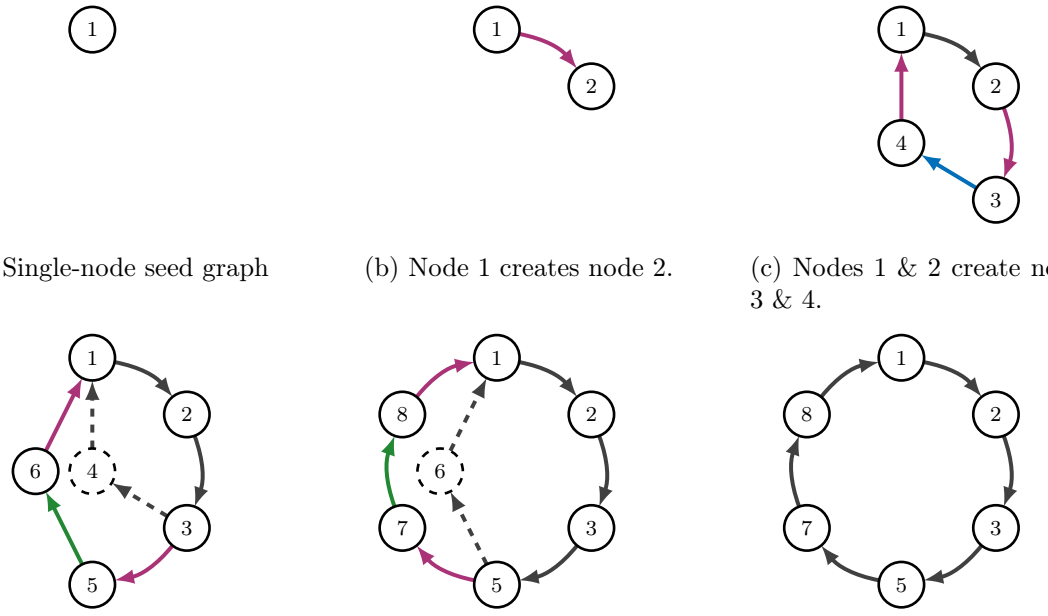
constitutes one full update step of the adjacency matrix. Time subscripts are omitted in the above equations but $\mathbf{A}^-$ can be regarded as $\mathbf{A}_{t+1}$. Note that the restructuring was entirely specified by the output of the action MLP, $\mathbf{Y_A}$ as given in Equation 6.45.

This section has shown a worked example of the graph restructuring step, illustrating how the whole operation is conducted using matrix multiplication and tensor products. The node state update has been disregarded for the purposes of this illustration. See the diagrams in Appendix B for alternative illustrations of the graph update step in shown in this example.

6.5.2 Long-distance connections

It is a feature of the system that newly added nodes may only form edges to/from their local neighbourhood nodes. At first sight, this may seem like a major constraint as it suggests it would be impossible to create graphs with long-distance connections like the one shown in Figure 6.9a. It is also difficult to see how it would be possible to create a graph with a ring topology such as the one shown in Figure 6.9b. If the nodes were created in the order of their numbers, then it would be impossible for node 6 to connect to node 1 as they would not share a local neighbourhood.

Both examples are in fact possible, if care is taken in ordering the steps of graph development. Trying to create the graph in Figure 6.9a by first growing the "ladder" structure and then adding the long distance connection (the dashed edge) would not work. Likewise, trying to create the ring in Figure 6.9b by first growing a line of nodes and then creating a long-distance connection from the last to the first, would be impossible. In both cases the desired structure must be created in miniature first and then expanded. In this way the edge which will end up appearing to be a long-distance connection can start off as a connection to a node in its local neighbourhood. The source and destination node of this edge are then be pushed apart by subsequent growth steps. This approach allows the creation of arbitrarily



(a) Single-node seed graph

(b) Node 1 creates node 2.

(c) Nodes 1 & 2 create nodes 3 & 4.

(d) Nodes 3 & 4 create nodes 5 & 6. Node 4 is removed.

(e) Nodes 5 & 6 create nodes 7 & 8. Node 6 is removed.

(f) The final graph

Figure 6.10: A possible growth process to make the ring graph shown in Figure 6.9b. Once the ring shape is established (c), each subsequent step adds two nodes and removes one. More nodes could be added in the same way to expand the ring further. Note that the colour of the edges correspond to the colours used in the explanations above.

long-distance connections in the final graph, as long as the process of their creation starts early on in the growth process. The full succession of steps to grow the graph in Figure 6.9b from a single-node seed graph is shown in Figure 6.10. A similar approach can be followed for growing the graph in Figure 6.9a, constructing the long-distance connections (dashed edges) early on in the process whilst the source and target nodes are still neighbours, and then pushing them apart by adding more nodes in between them.

This approach to generating large-scale structure by creating it in miniature first is reminiscent of biological morphogenesis: embryos often form miniature versions of structures which will later be scaled up (Gould, 2002).

Note that it would not be possible to grow either of these structures from a single node seed graph using DGCA v1 owing to the constraint that child nodes must duplicate all of their parents' edges and are not allowed edges to "cousin" nodes. For example, in Figure 6.10c when node 3 is created as the child of node 2, in DGCA v1 the edge $1 \rightarrow 3$ would be created and the edge $3 \rightarrow 4$ would not be allowed since it is to a node which did not exist at the previous timestep.

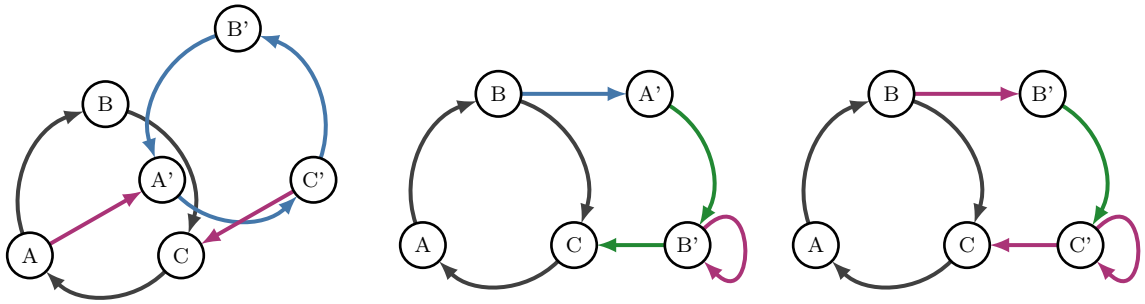


Figure 6.11: Further examples of subgraph (partial) duplication. The new edge colours follow the colours used in Fig. 6.6. In (a) all nodes (ABC) divide but the direction of edges amongst the new nodes (A'B'C') are reversed; A' gets an edge *from* its parent node A, and C' gets an edge *to* its parent node C. In (b) only nodes A and B divide; A' decides to keep an edge to B' whilst B' creates a self-loop; B' has an edge to its parent's outgoing neighbour, A' has an edge from its parents outgoing neighbour. Note that the same structure can often be created in multiple ways, as shown in (c) where nodes B and C divide and have an edge from/to their parent nodes.

6.5.3 Replication of subgraphs

The fact that DGCA v2 allows the creation of "cousin edges" means that subgraphs can be duplicated in their entirety. Importantly, the "decisions" are still taken at the level of individual nodes, so for a whole subgraph to be replicated, all of its constituent nodes must decide to divide and replicate their mutual edges. This is what is shown in Figure 6.3c. Of course, not all of the nodes may choose to divide, and different edges may be chosen from the possibilities shown in Figure 6.6 and Equation 6.34. This can lead to partial duplication of subgraphs, or duplication with structural modification. Over repeated iterations, this can lead to the creation of complex new structure. Some alternative possibilities for duplication of the simple three-node ring in Figure 6.3a are shown in Figure 6.11. This figure also highlights how there can be multiple routes to the same structure. This redundancy in the specification of the model potentially increases evolvability as there may be multiple ways of growing the same complex structure, any of which can be found by evolution.

6.6 Proof of Universality

It is desirable for the DGCA v2 model to be able to produce *any* graph, given the right parametrisation of the update function. However, as discussed in Section 6.1, the final graph depends not only on the update function but also on the starting graph. Producing an arbitrary graph given the choice of any starting graph is therefore trivially easy and could be achieved by using the target graph as the seed graph and applying zero update

steps. To claim in any meaningful sense that the model is producing any graph structure, it is therefore necessary to limit the set of allowable seed graphs to very simple ones. In the following discussion we only allow the simplest seed graph of all, namely a single node with zero edges. A proof is presented that this seed graph can be transformed into any graph, given appropriate parametrisation of the model. Furthermore, since any seed graph can be transformed into a single node by the removal of all but one of the nodes, this proof also means that the system can transform *any* graph into any other graph (albeit using an indirect route via the single node graph).

Of course, the actual development process produced by an instance of the model depends on the model parameters (ie. the MLP weights). Each set of model parameters will lead to one trajectory from the seed graph through the space of possible graphs. As explored in Chapters 4 & 5, this trajectory may enter an attractor or may carry on developing forever, since the state space of all possible graphs of any size is unbounded. This proof is therefore not concerned with showing which graphs are produced by any particular parametrisation of the model, but rather that the graph transformation process used is expressive enough to generate any finite graph via a finite number of development steps from the single-node seed graph. In other words, for the purposes of this proof we ignore the MLP part of the system, and focus on the expressive power of the graph transformation operation which is controlled by the output of the MLP. We look at the part of the system concerned with node addition (Eq. 6.40) and removal (Eq. 6.42).

It is safe to ignore the MLP part of the system in this proof if we assume that it is *possible* to find parameters (and hyperparameters such as number of layers) of the MLPs which are capable of outputting any graph transformation that is expressible in this system. This is a valid assumption, if we allow the number of node states in the system to be large. In the extreme, if every node at every timestep has a different state, then the MLPs could be parametrised to produce the required transformation at each node, even without taking into account the node's neighbourhood. For most development paths, it should be possible to find a more efficient parametrisation than this.

We now use proof by induction that the system can reach *any* finite graph from a single-node seed graph in a finite number of steps. We can enumerate all graphs of a given size using the lexicographic ordering of their adjacency matrices. The lexicographically ordered

adjacency matrices of all two-node directed graphs are can be enumerated as:

$$\begin{aligned}
A_0^2 &= \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \quad A_1^2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad A_2^2 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \quad A_3^2 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \\
A_4^2 &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad A_5^2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \quad A_6^2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad A_7^2 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \\
A_8^2 &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \quad A_9^2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad A_{10}^2 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \quad A_{11}^2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \\
A_{12}^2 &= \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \quad A_{13}^2 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \quad A_{14}^2 = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \quad A_{15}^2 = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}
\end{aligned} \tag{6.55}$$

Where the superscript 2 indicates that these are adjacency matrices for two node graphs, and the subscript numeral indicates the decimal value of the binary number produced by concatenating the rows of that adjacency matrix.

Enumerating adjacency matrices in this way will include many isomorphic graphs. For example in Equation 6.55 the graph of the A_7^2 adjacency matrix is isomorphic to the A_{14}^2 one. However, it is not necessary to eliminate these duplicates: if we can show that it is possible to create all of the lexicographically enumerated adjacency matrices of a particular size, then it must *a fortiori* be possible to create all of the possible directed graphs of that size.[2]

The proof by induction proceeds as follows:

1. Show that we can add a row and column containing any ordering of 1s and 0s to an existing adjacency matrix.

2. Starting with the two possible single-node graph adjacency matrices shown in Equation 6.56 we can add an arbitrary row and column (by step 1).

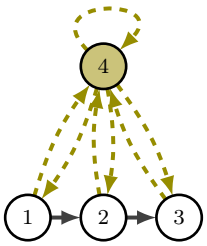
3. By doing this repeatedly we can build up any of the lexicographically enumerated adjacency matrices of a given size (ie. any ordering of 1 and 0 entries).

4. If we can create any adjacency matrix of a given size then *a fortiori* we can create at an adjacency matrix representing any isomorphism class of that size.

$$\begin{aligned}
A_0^1 &= \begin{pmatrix} 0 \end{pmatrix} \\
A_1^1 &= \begin{pmatrix} 1 \end{pmatrix}
\end{aligned} \tag{6.56}$$

Steps 2, 3, and 4 are self-evident and it is only step 1 which requires proof. The process of adding a single row and column to the the adjacency matrix at each timestep of development

[2] The first of the lexicographically sorted adjacency matrices is often used as the canonical form of a graph isomorphism class (Babai & Kucera, 1979).



$$\mathbf{A}_t = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{6.58}$$

Figure 6.12: An example graph with a "master node" which has bidirectional edges with all other nodes in the graph. This node is used in the proof that the system can construct any graph. It may be removed in the final development step. The adjacency matrix is show on the right with corresponding shading of the edges of the master node.

is illustrated in Equation 6.57. This means adding a single node at each timestep. Allowing this new row and column to contain an arbitrary ordering of 1 and 0 entries (as shown by the "?"s in Equation 6.57) means allowing the new node to have any edge to/from *any* existing node.

$$\mathbf{A}_{t+1} = \left(\begin{array}{c|c} \mathbf{A}_t & \begin{matrix} ? \\ \vdots \\ ? \end{matrix} \\ \hline ? \cdots\cdots ? & ? \end{array}\right) \tag{6.57}$$

From the description of the system given in Section 6.4, the DGCA v2 system does not appear to allow this. Indeed the system is designed specifically so that only local edges can be created.

One way around this is to maintain throughout the growth process a "master node" which is connected to all other nodes, as shown in Figure 6.12. A single node may be added at each timestep which is always the child of this master node. This allows the newly created node to connect to any of the existing nodes, since they are all in its local neighbourhood. If such a universally connected node is not required in the final graph it can be removed as the final step in the developmental process. Since we are starting with a single-node seed graph, it is easy to ensure that we maintain the existence of a universally connected master node throughout the growth process.

It would not necessarily be possible to create the universally connected master node if we were starting from a multi-node seed graph, in which such a universally connected node did not already exist. However, in this proof we *are* starting with a single node seed graph, and as mentioned, any existing graph can in fact be easily transformed into the single-node seed graph via the removal of all but one node.

Normally, the addition of a single node in DGCA v2 involves the addition of a row and column to the bottom and right of the adjacency matrix, as implied in Equation 6.57. (In our implementation, this involves the duplication of all nodes to get $\mathbf{A}^+$ followed by the

removal of all but one to get $\mathbf{A}^-$). However, this approach does not allow the added row and column to have arbitrary 1/0 entries. As illustrated below, if the master node divides, then the new row and column added to the matrix must each be chosen from the corresponding vectors **o** (all zeros), **i** (which will have 1 in the last position), **a** (all ones) or **t** (all ones).

$$\mathbf{A}_{t+1} = \begin{pmatrix} 0 & 1 & 0 & 1 & \mathbf{o/i/a/t}_{*,4} \\ 0 & 0 & 1 & 1 & \\ 0 & 0 & 0 & 1 & \\ 1 & 1 & 1 & 1 & \\ \mathbf{o/i/a/t}_{4,*} & & & & ? \end{pmatrix} \tag{6.59}$$

However, if instead all nodes are duplicated and all but one of the originals are removed, we can achieve the same effect as adding a single node, as shown in the Equations 6.60 & 6.61 below. The columns of the upper-right quadrant in $\mathbf{A}^+$ are again selected from the corresponding **o**, **i**, **a** or **t** vectors, but since only the bottom entry will be kept we can just choose from **o** (all zeros) and **a** (the final entry of which will be 1). This allows arbitrary selection of ones and zeros in what will become the new first row of $\mathbf{A}^-$. Similarly for the lower-left quadrant the rows can be selected from the corresponding **o** or **a** vectors - since only the final entry will be kept, this allows arbitrary selection of ones or zeros in what will become the new first column of $\mathbf{A}^-$:

$$\mathbf{A}^+ = \left(\begin{array}{cccc|cccc} 0 & 1 & 0 & 1 & ? & ? & ? & ? \\ 0 & 0 & 1 & 1 & ? & ? & ? & ? \\ 0 & 0 & 0 & 1 & ? & ? & ? & ? \\ 1 & 1 & 1 & 1 & ? & ? & ? & \mathbf{1} \\ \hline ? & ? & ? & ? & 0 & 1 & 0 & 1 \\ ? & ? & ? & ? & 0 & 0 & 1 & 1 \\ ? & ? & ? & ? & 0 & 0 & 0 & 1 \\ ? & ? & ? & \mathbf{1} & 1 & 1 & 1 & 1 \end{array}\right) \tag{6.60}$$

$$\mathbf{A}^- = \left(\begin{array}{c|cccc} 1 & ? & ? & ? & \mathbf{1} \\ \hline ? & 0 & 1 & 0 & 1 \\ ? & 0 & 0 & 1 & 1 \\ ? & 0 & 0 & 0 & 1 \\ \mathbf{1} & 1 & 1 & 1 & 1 \end{array}\right) \tag{6.61}$$

Note that 1 should always be chosen for the final column in the new first row, and for the final row in the new first column (shown as a coloured **1** in the matrices above). This is to give the master node bidirectional connections to the newly created node, so that the same process may be repeated at future timesteps.

This shows how a row an column filled with any ordering of 1s and 0s may be prepended

to the adjacency matrix, which is Step 1 in the proof by induction set out above. The other steps are self evident. By using a slightly contrived approach of retaining a master node throughout the growth process, each development step may add a single node with connections to any of the other nodes. The master node may be removed in the final development step if it is not required in the final graph. Whilst in most cases, there will be faster and more efficient ways to develop to the desired graph structure, the above method represents a "worst case scenario" showing that it is always possible to get to the desired structure in some way. As explained, this relies on the proper parametrisation of the MLP weights.

6.7 Conclusion

This chapter introduces DGCA v2, which solves some of the shortcomings ofr DGCA v1. In particular v2 allows a wider range of options for how new nodes can connect to their local neighbourhood. This can include connections to other new ("cousin") nodes, making it possible for subgraphs to be duplicated as a whole, with their internal connections intact. Nevertheless, the "decisions" about whether to divide (or be removed) and the edges selected are made at the level of the individual nodes. Therefore subgraphs may only be replicated if all nodes in the subgraph decide to divide and keep their respective edges. Incomplete duplication of subgraphs (or duplication with reversed or missing edges) may take place. This may allow the emergence of naturalistic network structure: biological networks frequently exhibit modular repetition but with variation (see eg. Ravasz et al. (2002)).

The next chapter shows some experiments using DGCA v2 to create networks with particular biologically inspired microstructure.

Chapter 7

Network Motifs

7.1 Introduction

Networks are found in living organisms in many forms: neural networks, metabolic networks, gene regulatory networks (GRNs), signal transduction networks, and more. Whilst Artificial Neural Networks (ANNs) are inspired by the first of these, studying the "biochemical connectionism" of the others may provide additional insight into how network structure can affect function (Lones et al., 2013).

Traditional network science metrics such as degree distribution, average path length, clustering coefficient and so on can be a useful starting point for describing these networks, but can get us only so far in actually understanding how they work. The field of Systems Biology has made significant progress in analysing the microstructure of biological networks and how it relates to network function using the concept of network motifs (Alon, 2007). In biological neural networks we also see larger-scale repeated structures, for example in cortical micro-columns, which may form functional modules (Bennett, 2020). Repeated structure, from small scale motifs to larger modules, may confer benefits on the network.

Neuroscientists such as Hiesinger (2021) have highlighted the large gap between biological neural networks and ANNs in this regard. The former are rich in structure with a developmental process "unfolding" genetic information into the final network. In conventional feedforward ANN architectures, the structure is predetermined and fixed, and the weights are trained. Training procedures such as backpropagation usually do not result in any repeating network structures. When the network structure is the result of an evolutionary algorithm, as in Topology and Weight Evolving ANNs (TWEANNs) such as NEAT (Stanley & Miikkulainen, 2002), modularity also does not typically arise. Indeed, even if the network is seeded with modular structure it normally disappears over the course of evolution, as there are likely to be mutations which increase fitness and break modularity (Kashtan & Alon, 2005).

Although breaking modularity may increase fitness for a particular task, there is ample

evidence to suggest that modular structure may confer other benefits, such as robustness, generalisation, interpretability and evolvability. For example, in the field of Reservoir Computing (RC), Wringe et al. (2023) find that Restricted ESNs, which have a degree of enforced modularity by having sub-reservoirs with sparse interconnections, can improve performance. Kashtan and Alon (2005) find that networks with modular structure are more able to evolve, as it can take very few mutations to reconfigure the interactions between modules, allowing higher-level functions to be composed out of subroutines. This can also make it easier to interpret the network structure. Recurrent Neural Networks (RNNs) and Echo State Networks (ESNs) can be viewed as dynamical sytems. Network motifs can affect these dynamics: Prill et al. (2005) find that different 3-node subgraphs (triads) confer differing degrees of stability on the network, with some encouraging static or oscillatory behaviour whilst others are more prone to chaotic behaviour. In ESNs such properties may impact higher level functionality of the network such as kernel rank, generalisation rank and memory capacity, metrics often used to characterise RC substrates (Dale et al., 2019). Alon (2007) discusses how particular biochemical network motifs can give rise to lasting or fading memory of input signals, properties which are highly relevant to ESNs, which rely on fading memory to ensure the "echo state property" (Jaeger, 2001).

This chapter uses DGCA v2 (as introduced in Chapter 6) to create networks with far-from-random motif distributions. An evolutionary search over growth rule parametrisations is used to find rules which generate particular microstructure.

This chapter is organised as follows. We review existing methods of creating networks with modular structure, describe how evolutionary and developmental processes may influence the occurrence of motifs in biological networks, and describe a commonly used method of motif detection. We then present the results of two sets of experiments that demonstrate the ability of DGCA v2 to grow networks with a range of different motif profiles: the first uses random search over growth rules; the second uses-goal directed evolution towards a particular distribution of motifs based on real biological networks.

7.2 Background

7.2.1 Generating Networks with Structure

The most basic way of creating a random network is the Erdős–Rényi model, which requires only two parameters: n, the number of nodes and p the probability of a connection between any pair of nodes (Erdős & Rényi, 1959). That model does not tend to create networks with repeated structure, and instead is used as a baseline against which motif occurrence is measured. Other graph construction algorithms such as preferential attachment (eg. the Albert and Barabási (2002) model for undirected graphs or the Price (1976) model for directed graphs), do not tend to create networks with a diverse range of microstructure: see

Figure C.3 in Appendix C for an analysis of network motifs in Price networks.

An elaboration of the Erdős–Rényi model is the Second-Order Network (SONET) model (Zhao et al., 2011). It defines four two-edge subgraphs (reciprocal, convergent, divergent and chain) whose probabilities can deviate from the independent edge probability p. For instance, whereas the independent probability of reciprocal edges between two nodes is p^2, this can be modified by the parameter α_{recip} to $p^2(1 + \alpha_{recip})$. There are four such α parameters that modify the probabilities of the four two-edge subgraphs. This approach is likely to be effective only for very small motifs, as the higher order correlations between edge probabilities quickly become very complex with larger motifs.

An alternative approach is to have a predefined library of larger motifs or modules that can be composed to produce the final network. This is the approach taken by Walter et al. (2023), who use a genetic algorithm to find effective combinations of predefined modules. This works well in the context of electronic circuits where it makes sense to have a predefined library of blocks. However, it is difficult for new motifs or modules to emerge.

One approach that encourages the spontaneous emergence of modular network structure during the evolutionary process is described by Kashtan and Alon (2005). During evolution, they regularly switch between two target functions which are easily decomposed into shared subfunctions. For example, they use the two Boolean logic functions: (X XOR Y) AND (Z XOR W) and (X XOR Y) OR (Z XOR W). By switching between these two targets, they encourage the networks to develop two submodules implementing the XOR function, which can then be combined in different ways (AND/OR). This approach relies on having target functions that can be easily decomposed, and it is not clear how it would scale to more complex tasks.

7.2.2 Evo-Devo Approach

It is hypothesised that the over-representation of certain motifs in biological networks is due to evolutionary selection: abundance or lack of particular subgraphs affects the dynamical behaviour of the network, which in turn affects the function of the network, ultimately influencing the fitness of the organism (Prill et al., 2005). Eom et al. (2006) show that the motif profiles of metabolic networks are more similar to each other in organisms from the same taxonomic group, supporting the claim that evolution is responsible for the distribution of subgraphs. Shellman et al. (2014) claim that analysis of network motifs of metabolic networks can provide information about the evolutionary origin of organelles.

An alternative hypothesis (also in Prill et al. (2005)) is that there are certain constraints on the development of the network that lead to over- or under-representation of particular motifs. Leier et al. (2007) support this view, arguing that the particular far-from-random motif distributions observed in GRNs are a consequence of the processes of network generation (gene duplication followed by functional divergence) rather than evolutionary selection.

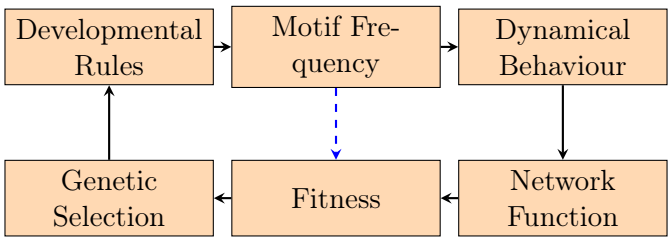


Figure 7.1: The chain of causation implied by combining the two hypotheses (motifs are evolved / due to developmental constraints) into an evo-devo process. The dashed arrow shows the simplified loop used in experiments here: we search directly for motif occurrence rather than assessing the functional properties of the network.

From an evo-devo standpoint these two views are not incompatible. It may be that the rules or constraints on network development are the direct cause of motif abundance, but that these rules are themselves genetically encoded and selected by evolution. The full chain of causation for this combined hypothesis is shown in Figure 7.1.

In this paper, we used a simplified version of this loop, basing our fitness evaluation directly on the motif distribution rather than actually assessing the behaviour of the network. The aim is to assess the ability of our system to produce networks with a wide range of different motif profiles, rather than to optimise a network for a particular task. The dashed arrow in Figure 7.1 shows this simplified loop.

DGCA naturally lends itself to this evo-devo model. The experiments conducted in this chapter evolve the developmental rules (MLP weights), and assess fitness based on the motif distribution in the graphs which develop as a result.

7.2.3 Motif Detection

A k-node subgraph is termed a "motif" if statistically over-represented in a particular network. Subgraphs that occur less than expected are "anti-motifs". Following Milo et al. (2004), Prill et al. (2005), and Shellman et al. (2014) and others, we consider three-node subgraphs. There are thirteen possible node-induced connected three-node subgraphs, or triads as shown in Figure 7.2. Note that throughout this analysis self-loops are not considered.[1]

We follow the methodology of Milo et al. (2002) to determine whether a triad is under- or over-represented in a network. Since the degree distribution of a graph might have a large impact on the occurrence of motifs, the count of the triads in the observed network is compared with the counts in an ensemble of randomised graphs with the same degree distribution. The ensemble is generated using a shuffling procedure: for each edge in the network, its source or target is swapped with the source or target of another randomly

[1]There are 86 possible 3-node motifs with self-loops, only 13 without self-loops. Excluding self-loops also follows the literature: although this is not explicitly stated in Milo et al. (2002), it is evident from the reported size of networks used.

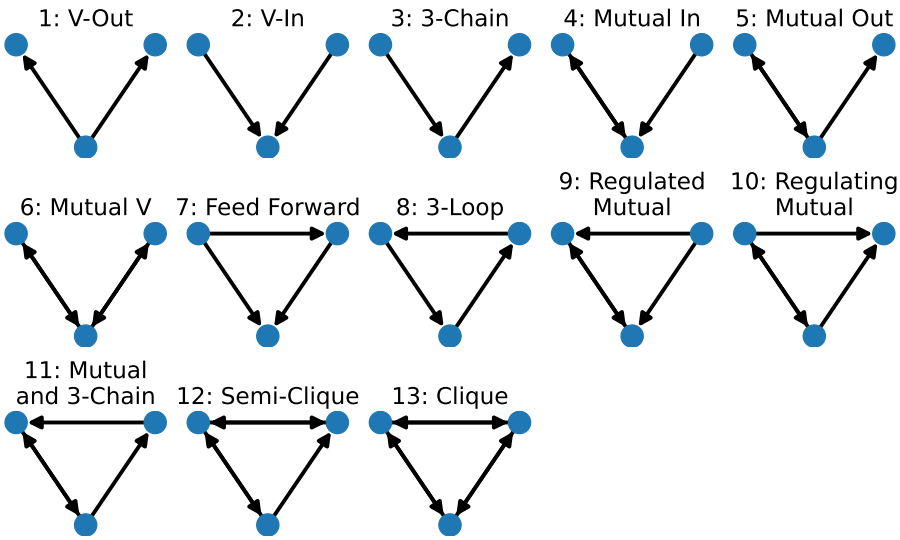


Figure 7.2: The 13 possible three node subgraphs (triads), which are used in experiments in this chapter. Triad numbering and names follow Milo et al. (2004) and Shellman et al. (2014). Subsequent figures refer to the triads by the numbers here.

selected edge.

The Z-score of triad i is calculated as:

$$z_i = \frac{c_i^{real} - \langle c_i^{rand} \rangle}{std(c_i^{rand})} \tag{7.1}$$

where c_i^{real} is the count of triad i in the real network, and $\langle c_i^{rand} \rangle$ and $std(c_i^{rand})$ are the mean and standard deviation of the counts of triad i in the ensemble.

Treating the collection of Z-scores (13 of them when $k = 3$) as a vector $\mathbf{z}$, the length (L2 norm) of this vector can summarise in a single number how atypical the motif count distribution of the observed network is compared to the randomised networks.

$$\|\mathbf{z}\|_2 = \sqrt{\sum_j z_j^2} \tag{7.2}$$

A larger value of $\|\mathbf{z}\|_2$ indicates that the graph has a statistically improbably distribution of motif counts. In a biological context, this may be the result of evolutionary selection.

Following Milo et al. (2004) we normalise the vector of Z-scores to get the "triad significance profile" (TSP) vector[2]:

$$\text{TSP} = \mathbf{z}/\|\mathbf{z}\|_2 \tag{7.3}$$

These normalised Z-scores range from -1 to 1. Triads with positive normalised Z-scores (or greater than some threshold) can be considered motifs of the network, whilst those with negative scores can be considered anti-motifs. The TSP can be used as a concise

[2]The acronym TSP is ues frequently in the remainder of this thesis - it should not be confused with the more common meaning in computer science of the Travelling Salesman Problem!

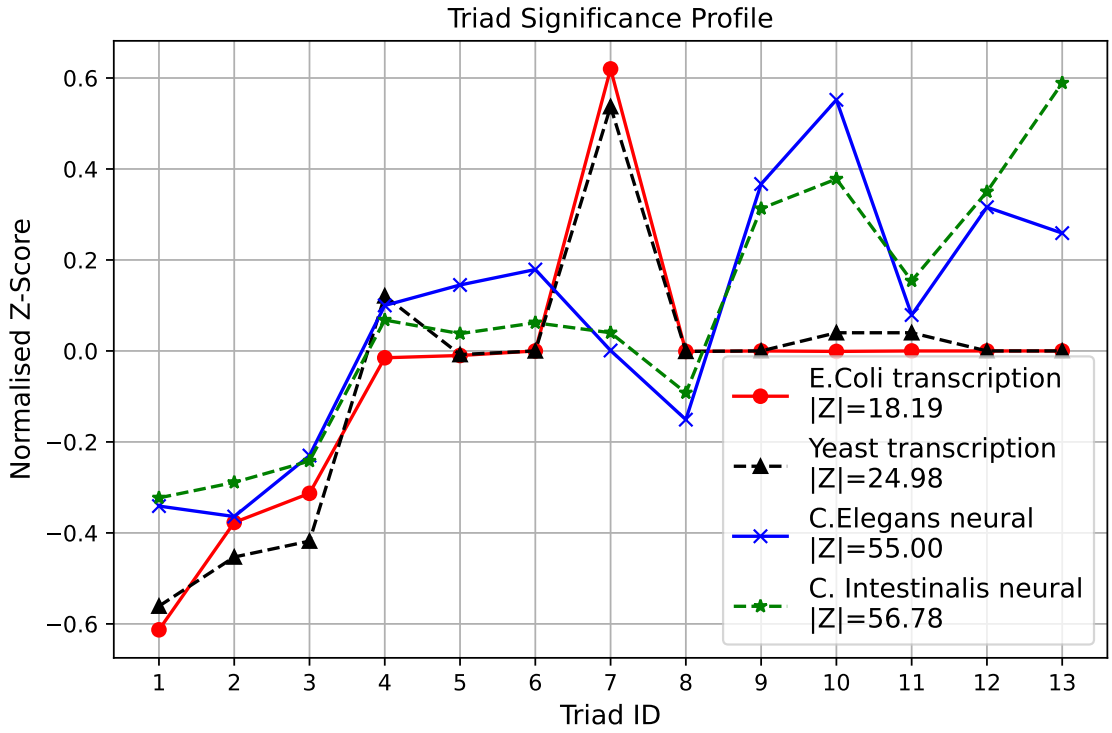


Figure 7.3: Triad significance profiles of some biological networks. The triad IDs on the x-axis refer to Figure 7.2. Although the x-axis data is categorical we link the points with a line for visual clarity, in effect treating each TSP as a vector in 13-dimensional space and using the parallel coordinate system of Inselberg (1985). As noted in Milo et al. (2004), TSPs can be used to group networks into families: the two transcription networks have similar TSPs to each other, as do the two neural networks. Note that $\|\mathbf{z}\|_2$ is higher for the neural networks, indicating that their triad distribution is more atypical (compared to random networks). The data for the *E.Coli* and Yeast transcription networks were compiled by Milo et al. (2002) The data for the *C.Elegans* and *C.Intestinalis* neural networks were compiled by White et al. (1997) and Ryan et al. (2016) respectively. In all cases, data were downloaded from https://networks.skewed.de/.

characterisation of a network's microstructure. Figure 7.3 shows the TSPs of four biological networks.

A key step in the procedure described above is counting the number of k-node subgraphs (in the observed network and in the ensemble of randomised ones). This has received considerable attention in the literature, as it is computationally intensive. We use the RAND-ESU algorithm (Wernicke, 2006), as implemented in the graph-tool Python library (Peixoto, 2014a). This uses an unbiased random sample of k-node subgraphs from the input graph. We sample 50% of the 3-node subgraphs, with sampling rates of $[1, 1, 0.5]$ at each level of the ESU tree. These values were chosen after a parameter sweep to determine a good trade-off between significance profile stability and speed. See Appendix C for the results of this parameter sweep and an explanation of the RAND-ESU algorithm.

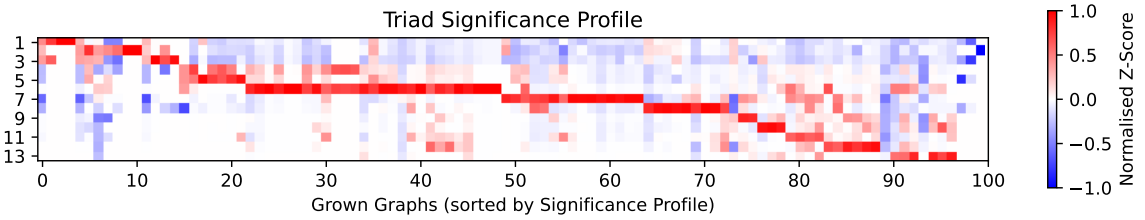


Figure 7.4: The triad significance profiles of 100 graphs grown with random growth rules (SLP weights). Each column shows the TSP of one graph using colours ranging from blue for anti-motifs to red for motifs. Columns are sorted so that the graphs with higher normalised Z-scores for triads with lower ID numbers come first. This highlights that the random search over growth rules has created graphs with a wide range of different significance profiles. Refer to Figure 7.2 for the key to the 13 triad numbers shown on the y-axis.

7.3 Experiments

7.3.1 Random Search

This experiment aims to establish whether DGCA v2 is able to grow graphs with a range of different motif profiles. We randomly initialise the weights of the two SLPs that control the growth process. We use a 3-state system: each node can be in one of three states (ie. $|\Sigma| = 3$). Systems with more possible node states possibly allow increased expressivity in that the growth rule may be able to make finer distinctions between neighbourhood configurations, but this comes at the cost of having more weights in the growth rule SLPs (see Section 6.4.4). Having more parameters may hamper evolvability, or require longer evolutionary runs to optimise. A 3-state system was felt to give a good trade off between these two considerations, and also seems appropriate given that we are considering three-node subgraphs in the motif analysis.

Starting with a "seed graph" of a single node, we run the system for up to 256 development steps or until the graph has reached 300 nodes. We discard any graph with < 100 nodes. We also only look at graphs with relatively sparse connectivity, between 0.003 and 0.03, roughly spanning the orders of magnitude of connectivity of the biological networks considered here. This is because the motif analysis used here works best on sparse but not overly fragmented networks. We run this procedure until we have 100 valid graphs, and we then calculate the TSP for each. To get 100 graphs which meet these conditions 1632 had to be discarded: 1321 because they were too small (including those where all nodes had been removed), 240 because they were too sparse, and 71 because they were too dense. The results are shown in Figure 7.4. Each column shows the TSP of one graph over the 13 triads. The columns are sorted to highlight the wide range of different significance profiles found.

This demonstrates that DGCA v2 is capable of growing graphs with a range of different motifs and anti-motifs. It appears that, when using random growth rules, it is "easier" for the system to produce graphs with certain triads as motif. For example, in Figure 7.4 we see

25 of the 100 graphs have the "Mutual-V" triad (number 6) as their most significant motif, whereas only 3 have the "Regulated Mutual" triad (number 9) as their most significant motif.

7.3.2 Goal-directed Evolution

This experiment aims to evolve developmental rules (in the form of SLP weights) that produce graphs with TSPs matching those of various biological networks. This follows the evo-devo loop shown in Figure 7.1 with the "short circuit" route of judging fitness directly on motif profile rather than on the functional behaviour of the network.

We use the TSPs of two biological networks as targets for the evolutionary search: the *E. Coli* transcriptome and the *C. Elegans* neural network. These are shown in Figure 7.3. We use one from each "family" of networks. Since $\|\mathbf{z}\|_2$ is higher for the *C. Elegans* neural network, indicating that its distribution of triads is more non-random, we might expect that it would be more difficult for evolution to generate a network with this TSP.

We use a modified version of the Microbial Genetic Algorithm (MGA) (Harvey, 2011), to evolve the SLP weights. This is a steady-state evolutionary algorithm in which pairs of individuals are chosen at random to be evaluated. After each contest the losing individual receives some of the genetic material of the winner, inspired by horizontal gene transfer in bacteria. In our case this crossover operation is effected by replacing half of the columns of the losing SLP weights matrix, with the equivalent rows from the winner. The losing individual is also mutated: we randomly change 2% of the SLP weights.

Since we have two separate SLPs (one for action choice and one for state update), we treat the two sets of weights as separate *chromosomes*. We use a population of 6 of each chromosome, combining them to give 36 (6^2) "genetically complete" individuals. When two individuals are evaluated, the weaker individual receives genetic material from the stronger in both chromosomes *unless* one of these has previously been part of an individual fitter than the winner. In this case it is not changed since we wish to avoid changing the weights of an SLP that has been effective in combination with a different partner, even if it does not perform well in its current combination.

Since we are running an evo-devo experiment, the fitness evaluation step involves growing the graph. We use the same procedure as described in the random search experiment above. We also use the same criteria for graph size and range of connectivity. If the grown graph does not meet these conditions, we assign it a fitness of 0 (or an error of ∞), making it automatically the loser of the contest. If it does meet the conditions, we calculate its TSP, as described in the Section 7.2.3.

We then calculate the Mean Absolute Error (MAE) of this significance profile compared to the target:

$$\text{MAE}(\mathbf{t}, \hat{\mathbf{t}}) = \frac{\sum_{i=1}^{13} \mid \mathbf{t}_i - \hat{\mathbf{t}}_i \mid}{13} \tag{7.4}$$

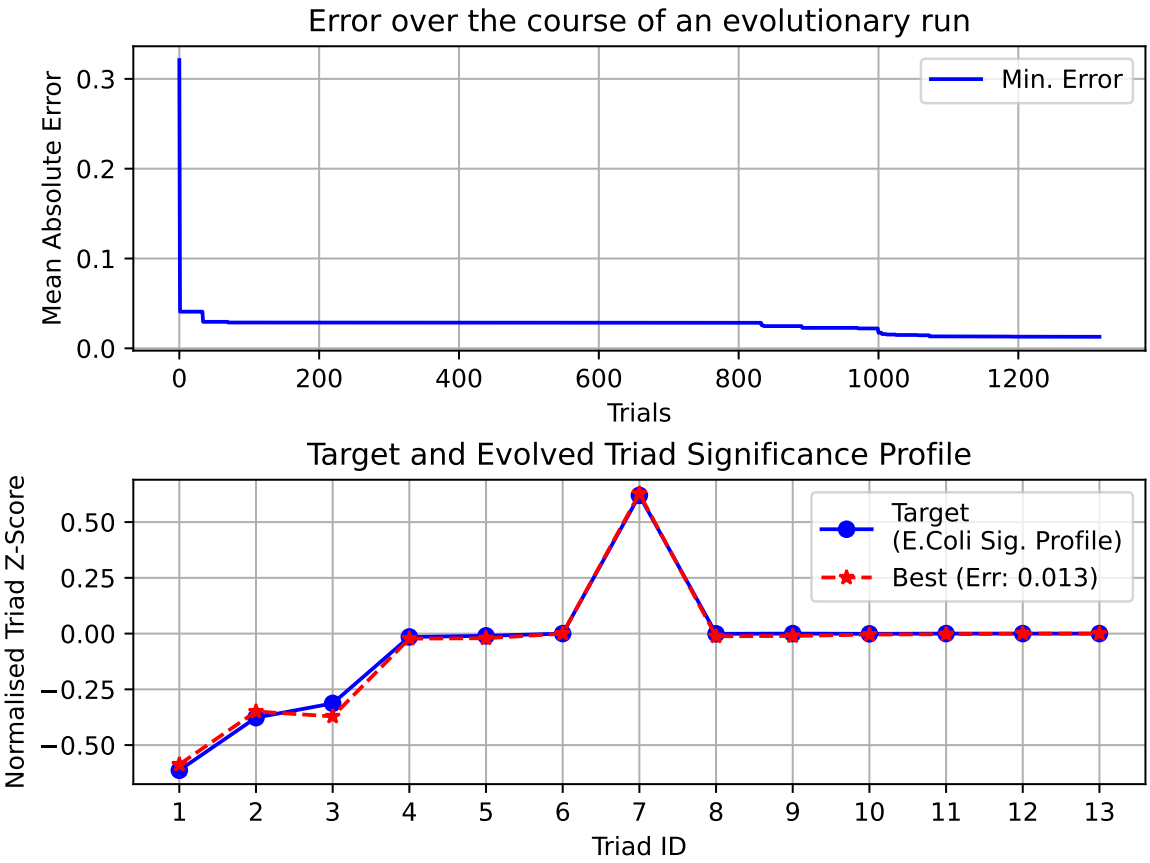


Figure 7.5: The evolutionary run with the *E.Coli* transcription network TSP as the target. The upper chart shows the minimum MAE over the course of the evolutionary run. The best result was found after 1253 trials, with a MAE of 0.013. The lower chart plots the TSP of the best evolved network against the biological network.

where $\mathbf{t}$ is the target TSP (ie. the normalised vector of motif Z-scores $\mathbf{z}/\|\mathbf{z}\|_2$) and $\hat{\mathbf{t}}$ is the found TSP. The inverse of the MAE is used as the fitness in the evolutionary process.

The results of running this evolutionary process are shown in Figures 7.5 and 7.6. The best evolved growth rules are able to create graphs with MAEs of 0.013 for the *E. Coli* transcription network, and 0.063 for the *C. Elegans* neural network. In the first case the error became low almost immediately: this is probably because it is relatively easy for the system to grow networks with an abundance of triad number 7, as discovered in the random search experiment. It took the evolutionary process longer to find developmental rules to grow a network with a TSP like that of the *C. Elegans* neural network. The random search experiment shows that random rules produce few networks with triad number 10 as their most significant.

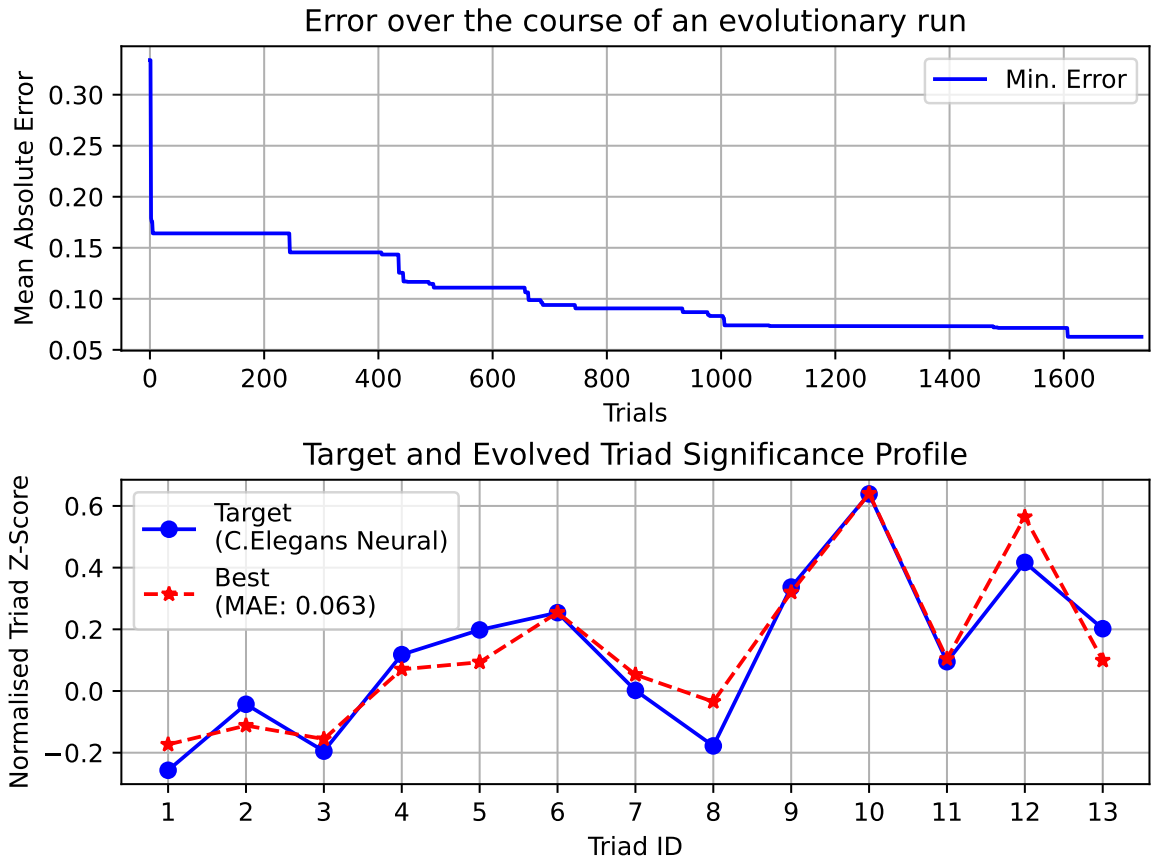


Figure 7.6: The evolutionary run with the *C.Elegans* neural network TSP as the target. The best result was found after 1607 trials, with a MAE of 0.063.

7.4 Conclusion

The experiments shown in this chapter indicate that DGCA v2 is able to grow graphs with a wide variety of microstructure, as assessed by triad significance profiles, a metric commonly used in the systems biology literature. Used in an evo-devo setup, it can create networks with TSPs similar to those of certain biological networks. These could be useful as ANNs or in an RC context, since it is hypothesised that motifs affect the network's dynamics and function. They could also be useful in other ALife contexts where modular structure is required. Milo et al. (2004) have observed that biological and designed networks can be classified into a few "superfamilies" based on network motifs. The approach introduced here could be used to generate families of networks with similar TSPs to observed biological networks in order to study how much of their behaviour is generically due to the TSP rather than the exact wiring. This highlights the value of DGCA as a graph construction alogrithm. Standard algorithms such as Erdős–Rényi, Barabási–Albert, Watts–Strogatz can be used to generate graphs with the desired high-level statistics such as degree distribution and clustering coefficient, but cannot be used to implement specific microstructure as measured by motif analysis.

Chapter 8

Transition Graphs and Linked Cycles

8.1 Introduction

In Chapter 5 the dynamical behaviour of DGCAs were explored. A random search and a novelty search were conducted over the space of transient length and attractor cycle length. The novelty search was able to find growth rules which resulted in very long attractor cycles. However, on examination of these it was found that in some cases the graph had actually split into several components, each of which was in a shorter attractor cycle (see Figure 5.3 for example). The length of the overall (system-level) attractor cycle was actually the lowest common multiple of the lengths of the component attractor cycles. Furthermore, in some cases where the system did not reach an attractor at all, it was found that all of the individual components *were* reaching attractors, but that at some point on their trajectory they were splitting off smaller components (which in turn underwent the same process). In some cases, this resulted in a form of self-reproduction: the smaller components that are split off are like "seeds" which in turn grow into the larger graphs and split off the next generation of "seeds". In fact there are several different biological metaphors which can be applied to the observed behaviour, which will be explored below.

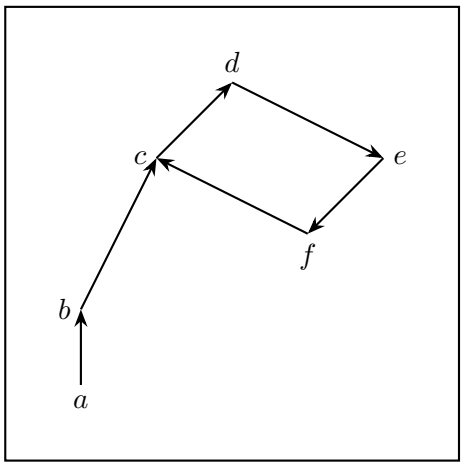
In order to analyse this complex behaviour we must examine the dynamics of the system at the level of individual graph components rather than at the at the overall system level. This leads to an unusual type of dynamical system, in which there are multiple objects moving through the state space and moreover those objects can be created or destroyed *as a result of the same dynamics*. Generally when studying dynamical systems, we focus on the orbit of a single entity through the state space. If the entity is complex, the state space may be high dimensional, but the entity is represented as a single point moving through this state space. This kind of representation was used in Chapter 5 when all nodes were treated as being part of a single graph, whether or not they were part of a single connected graph component. The state space was the space of all possible vertex-labelled directed graphs.

In this chapter, we consider each connected graph component to be a separate entity. The

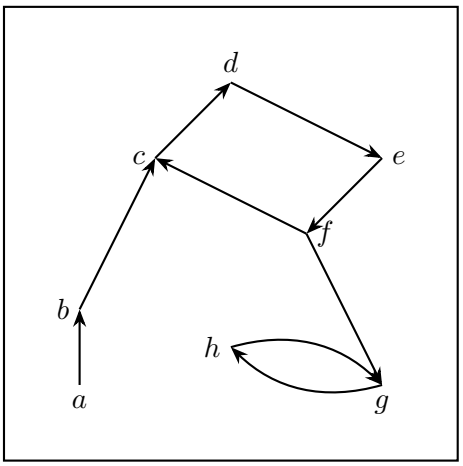
state space is therefore the space of all possible vertex-labelled *connected* directed graphs. When a graph splits into two or more disconnected components, this effectively means multiple points in the state space are instantiated. The graph can split as a result of the same developmental process which causes it to grow, shrink and restructure. Entities are therefore be created not by some exogenous process, but by the same underlying dynamics which moves the entity through the state space. Graph components can also be removed from the system if all their nodes are removed, which also happens as a result of the same growth dynamics. This removal of a graph component could alternatively be described as the graph moving to the zero-node graph-state, which is always a point attractor since there is no way for an empty graph to add more nodes (see the discussion in Chapter 4 of the "death" behaviour). However, since we will be interested in counting the number of entities extant at any point in time, we prefer to say that empty graph components have been removed and so no longer exist, rather than saying that there are an undefined (or infinite) number of components in the zero-node graph-state. Importantly, two or more graph components *cannot* reconnect to become a single component. This is because the graph is not spatially embedded: once a graph has divided into separate components there is no way for them to encounter each other again.

The phenomenon of entities being able to divide and disappear is illustrated in a generic two-dimensional state space in Figure 8.1: Figure 8.1a shows a conventional orbit through a state space, with a transient and an attractor; Figure 8.1b shows the same thing but with the entity dividing once during the transient part of the orbit and once on the attractor cycle. It may seem paradoxical to say that an entity is in an attractor cycle and yet can divide to create more entities. At the level of the whole system this is clearly not an attractor, but at the level of individual entities we can say that the original entity is in an attractor because it is cycling between a finite number of states, even though it is creating an "offspring" entity each time it goes round the cycle. Since we are treating the offspring as an entirely separate entity, and *not* as an additional attribute of the original entity, its creation does not invalidate the claim that the original entity is in an attractor cycle. An alternative diagram which may make this clearer is shown in Figure 8.1c: here we imagine that the offspring entity jumps to a "parallel universe" - after its creation it has no bearing on the orbit of the original entity. This is indeed true in the sense that disconnected graph components can never reconnect or impact each other in any way in the DGCA system, as noted above. If it *were* possible for this to happen, then it would *not* be correct to say that the original entity was in an attractor, because at some point many timesteps in the future, one of the offspring entities might have an impact on the state of the original entity.[1] As it is, the offspring can never affect the original as if it has moved to a different "universe". It should be noted each

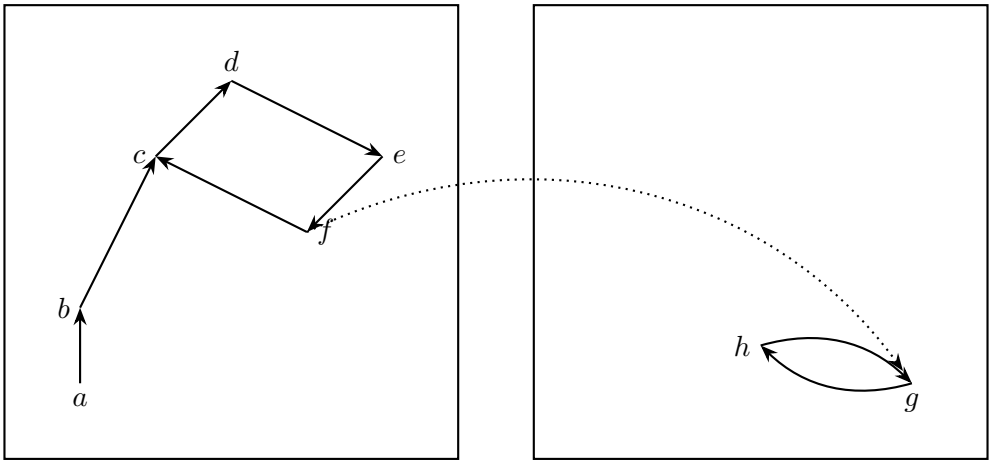
[1]This is the case in conventional grid-based CA. For example in Game of Life, a small region of the grid may appear to be in an attractor (for example a "spinner"), but it can always be disrupted in the future (for example by a moving glider), so no part of the system can truly be in an attractor unless the whole system is in one.



(a) Orbit of a single entity in state space



(b) Orbit of a dividing entity in state space



(c) Orbit of a dividing entity in state space with the "parallel universe" depiction.

Figure 8.1: A conventional depiction of a discrete dynamical system moving through state space is shown in (a): after a transient of two steps an attractor is reached ($cdef$). In (b) a similar diagram is shown for a dividing entity - after point f the object divides in two, with one element going to point c in the state space and one going to point g. The second element ends up in an attractor (gh), and we argue that the first element is also in an attractor cycle ($cdef$) despite the fact that it divides each time it goes round this cycle. To make this clearer we imagine in (c) that the second element comes into existence in a different universe: its creation has no bearing on the attractor dynamics of the original entity.

"universe" consists of an identical state space (in our case the space of graph-states).

8.2 Analogies

This section outlines a few different analogies for the graph division behaviour, showing that it could be used as a simple Artificial Chemistry model, or as a model of biological cell differentiation.

8.2.1 Artificial Chemistry analogy

If we treat each disconnected graph component as a a "molecule" then we can think of the system as an Artificial Chemistry (AChem) and perform an analysis of the concentration of each type of "molecule". Care must be taken in appyling this analogy to DGCAs. In most AChems, molecules interact with each other, reacting together to create other molecules, or acting as catalysts for other reactions. In the present system this does not happen: once graph components ("molecules") have become disconnected, they no longer interact with each other. Instead, each graph component continues on its own developmental trajectory. In other words each graph has its own (closed) dynamics, whereas in a most AChem models, the molecules themselves do not possess dynamics, but rather the dynamics comes from the interaction between molecules (Banzhaf & Yamamoto, 2015). Subsymbolic AChems such as SpikyRBN (Krastev, 2018) are an exception: each individual molecule *does* possess (open) dynamics which affects its ability to bond with other molecules. Reactions are then an emergent property of the interacting dynamics of two molecules. The same is true of automata chemistries such as Stringmol (Hickinbotham et al., 2016). The DGCA system is subsymbolic, since each molecule is formed of a graph of connected nodes and has its own internal dynamics, but it cannot strictly be considered an AChem because these molecules do not interact. Despite this, we can still interpret some of the behaviour of a DGCA system as a chemical reaction network by working at a higher level of abstraction. In a conventional chemical reaction network we might talk about the reactions:

$$\mathrm{A} + \mathrm{B} \xrightarrow{\mathrm{X}} \mathrm{Y} \tag{8.1}$$

$$\mathrm{C} + \mathrm{D} \xrightarrow{\mathrm{Y}} \mathrm{Z} \tag{8.2}$$

In which the reaction of molecules A and B is catalysed by X to produce Y. Y in turn catalyses the reaction of C and D to produce Z. In some cases we might be uninterested in the reactants A, B, C and D (which may be small molecules in plentiful supply) and more interested in the catalysts and products X, Y and Z. We therefore might represent these two reactions as a simple chain:

$$X \rightarrow Y \rightarrow Z \tag{8.3}$$

In this representation, the production of one molecule leads directly to the production of the next via catalysis, and the actual nature of the reaction and the substrate molecules used are abstracted away. It is this representation which allows us to treat the DGCA system as an AChem. At each timestep, each molecule/graph may: transform into another molecule/graph, an analogy for the kind of catalysed reaction described above; or it may remain static if it is in a point attractor, like an inert molecule; or it may split into multiple components, as in a catalysed reaction with multiple molecular products.

Even in a standard AChem (such as that used in Dittrich and Di Fenizio (2007)), it

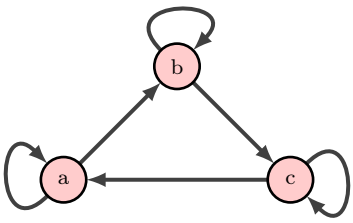


Figure 8.2: The simplest form of hypercycle. Each molecule catalyses itself and another molecule which is the next step in a larger cycle. The single molecule autocatalytic cycles and the three-step autocatalytic cycle are mutually reinforcing.

is allowable to have some reactions which only have one input, and one or more output molecules. This could be interpreted as molecular decay. On this view, we could say that the DGCA system is a constrained AChem where *only* this kind of reaction exists.

8.2.2 Hypercycles

Under this analogy, a cyclic attractor such as those considered in Chapter 5 becomes an autocatalytic cycle. However the more complex behaviours which were found are reminiscent of chemical hypercycles (Eigen & Schuster, 1977). A hypercycle is a system of mutually reinforcing autocatalytic cycles - a canonical example is shown in Figure 8.2. Each molecule catalyses itself as well as the next molecule in a larger autocatalytic cycle.

It is also possible for a molecule which is part of an autocatalytic set to catalyse the creation of a molecule which is *not* part of the set. The creation of this molecule is a kind of "side effect" of the cycle, but this phenomenon has also been described as a form of prebiotic parasitism. A molecule (or other reaction network) which is aided by the products of an autocatalytic set, but does not contribute to it can ultimately disrupt its "host" by decreasing the concentrations of molecules in the autocatalytic set below the level necessary for its survival (Eigen & Schuster, 1977). Parasitism has also been observed as a prominent feature in some AChems (Hickinbotham et al., 2021).

This aspect of competition and parasitism between the "quasi-species" of different autocatalytic sets will not be explored here. However, it provides some useful concepts for describing the relationships between different linked groups of cycles. As shown in Figure 8.3, individual cycles which are mutually reinforcing can be considered a single hypercycle. Where the reinforcement is only in one direction we consider them separate groups in a host/parasite relationship. Where there is no reinforcement in either direction, we consider them entirely separate cycle groups. This terminology provides a useful way to describe the higher-level organisation of the networks we find.

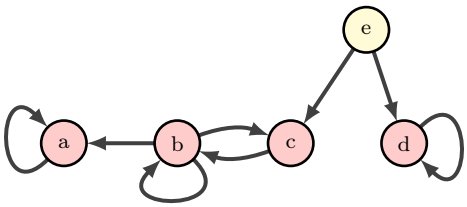


Figure 8.3: This system illustrates the different possible relationships between cycles. There are four cycles shown (a, b, bc, and dd). The b and bc are cycles are mutually reinforcing and can be considered a single hypercycle. The a cycle is not part of this hypercycle but is parasitic on it, since it benefits from the $b + bc$ hypercycle but does not give anything back to it (in AChem terms, the concentration of molecule a increases thanks to the b and bc cycles but not vice versa). The cycle d is also separate from the $b + bc$ hypercycle but is not parasitic, even though it may share a precursor step, shown here as molecule e. We will generally use the term "cycle group" rather than "hypercycle" - this allows us to say that a is in its own "cycle group" (even though it is the only member of this group and so is not technically a hypercycle). The diagram therefore shows four cycles, but three cycle groups, one of which is in a parasitic relationship.

8.2.3 Cell Differentiation Analogy

An alternative biological metaphor for the behaviour investigated in this chapter is cell differentiation. On this view we can see each component as a single cell, with the cell state being indicated by the graph state (ie. its structure and individual node states). This is similar to the analogy used by Stuart Kauffman in presenting Random Boolean Networks (RBNs) (Kauffman, 1996), in which the overall state of the RBN represents the state of a single cell. Attractor cycles in the RBN state space are analogised as cell cycles. In applying the metaphor to DGCA, we get the additional feature that cells can divide. A graph component may be in an attractor cycle (cell cycle) and at a certain point of that cycle it may split off one or more additional graph components, as if it is undergoing cell division. Each of these "daughter cells" will then of course be subject to their own dynamics and may divide further and/or enter their own attractors.

It is a pleasing feature of this analogy that cell division takes place endogenously as a consequence of the same dynamics which power the changes in cell state. This means that the speed of cell division is variable. Some cells may divide N times per attractor cycle, with both N and the length of the cycle being different for different cells. Other cells may not be on an attractor cycle at all, but nevertheless divide at certain points on their trajectory through graph state space. Furthermore, there is no explicit cell division operator, but rather graphs divide as a consequence of the normal node removal operation. Node removal results in graph splitting only if it takes place at a sparsely connected point in the graph structure. It may also be the case that splitting is a multi-step operation as nodes are successively removed. This emergence of division from the "subsymbolic" dynamics contrasts with many ALife systems such as AVIDA which use an explicit cell-division operator (Adami, 1998).

Chapter 9 explores this cell division and differentiation metaphor in more detail.

8.3 Formalism

8.3.1 Growing graphs

To recapitulate the DGCA formalism in Chapter 6, a graph in the system is represented by the tuple (repeated from Equation 6.1):

$$g = (V_g, E_g, \sigma_g) \tag{8.4}$$

representing vertices $V_g \subseteq V$, edges which are ordered pairs of vertices $E_g \subseteq (V_g \times V_g)$, and a mapping from vertices to a finite set of states $\sigma_g \subseteq \sigma$, The domain of σ_g is V_g and all the graphs produced by a particular run of the system use node states from the same set Σ, hence:

$$\sigma_g : V_g \to \Sigma \tag{8.5}$$

In this chapter all the analysis is done at the level of individual graph components, so we add the condition that each graph must be *connected*. For directed graphs both strongly connected and weakly connected components are defined. A strongly connected component is a group of vertices and edges in which every vertex is reachable from every other by following the edges in their given direction. A weakly connected component is one in which every vertex is reachable from every other, disregarding the direction of the edges. Here we use the term "connected component" exclusively in the weak sense. Since all the graphs in this chapter are connected, we use the terms "graph" and "component" interchangeably.

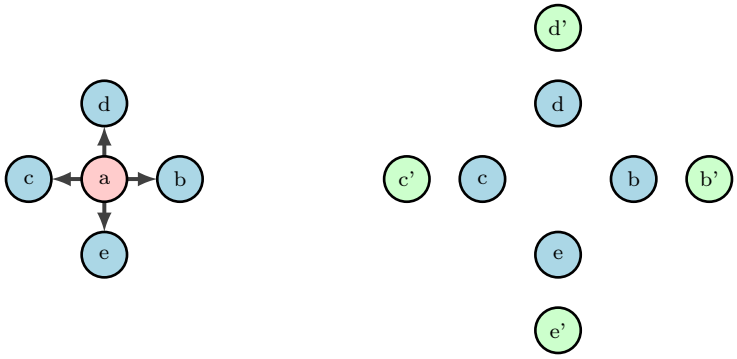
We define G as the set of all graphs as defined in Equation 8.4 that are weakly connected:

$$G = \{g \,|\, g \text{ is weakly connected}\} \tag{8.6}$$

The other element of the system is a transition function that transforms one graph into the next. In previous chapters we were working at the level of the whole system rather than individual connected components, so the transition function simply mapped from one graph (which could be disconnected) to another (which could be disconnected): see Equation 6.8. Since we are working at the level of single connected graph components in this chapter, and a growth step may result in splitting the graph, the transition function used here maps to a *set* of graphs. The size of set list can range between zero (if all nodes of the previous graph are removed) and $2(n-1)$ where n is the number of nodes in the previous graph. This value is the maximum amount of graph fragmentation possible in a single timestep, explained in Figure 8.4. The transition function f therefore maps from a graph to a set of zero or more graphs:

$$f : G \to \mathbb{P}\, G \tag{8.7}$$

The symbol $\mathbb{P}$ denotes the power set.



$t = 0,\ n_0 = 5$ $t = 1,\ n_1 = 8 = 2(n_0 - 1)$

Figure 8.4: This diagram shows the *maximum* amount of fragmentation of a graph that can take place in a single timestep. Fragmentation can occur via two methods: when a node is removed along with its incident edges (the DGCA system does not allow removal of edges alone); or when a node divides and the daughter node chooses not to have any connections with the existing graph. In the first method, at least one node must be removed to cause fragmentation. If the nodes are connected in a radial pattern, as in the graph on the left (at $t = 0$) then the removal of the central node (A) will cause all the other nodes to become detached. This will give rise to $n - 1$ disconnected components (where n is the number of nodes at the previous step) and is the maximum amount of fragmentation in a single step via the first method. If all of the remaining nodes also divide in the same step and their daughter nodes have no edges (the second method of fragmentation), then we can end up with $2(n - 1)$ graph components, as shown in the diagram on the right (at $t = 1$).

To run the system we start with a single seed graph at $t = 0$, denoted by g_0. After one update step (one application of f), we have a list of graph components:

$$f(g_0) = \{g_1, g_2, ...g_m\} \tag{8.8}$$

The size of the set of graph components produced (m) can range between 0 and $2(n - 1)$ where $n = |V_{g_0}|$ as explained above. Any of the graphs produced ($g_1...g_m$) may have the same "graph state" as any other (or as g_0), meaning that it is an isomorphic graph conditional on node states as laid out in Definition 1.

We define $G_{\neq} \subset G$ as the set of all isomorphism classes: it contains a single instance of each graph state. The subscript $\neq$ after a set indicates a set of unique isomorphism classes (notation adapted from Mirsky (1971)). We define a "de-duplicate" function, which takes a set of graphs and returns the set of unique isomorphism classes to which they belong; in other words, it removes duplicate "graph states":

$$\text{dedup} : \mathbb{P}G \to \mathbb{P}G_{\neq} \tag{8.9}$$

We define the function F as the lifted version of f, meaning it is applied to every member of a set of graphs isomorphism classes (analogous the `map` operation in many functional programming languages):

$$F : \mathbb{P}G_{\neq} \to \mathbb{P}G \tag{8.10}$$

So the composition of dedup and F, dedup $\circ F$, applies the transition function to a set of graphs and returns a set of unique isomorphism classes:

$$\text{dedup} \circ F : \mathbb{P}G_{\neq} \to \mathbb{P}G_{\neq} \tag{8.11}$$

We define a set $\mathcal{G}_t \subseteq G_{\neq}$ as the set of graph states we have in the system at a particular point in time, t. We can then say:

$$\text{dedup} \circ F(\mathcal{G}_t) = \{f(g) \,|\, g \in \mathcal{G}_t\}_{\neq} \tag{8.12}$$

The running of the system can be expressed as:

$$\mathcal{G}_0 = \{g_0\} \tag{8.13}$$

$$\mathcal{G}_{t+1} = \text{dedup} \circ F(\mathcal{G}_t) \tag{8.14}$$

The set $\mathcal{G}$ starts off containing the seed graph g_0 (Eq. 8.13); at each update step we apply the update function to all of the graphs in $\mathcal{G}$ and shrink the resulting set of graphs down to a single instance of each graph state.

8.3.2 Induced transition graph

As can be inferred from the form of Equation 8.14, this process describes a discrete dynamical system. However, unlike a standard dynamical system, it consists of many components which are each following their own trajectory through the state space, as described in Figure 8.1.[2] At any point, any of these components can divide into two or more components. The products of this division may set out on their own trajectories, or join the trajectories of components that have already been seen. This gives rise to what we could call "complex attractors", when several components have overlapping attractor cycles that repeatedly converge and diverge.

Having run the system for a fixed number of steps K or until every path has been explored (see Section 8.4) we can construct a higher-order directed graph showing how the development steps of graph components are linked. We call this the transition graph T. The vertices of T are all the unique graph states that have been seen during the run of the system, which we refer to as $\mathcal{G}_\Omega$.

$$\mathcal{G}_\Omega = \text{dedup}\left(\bigcup_{t=0}^{K} \mathcal{G}_t\right) \tag{8.15}$$

where K is either a finite number of steps for which we have run the system before an

[2] Whilst it would of course be possible to use a standard dynamical system representation by using a state space of *sets* of graph states, we prefer this representation as it allows clearer analysis of the relationship between individual graph states.

arbitrary cutoff, or in the case where we explore all development paths to completion and they all result in an attractor, $K = \infty$.

The edges of the transition graph T are ordered pairs of graph states, $\mathcal{E}$, as defined in Eq. 8.16. There is an edge from one graph state g to another h if h is one of the components produced by the application of the transition function f to g:

$$\mathcal{E} = \{(g, h) \,|\, g, h \in \mathcal{G}_\Omega, h \in f(g)\} \tag{8.16}$$

The transition graph T is a tuple of vertices $\mathcal{G}_\Omega$ and directed edges $\mathcal{E}$:

$$T = (\mathcal{G}_\Omega, \mathcal{E}) \tag{8.17}$$

Note that by construction T is a weakly connected graph: all vertices (graph states) are reachable from the seed graph state $g_0 \in \mathcal{G}_\Omega$ via the transitions $\mathcal{E}$. The set of edges $\mathcal{E}$ is really therefore the set of transitions under f from one graph state to the next. Note that self-loops are possible in T. This occurs when the application of the transition function f to a graph state results in no change, or results in several components, one or more of which is isomorphic to the original graph state.

The out-degree of a vertex in T indicates the number of unique graph states that a graph state splits into when the transition function is applied (Eq. 8.18). The in-degree indicates the number of graph states which produce *this* graph state when the transition function is applied (Eq. 8.19).

$$\deg_{out}(y) = \#(\text{dedup} \circ f(y)) \tag{8.18}$$

$$\deg_{in}(y) = \#(\{x \,|\, x \in \mathcal{G}_\Omega, y \in f(x)\}) \tag{8.19}$$

where $\#(.)$ denotes the cardinality of a set.

Note that the transition graph defined here is similar to the CA state transition graphs defined in Martin et al. (1984) and used in Wuensche et al. (1992) to map out basins of attraction for 1-dimensional CA. See Figure 8.5 for an example of such a graph. The similarity lies in the fact that vertices in this graph represent the overall state of the CA and edges represent state transitions. However, these conventional CA transition diagrams are instances of *functional graphs* which are used to represents a function from a finite set to itself ($f : S \to S$) (Harary, 1969). Each vertex represents an element of this set and each edge shows the application of the function (ie. the set of edges $E = \{(x, f(x)) \mid x \in S\}$). Since S is a finite set, repeated applications of the function will eventually lead to a cycle. Each connected component of a functional graph therefore contains exactly one cycle, possibly with trees leading into it. There may be multiple components (indicating multiple basins of attraction) in which case the graph can be described as a *pseudoforest*. Functional graphs are appropriate for illustrating the transitions in a conventional CA, where the state space

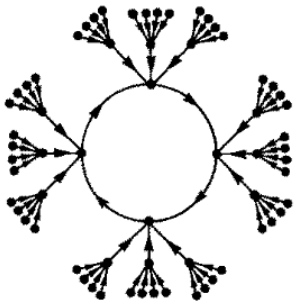


Figure 8.5: ECA state transition diagram showing the characteristic topology of a functional graph, with trees leading into a single cycle. Figure reproduced from Martin et al. (1984) Fig. 2

is finite.

There are some important differences between such functional graphs and the transition graph defined here. In a functional graph each vertex has exactly one outgoing edge, since the application of the function leads to one result, whereas here vertices may have multiple outgoing edges due to the phenomenon of graph-state division. This is indicated by the fact the transition function in Equation 8.7 has as its range the powerset of graph-states ($f : G \to \mathbb{P}G$). The form of f already precludes a functional graph since its domain and range are not the same. Furthermore the set G of graph-states is unbounded. This means that not all trajectories will necessarily result in a cycle. Thus, despite the superficial similarities to typical CA state transition graphs, the transition graphs used here do not have any of the key properties of functional graphs. We should therefore not expect them to have the characteristic topology of trees leading into a single cycle as shown in Figure 8.5. Instead they have more flexible topologies, with each vertex having unconstrained in- and out-degree, and the possibility of multiple (or zero) cycles in a single connected graph component.

Diagrams The transition graph T is illustrated in this chapter with diagrams of the type shown in Figure 8.6. These show the development steps undergone by a DGCA graph. Each large circle contains the state of the graph at one point in time. These circles are linked together by arrows which indicate one development step. A green outer circle indicates that this is the seed graph; a yellow circle denotes a step on the transient of the system; a red circle indicates that this state of the graph is part of an attractor cycle. The nodes of the graphs themselves (inside the circles) are coloured according to their states. Arrows between the large outer circles indicate development steps. For clarity we will refer to the underlying graphs as having “nodes”, “edges” and “node states”, but will refer to the higher-order nodes (which represent graph states) as “circles” and the edges between them (representing graph state transitions) as ”arrows”. Figure 8.6 shows a simple four step transient followed by a

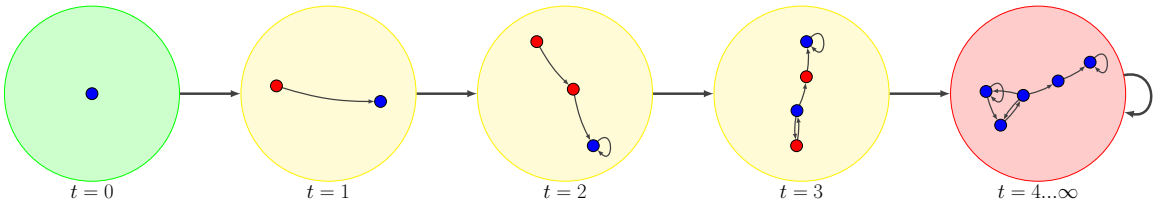


Figure 8.6: A two-state DGCA system is shown - nodes can either be blue or red. The seed graph is a single blue node (shown in the green circle to indicate it is the starting graph, at $t = 0$). This undergoes three development steps on the transient (shown in yellow circles) and after the fourth step reaches a point-attractor, indicated by a red circle. The fact that this final circle has an arrow pointing back into itself indicates that further development steps will not result in any change to the graph (ie. it is in a point attractor). Future diagrams will omit labelling the timesteps - after a cycle is reached the timesteps lose meaning. The growth rule used to generate this example was found by random search.

point attractor.[3] Figure 8.7 shows a six step transient followed by a point attractor.

8.3.3 Cycles

We refer to the set of cycles in the higher-order transition graph T as $\mathcal{C}$. Each cycle is an ordered sequence of graph states in $\mathcal{G}_\Omega$ (ie. vertices in T) with edges from one graph-state to the next (and from the last back to the first):

$$\mathcal{C} = \{[x_1, x_2, ...x_n] \,|\, x_1, ...x_n \in \mathcal{G}_\Omega,\ (x_1, x_2), (x_2, x_3), ...(x_n, x_1) \in \mathcal{E}\} \tag{8.20}$$

Note that the of graph states forming a cycle $[x_1, ...x_n]$ should be the smallest list covering all graph states in a cycle, so that it only goes around the cycle once. Cycles in the transition graph T indicate attractors in the graph-state space. However, as discussed, this does not preclude division at some point on the attractor cycle. When a graph-state on an attractor cycle divides, one of the resultant graph-states continues on the cycle; the others may embark on different trajectories. These may include trajectories which do not find an attractor but continue growing indefinitely ("runaway growth"), ones which diminish to zero nodes (a special case of a point attractor), and those which end up in non-zero point or cyclic attractors. An example of the first case is shown in Figure 8.8. The largest graph component proceeds on a runaway growth trajectory but along the way it splits off single nodes which restart the growth, creating a cycle (the graph-states on this cycle are shown in red circles). This kind of behaviour is somewhat analogous to plant or fungal growth where the main organism keeps growing but splits off small seeds/spores which start out on the same growth trajectory.

[3] It so happens that this particular transition diagram *does* have the form of a functional graph (out-degree of 1, single cycle), but that is not the case for the rest of the transition diagrams shown in this chapter.

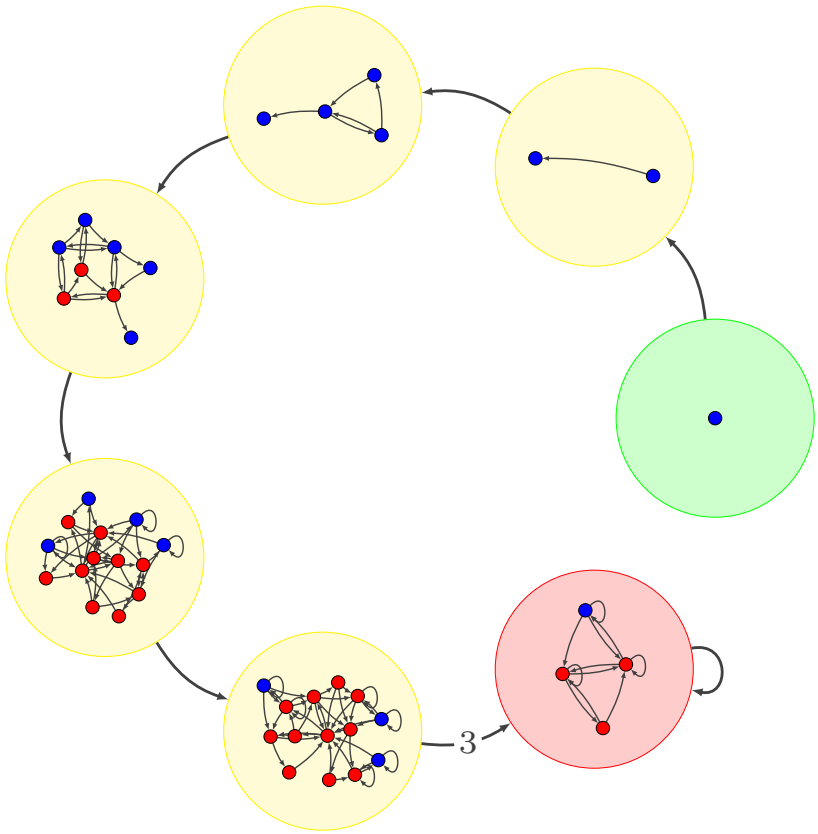


Figure 8.7: In this example, the transient is six steps. However on the final step of the transient, the graph splits into three identical components (this is indicated by the number 3 on the arrow out of the final yellow circle; the arrows without number labels indicate that a single instance of the subsequent graph is produced). These final three components are each in the same graph state, which is a point attractor.

8.3.4 Linked Cycles and Cycle Groups

We define a set of ordered pairs of linked cycles $\mathcal{L}$ where the application of the transition function to the members of the first cycle leads to one or more members of the second cycle.

$$\mathcal{L} = \{(\alpha, \beta) \,|\, \alpha, \beta \in \mathcal{C},\ F(\text{set}(\alpha)) \cap \text{set}(\beta) \neq \varnothing\} \tag{8.21}$$

We use lowercase Greek letters to refer to particular cycles, or we may refer to them directly as the ordered list of their graph states (for instance we could define $\alpha := abc$). We can now construct a directed graph of these linked-cycles, with vertices $\mathcal{C}$ and edges $\mathcal{L}$.

$$\text{LinkedCycleGraph} = (\mathcal{C}, \mathcal{L}) \tag{8.22}$$

This is another level of abstraction above the transition graph T. For example, if we take the graph in Figure 8.2 as a transition graph, we can see that it has four cycles: a (self-loop); b (self-loop); c (self-loop); and abc. These form the nodes of the linked cycle graph shown

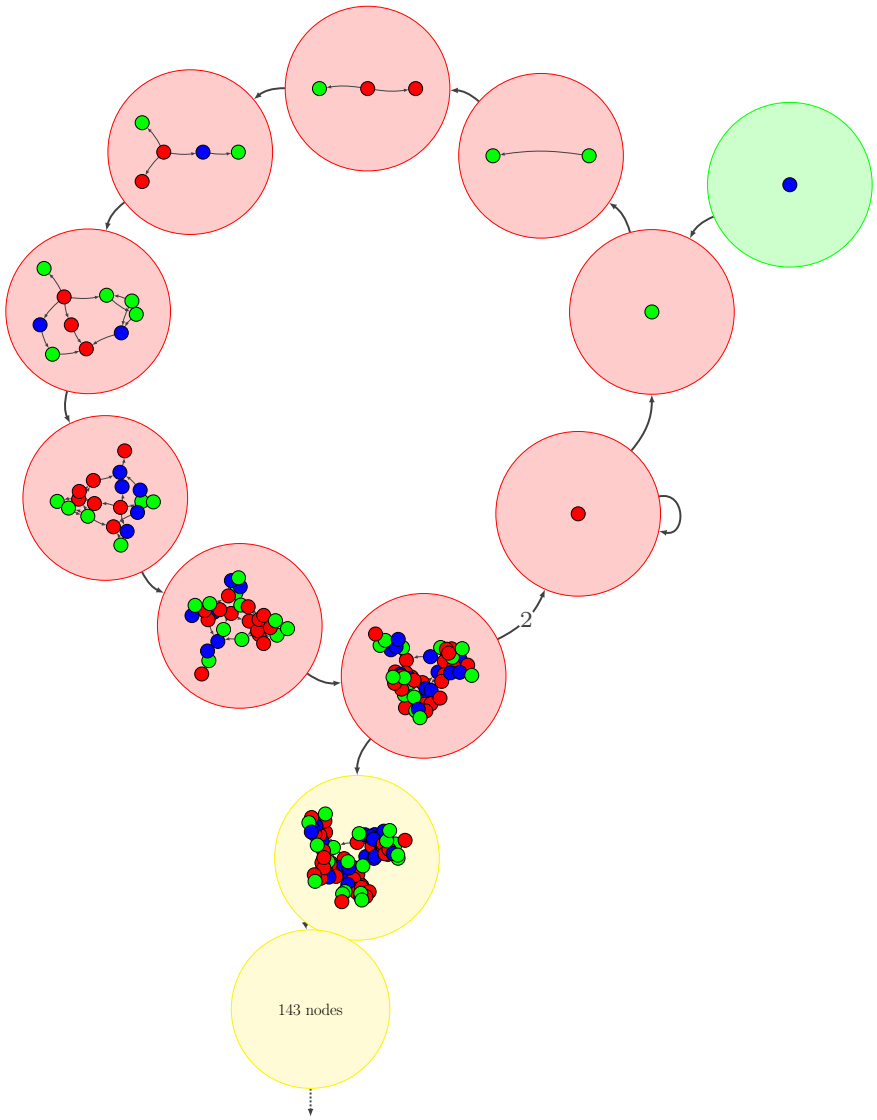


Figure 8.8: In this example the graph grows for 8 steps and on the 9th splits off two single red nodes, which in turn produce single green nodes and the cycle restarts. The "main" graph carries on growing as shown in the two yellow circles at the bottom of the diagram. We stop tracking growth when the graph gets larger than 150 nodes, so it is assumed that the rest of the trajectory displays "runaway" growth (hence these steps being displayed in yellow circles - they are not part of a cycle). Despite the largest graph component exhibiting runaway growth, it also reproduces itself thanks to the "spawning" of two singleton nodes early in the growth process. Note that in this example there are actually two cycles (the singleton red node reproduces itself), but since they are linked we say there is one "cycle group".

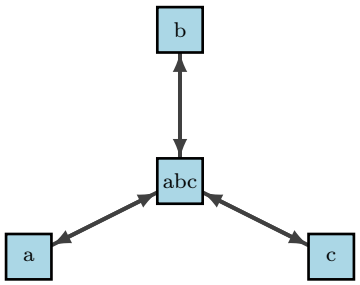
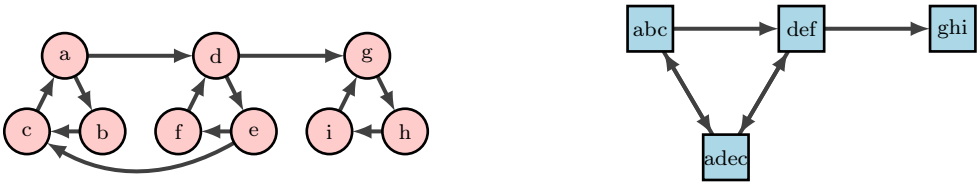


Figure 8.9: A linked cycle graph of the transition graph shown in Figure 8.2. The nodes represent cycles in the transition graph, and the edges represent routes between these cycles.



(a) An example graph-state transition graph (T) (b) The corresponding linked cycle graph

Figure 8.10: There are four cycles in the transition graph in shown on the left. Each of these is represented as a vertex in the linked cycle graph shown on the right.

in Figure 8.9. In this case it is trivially true that the three single-node cycles a,b,c are each linked bidirectionally to the larger cycle abc.

A more interesting example is shown in Figure 8.10. Here there are four cycles in the transition graph: abc; def; ghi; and $adec$. There is a edge from the def cycle to the ghi cycle (because $f(d) = g$), but there is no edge back from ghi to def (because $F(\{g, h, i\}) \cap \{d, e, f\} = \varnothing$).

We could therefore describe the ghi cycle as "parasitic" on the def cycle. By contrast, in the case of abc and def, they are mutually linked by the larger cycle $adec$ with which they both share components (graph-states). We can therefore refer to the set of cycles abc, def and $adec$ as a "linked cycle group". This can be more formally defined as a *strongly* connected component in the linked cycle graph (ghi is not part of this component: it is not strongly connected as there is no edge from it going back to the other nodes).

$$\text{LinkedCycleGroups} = \text{SCC}(\text{LinkedCycleGraph}) \tag{8.23}$$

where SCC is a function giving the set of strongly connected components of a graph. Returning to the example shown in Figure 8.9, we can see that there is a single strongly linked component, and therefore a single cycle group, comprising a, b, c and abc.

We now have several hierarchical levels of abstraction: there are the underlying graph states that we see during a run of the system, $\mathcal{G}_\Omega$; the transition graph between these graph states, T; and the LinkedCycleGraph of cycles within T. Finally there is the set of strongly connected components of the LinkedCycleGraph, namely the LinkedCycleGroups.

We could go one step further and introduce a graph in which the vertices are the LinkedCycleGroups, and the edges between them are observed transtions between underlying graph states. For Figure 8.10b this would be a graph with two nodes and a single egde: $\{abc, def, adec\} \rightarrow \{ghi\}$. In graph theoretic terms, this is the *condensation* of the LinkedCyclesGraph. Although we do not use this representation here, it is useful to bear in mind as a higher-level of organisation, as it provides a visualisation of all the parasitic relationships between cycle groups (relationships in this representation must by definition be parasitic, or the vertices would collapse into a single cycle group). In the case of development rules which generate very complex systems of linked cycles, this higher level of organisation may give a useful coarse-grained description.

8.4 Implementation Details

Given a seed graph g_o an update function f (parametrised by the MLP weights) the system involves repeatedly applying Equation 8.14. In other words, each update step consists of running the update function for all graph components currently in the system and capturing all of the outputs from each update. Since we wish to keep track of the transitions between graph states, we in fact use a recursive update procedure as shown in in Algorithm 2. Starting with the seed graph (for which we use a single node graph for all examples in this chapter), we iteratively apply the update function. After each application, we examine how many connected graphs have been produced. We then apply the update function to each of them in turn, and repeat the process. We record each connected graph in an archive together with a the graph at the previous timestep. This allows us to construct a graph-state transition T. If at any point in the run a graph-state is produced which is already in the archive, we do not need to continue running that element: since the system is deterministic, we already know its trajectory. This means that we only run updates for unique graph-states, effectively applying the deduplicate function (Eq. 8.9) after each update. In order to check whether a graph-state is in the archive we use the same procedure as introduced in Section 4.4.1, in other words first performing a Weisfeiler–Leman graph hash as a quick check to see whether it is possible that they are in the same state; if it is possible, a full isomorphism check is conducted (see Definition 1).

The transition graph T is built as the process is run: each new graph state is added as a node, and each transition between states is added as an edge. Note that we assign each graph-state a unique string ID for these purposes (the Weisfeiler–Leman hash is initially used but modified in the event that the full isomorphism check reveals a hash collision).

The update process may be run until all developmental trajectories have been explored. This is only possible if all trajectories lead to cycles. In most cases, some trajectories will lead to runaway growth. To prevent the system running forever, these trajectories can be abandoned if the size of graph-states breaches an arbitrary upper limit (the max_nodes

Algorithm 2 Recursive DGCA update

```
1: f ← parameterised update function
2: g ← single node seed graph
3: archive ← ∅
4: seed_graph_id ← GenerateID(g)
5: T ← empty digraph                ▷ initialise transition graph (vertices T_V, edges T_E)
6: max_nodes ← 300                  ▷ example value to cut off runaway development
7: procedure RunDGCA(g)
8:     if (g ∈ archive ‖ #(g) = 0
9:         ‖ #(g) > max_nodes) then
10:        return                   ▷ graph seen before/empty/too big
11:    end if
12:    archive ← archive ∪ g
13:    T_V ← GenerateID(g)          ▷ add graph-state id as vertex to T
14:    new_graphs ← f(g)            ▷ run a step
15:    for i = 1 to #(new_graphs) do
16:        T_E ← (GenerateID(g), GenerateID(new_graphs[i]))   ▷ add transition as edge
17:        RunDGCA(new_graphs[i])
18:    end for
19: end procedure
```

parameter in Algorithm 2).

8.5 Example

Figure 8.11 shows an more complex example of a graph-state transition graph (as defined in Equation 8.17). This diagram follows the form of Figures 8.6 & 8.7. The diagram shows the development of system from a single node seed graph (the graph-state in circle A) by application of a single growth rule (found by random experimentation). After various graph division events, the system falls into several interlocking attractor cycles. The linked cycle graph for this example is shown in Figure 8.12, highlighting the relationships between the four cycles.

8.5.1 Comparison with other Generative Systems

In Section 8.2, parallels were drawn between DGCA and subsymbolic or automata AChems. Here we explain this view with reference to the transition graph T. If we treat each graph state as a single symbol, then we can represent the transition graph as an L-system. A minimal example of a transition graph is given in Figure 8.13 which represents three graph

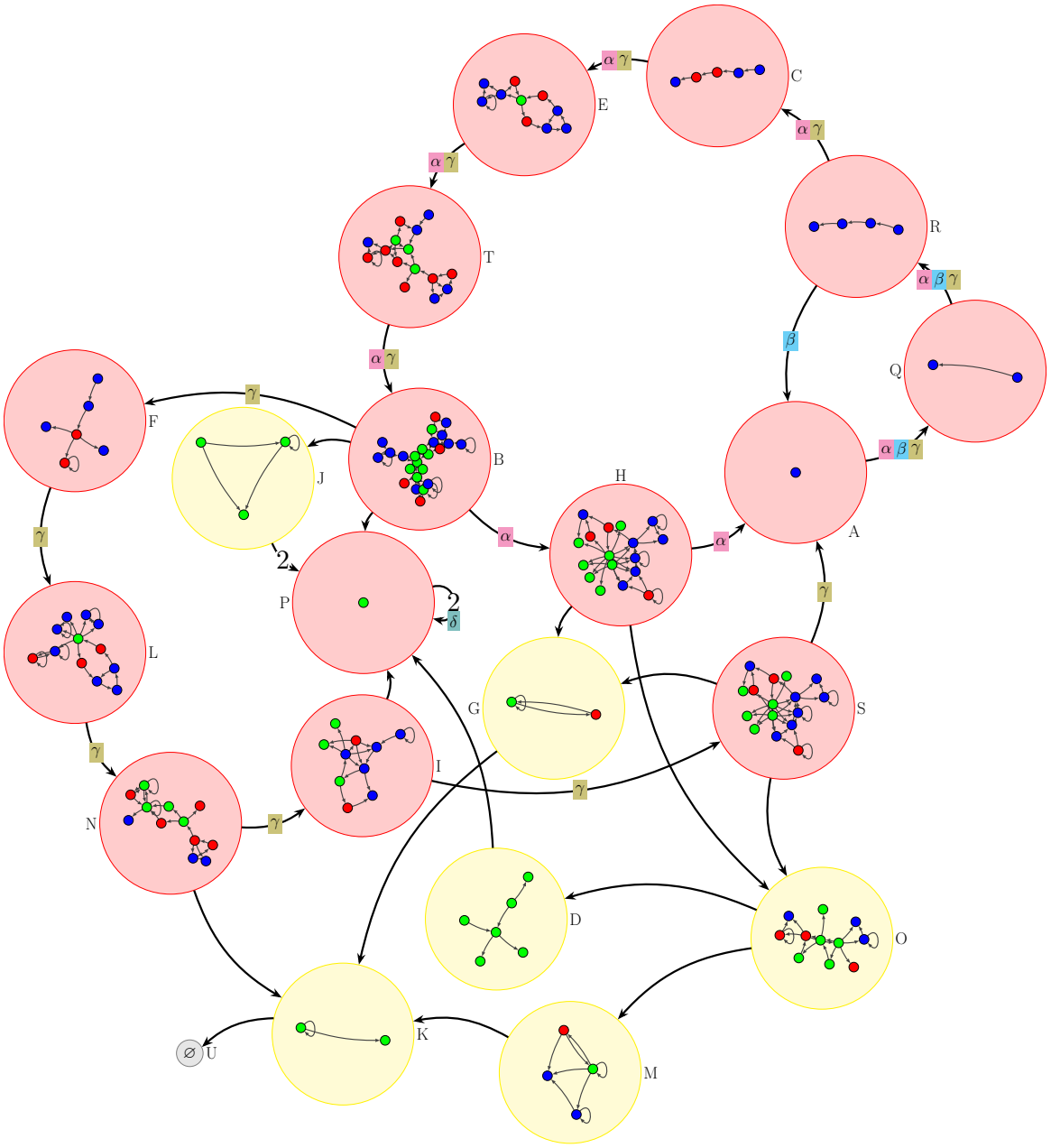


Figure 8.11: We can count 4 cycles in this system, which are highlighted with Greek letters in coloured boxes on the arrows (multiple boxes/letters on the arrows indicates membership of multiple cycles), `AQRCETBH` (α) `AQR` (β), `BFLNISAQRCET` (γ), and `P` (δ). The first three of these are part of one cycle group because they all share elements. The fourth cycle δ is not part of the same group, but is parasitic on cycles α and γ. Note that the yellow circles are not part of any cycles - for example `J` provides a conduit from `B` to `P`, which *are* both parts of cycles, but it is not part of a cycle itself. Similarly, the arrow `I` $\rightarrow$ `P` is not part of a cycle although it links two circles which are in (separate) cycles. Note that `U` is the empty graph, where all nodes have been removed.

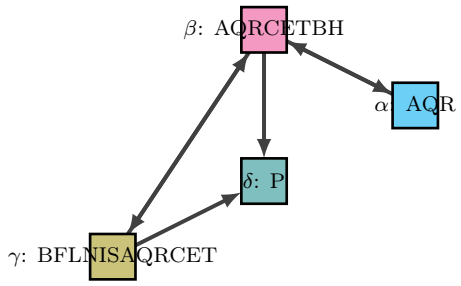


Figure 8.12: A linked cycle graph of the transition graph shown in Figure 8.11. The nodes represent cycles in the transition graph, and the edges represent routes between these cycles. The cycle δ is parasitic on cycles β and γ, indicated by the fact that it has only incoming edges from these nodes. The cycles α, β and γ constitute a linked-cycle group with three members.

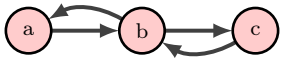


Figure 8.13: A simple graph-state transition graph.

states as the symbols a, b, and c. We could rewrite this as L-system style rules:

$$\begin{aligned} a &\rightarrow b \\ b &\rightarrow ac \\ c &\rightarrow b \end{aligned} \tag{8.24}$$

Starting with the axiom (or seed graph state) a, this system would evolve as follows: $a, b, ac, bb, acac, bbbb, acacacac$ and so on. The symbol string at each timestep represents the overall system state and gives an indication of the quantity of each graph state. Note that unlike a true L-system, the ordering of symbols in the string is arbitrary since the DGCA system does not specify any ordering or spatial embedding for the products of a graph division. The symbol string should therefore more properly be regarded as an unordered multiset of symbols. In this sense it is more like an AChem, as noted in Section 8.2.1, where reaction rules may have the form $b \rightarrow \{a, c\}$, giving the products of a reaction as a multiset. As already noted, whether represented as AChem reactions or L-system rules, all production rules based on a DGCA graph-state transition graph will have exactly one symbol on the left-hand side.

The more complex set of L-system rules derived from the graph state transition diagram shown in Figure 8.11 is given in Equation 8.25. Note that the only terminal symbol used is $\varnothing$, the empty graph state. All of the other symbols are non-terminal, reflecting the fact that all of the attractors in the graph state transition diagram are cyclic rather than point attractors. (Graph state P is almost in in a point attractor but divides to create two copies of itself: $P \rightarrow PP$). As a result the L-System would run forever, producing an ever-growing

string of symbols. The first 10 steps of this run are shown in Equation 8.26

$$\begin{array}{ll} A \rightarrow Q & K \rightarrow \varnothing \\ B \rightarrow FHJP & L \rightarrow N \\ C \rightarrow E & M \rightarrow K \\ D \rightarrow P & N \rightarrow IK \\ E \rightarrow T & O \rightarrow DM \\ F \rightarrow L & P \rightarrow PP \\ G \rightarrow K & Q \rightarrow R \\ H \rightarrow AGO & R \rightarrow AC \\ I \rightarrow PS & S \rightarrow AGO \\ J \rightarrow PP & T \rightarrow B \end{array} \tag{8.25}$$

$$A, Q, R, AC, QE, RT, ACB, QEFHJP, RTLAGOPPPP, ACBNQKDMPPPPPPPP, \tag{8.26}$$

This abstraction of the DGCA state transition graph is mainly useful for analysing the quantities of different graph-states produced by the system after a given time period. As shown, in the case of this system, the symbol / graph-state P rapidly increases. If all of the development paths are fully mapped out, which is possible if they all end up in attractors (see Section 8.4), a comprehensive set of production rules can be recorded. At that point there is no need to run the actual DGCA system further, and the production rules may be used to simulate further running of the system. If not all components end up in attractors, with some following runaway growth trajectories, these trajectories could be cut off at an arbitrary point (above a certain number of nodes), and these graph states could be regarded as terminal symbols in the L-system.

The set of production rules *emerges* from the sub-symbolic dynamics of the DGCA system. As shown in Chapter 5 and further explored in Chapter 9, these dynamics are evolvable via changes to the growth rule which is represented as real-valued MLP weights. This offers an intriguing approach to evolving L-system rule-sets. Evolving such symbolic rules directly can be difficult as changes to individual rules may have widely varying impacts, depending on whether the rule is applied early or late in the developmental process (McCormack, 2008).[4]

[4]It is easier to implement "smooth" changes with stochastic and parametric L-systems as probabilities or parameter values can be changed by small amounts.

8.6 Conclusion

This chapter has introduced a formalism for describing the higher-order behaviour of DGCA where we treat each disconnected graph component as a separate entity in state-space. The state transition graphs can create complex interlocking cycles as an emergent property of the developmental system. The higher-order relationships between these cycles can be described with linked cycle graphs. This formalism highlights the links between DGCA and sub-symbolic AChems and generative grammars.

The next chapter will use this formalism in an analogy of cell division and differentiation. This is inspired by Kauffman's use of different RBN attractor cycles to model different operational modes of GRNs, corresponding to differentiated cells (Kauffman, 1969).

Chapter 9

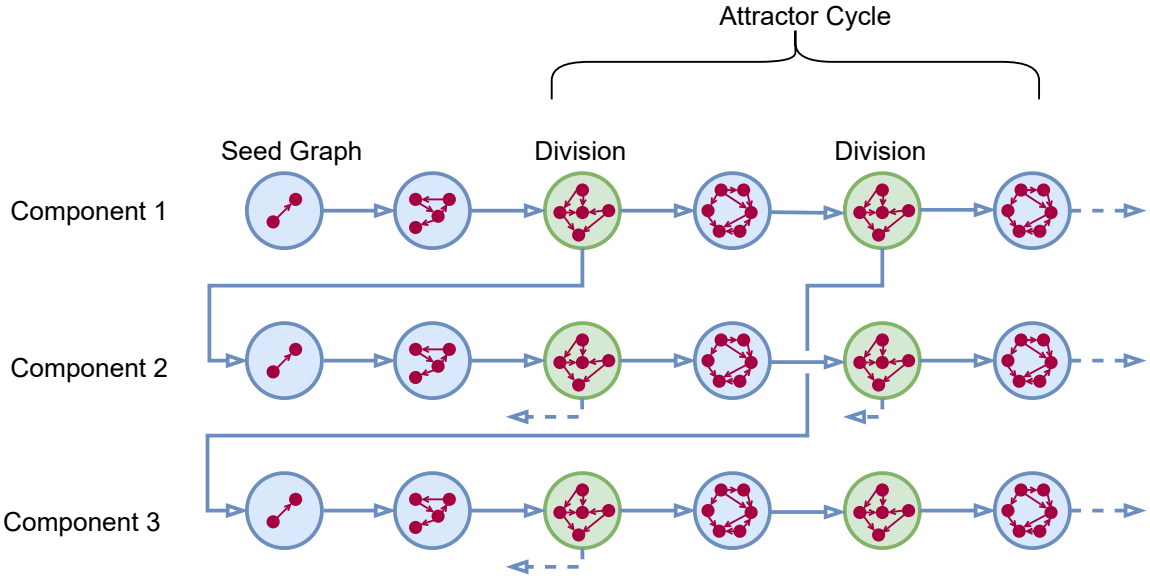
Cell Division and Differentiation

9.1 Introduction

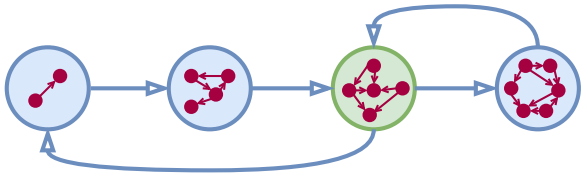
Chapter 8 introduced a new formalism for describing the phenomenon of dividing graphs in DGCA. Each separate graph component pursues its own trajectory through state space. These trajectories can bifurcate as a result of graph division, adding new entities to the system. Trajectories may enter attractor cycles, lead to the zero-node graph (a special case of a point attractor) or continue with unbounded runaway development. Dividing trajectories can lead to interlocking attractor cycles (these are attractors at the level of individual components, not the whole system).

One phenomenon which is frequently observed when running the system is that a small "seed" graph can develop into a larger graph and at a certain point may split off a small component which is the same as the original seed graph. Since the system is deterministic, this new seed graph will follow exactly the same process. Sometimes the larger "parent" component enters a point or cyclic attractor in which it splits off a seed each time it goes round the cycle. A diagram of this prototypical case is shown in Figure 9.1. There are various different biological analogies which could be drawn with this process, including most obviously that of a whole organism reproducing itself. This would necessarily be asexual reproduction since, as mentioned, there is no way for the graph components or "organisms" in this analogy to encounter each other again after they have divided. In the prototypical example described here, bacterial division could be a good analogy.

However, in many cases, when a graph divides it is *not* the case that a perfect copy of the original seed graph is produced. This can lead to the kind of situation shown in Figure 9.2, where a seed graph divides into two or more components at some point during its developmental trajectory. The components produced by this division may continue developing and eventually reach separate attractors. This invites comparison with biological cell differentiation, where a pluripotent stem cell divides and after some intermediate steps a variety of specialised cells are produced. In this analogy the specialised nature of the cells



(a) Full schematic of graph division.



(b) More abstract representation of the same process.

Figure 9.1: A prototypical graph division is shown in (a). Each large blue/green circle represents the state of a graph component at a point in time (the actual graph is shown inside the circle). A seed graph develops for two steps before dividing into two components (indicated by the fact that two arrows are coming out of the green circle). One of the components is a copy of the seed graph which undergoes the same growth process. The other enters a two step attractor cycle, dividing every other step. All division points are shown in green, with two arrows coming out of them. To save space, not all products of division are shown. The same system is shown in (b) in an abbreviated form: when a previously seen graph component is created, the arrow points back to that circle. This depiction highlights that there are two interlocking cycles.

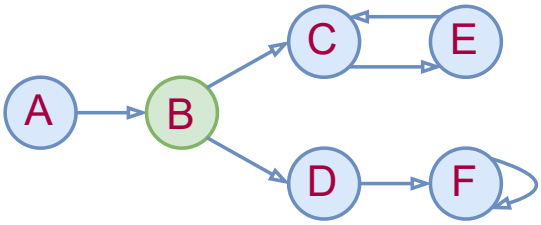


Figure 9.2: A different attractor structure. This diagram uses the same representation as Fig. 9.1b except that graph-states are abbreviated to symbols. Seed graph A develops into B, which splits into components C and D. Component C enters an attractor cycle (C–E), and component D goes through one more developmental step before reaching a point attractor, F.

is represented by the fact that the graph components may have very different structures.

By treating disconnected graph components produced by the DGCA system as "cells" and examining their transient and attractor dynamics individually, we can gain insight into the complexity generated by simple developmental rules in the DGCA system. The biological analogy is necessarily loose and this is certainly not intended to be a realistic model of biological cell division. Indeed, other analogies are possible, including the AChem interpretation discussed in Chapter 8.

However, the cell analogy is useful because it allows discussion of division and differentiation in the same framework. This chapter examines both aspects in tandem, analysing the complex graph division process and the variety of differentiated graph structures it can create.

A more pragmatic motivation for this work is that the growth rule of a DGCA can be considered a highly compressed encoding of a graph structure. With only the seed graph (which can be a single node) and the growth rule (which in the 3-state DGCA v2 system used in this chapter, consists of 180 floating point numbers), it is possible to develop a graph of arbitrary size through the deterministic running of the DGCA. Since the graph can divide, we could say that a single growth rule can encode a collection of graphs. It would be noteworthy if the graphs produced had appreciably different structure from each other. This would mean that a single compressed encoding could be unfolded into a number of distinct graphs which could have different functional purposes.

9.2 Background

9.2.1 Cell Division and Differentiation

C. H. Waddington's influential metaphor of the *epigenetic landscape* imagines developing cells as marbles rolling down a slope containing branching valleys (Waddington, 1957). They eventually come to rest at the lowest points of the landscape, representing fully differentiated cell types. This view has enabled cell biologists to draw on the dynamical systems literature

when modelling cell development. This usually takes the form of continuous dynamical systems that are open to environmental influence (e.g. Brackston et al. (2018)).

One example of a *discrete* dynamical system being used to model epigenesis is the Random Boolean Network (RBN) model of Kauffman (1969). The overall state of the RBN, which encompasses all of the individual node states, is updated at each timestep. The system may move from its starting state, along a transient and end up in an attractor. The RBN is used as a model of a gene regulatory network (GRN). Different attractor cycles in an RBN state space are analogised different operational modes of the GRN, indicating different cell types. However, since RBNs are deterministic, any one starting state leads to only one attractor cycle. In order to model movement between cell types, Kauffman introduces small perturbations, effectively making the RBN into an open dynamical system. Some of these external perturbations are able to push the system into different basins of attraction. This is similar to Waddington's original conception of the epigenetic landscape, in which the "marbles" are pushed into different valleys in the slope by exogenous forces including environmental factors and external signals. Cell division is not modelled in RBNs.

Here, we use unequal graph division in DGCA as an analogy of asymmetric cell division (ACD), giving an endogenous mechanism for moving between cycles. ACD has been documented in *Drosophila* neurogenesis, where segregation of transcription factors between daughter cells leads to differing cell fates (Matsuzaki, 2000). Even the differing cytoskeletons of daughter cells may differentially affect regulatory pathways (Dye et al., 1998). Recent work on single-cell transcriptomics reveals further mechanisms by which differing transcription factor regulatory networks (TFRNs) are expressed in daughter cells (Zhang et al., 2023). In the DGCA analogy, the actual graphs "inside" cells could be thought of as representing these different TFRNs, or as a whole complex of cell characteristics (e.g. including the cytoskeleton).

In the DGCA analogy of cell development we start with a seed graph-state, which can be thought of as a germ cell or pluripotent stem cell. A graph-state represents the state of the cell at a particular point in time. Some of these states are transitory and can be thought of as states that a cell passes through on its way to becoming specialised (for example, graph-state D in Figure 9.2). Other graph-states form part of an attractor: these can be thought of as the states experienced by a fully specialised "final" cell type. It is not the case that a specialised cell is always in a fixed state. As in the RBN model, different cell types are represented by different attractor cycles.

In Figure 9.2, the cycle $C \leftrightarrow E$ represents one cell type which can exist in either state C or state E. It is also possible to have a differentiated cell which exists in only one state, as in graph-state F, which is in a point attractor.

In using the DGCA as an analogy for cell division and differentiation, no external perturbation or signalling is needed in order for multiple attractors in graph-state space to be explored (unlike the RBN model which uses external noise to move between attractors).

When a graph divides asymmetrically, the products of division can follow their own trajectories, potentially into different attractors in graph-state space. The DGCA model is a *closed* and deterministic discrete dynamical system.

The biological analogy breaks down when we consider that individual graph-states can be shared between cycles, as is the case with the third graph-state in Figure 9.1b. In biology, it is hard to imagine any meaningful sense in which it could be said that a liver cell is in the "same state" as a neuron, for example. It is worth noting that in the DGCA system, a graph-state can form part of more than one cycle only if it leads to a division. A "terminal cycle", which does not lead to any division (such as $C \leftrightarrow E$ in Figure 9.2), can contain only graph-states which uniquely belong to that cycle.

9.2.2 Graph motif analysis

In order to evaluate the level of "cell differentiation" that the DGCA system can produce, we need a measure of the difference between graph-states. Various options for this exist including the graph edit distance, the adjacency spectral distance, and the difference between various graph features, such as degree distribution (Wills & Meyer, 2020). Some of these are not appropriate for comparing graphs of different sizes, as we wish to here, whilst others focus on global properties rather than local structure. Building on Chapter 7, we use the difference between triad significance profile (TSP). The TSP is frequently used to characterise graphs in the systems biology literature (Alon, 2007; Milo et al., 2002). As previously outlined in Chapter 7, the TSP summarises the graph microstructure by counting the occurrence of each of the thirteen possible three node subgraphs (triads) and comparing the observed number with the counts in an ensemble of randomised graphs of the same size and degree distribution. The result is expressed as a normalised vector of 13 Z-scores indicating the enrichment or diminution of each triad compared to the random graphs.

Since the TSP of a graph is a vector of 13 numbers we use the Mean Absolute Difference (MAD) between two TSP vectors as a measure for the difference between two graphs:

$$MAD = \frac{1}{13} \sum_{i=1}^{13} |\mathbf{x}_i - \mathbf{y}_i| \tag{9.1}$$

Chapter 7 used DGCAs to create networks with TSPs similar to various real biological networks, by using a genetic algorithm to search over the DGCA growth rules. That study did not analyse graph division so calculated TSP values for whole graphs (potentially containing disconnected components). Here we analyse graphs produced by a DGCA growth rule at the level of connected components. We aim to use a *single* growth rule to create multiple graphs that have significantly different TSPs, as an analogue of cell differentiation. Using the TSP to describe the graphs works well with this analogy, since it highlights that different "cell types" may have different functional properties (see the discussion in Chapter 7

on the possible link between motif occurrence and the functional properties of networks).

9.3 Experiments

9.3.1 Maximising Differentiation

We first conduct an experiment to investigate whether a single DGCA growth rule can produce what we have described as "cell differentiation": whether it can produce separate graph-states that have significantly different structure from each other.

Because we use cycles in the graph-state transition diagram as analogy of cell types, we wish to examine the difference between graph-states on different cycles in the graph-state transition diagram. We choose the largest graph-state (by number of nodes) as the representative graph-state for that cell type. The calculation of TSPs is computationally expensive (since it involves enumeration of all three-node subgraphs) so we calculate them only for the two largest graph-states on different cycles in the whole transition diagram. The chosen graph states must also meet some conditions of size and connection density, because the TSP metric does not work well on graphs that are too small or too densely connected. The graphs must be between 100 and 300 nodes in size and have a connection density between 0.002 and 0.04.

We use a genetic algorithm (GA) to maximise the MAD between the TSPs of these two chosen graph-states. The "genomes" in the GA are DGCA growth rules, or in other words the weights and biases of the neural networks used in the DGCA node state and graph structure update. A 3-state DGCA is used in these experiments, and SLPs are used in the update rules, resulting in 180 parameters (see Section 6.4.4 for explanation).

We use a population size of 100 and use "natural elitism", conserving all individuals from the parent population that have a higher fitness than the offspring population into the next generation.

The results are shown in Figure 9.3. After only a small number of generations, the GA was able to find growth rules that could generate graph-states on separate cycles ("cells") with significantly different structure from each other. The MAD between two biological TSPs from very different families, the *E. Coli* GRN and the *C. Elegans* NN (see Figure 7.3) is only 0.25, which was exceeded in this experiment after 18 generations.

9.3.2 Maximising Cycles and Differentiation

We hypothesise that an individual growth rule will have more likelihood of creating highly differentiated "cells" if it produces more and longer cycles. Exploring more attractor cycles in the graph-state space gives more opportunity to encounter graph-states with very different structure. Furthermore, the longer the length of the cycles, the more steps the cell has available to develop differentiated structure.

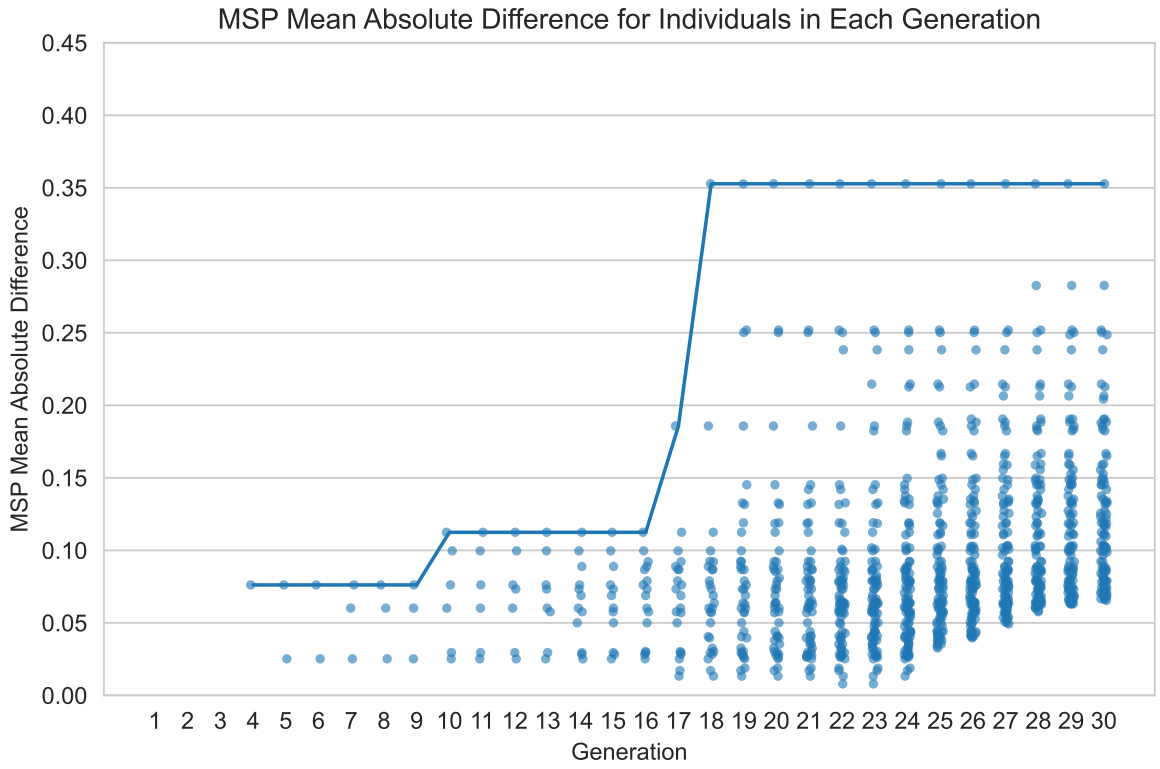


Figure 9.3: Maximising Differentiation experiment. The MAD between the TSPs of the two largest graph-states on different cycles are plotted for each individual in columns for each generation. Not all individuals have two valid graphs on different cycles to compare. The line shows the best (highest) MAD in each generation.

To test this hypothesis, the second experiment uses a multi-objective GA to maximise TSP MAD between the two largest graph-states on different cycles (as before) and also to maximise the number of cycles and the average cycle length.

We use the NSGA-II algorithm introduced by Deb et al. (2002), which again uses "natural elitism" but does so by performing a non-dominated sort on the parent population and offspring population together and keeps members of the successive non-dominated (Pareto) fronts until the next generation is full. It also maintains population diversity by using a crowding distance metric for selection.

The results are shown in Figure 9.4. As hoped, in the multi-objective context the GA is able to maximise TSP MAD faster, breaching the biologically relevant 0.25 threshold after only 6 generations, and reaching a high point of 0.43 in the 15th generation. This suggests an interesting interplay between the structural growth of the graphs and the cycle dynamics, which are an emergent property of the system.

The graph-state transition diagram for the best individual found in this experiment is shown in Figure 9.5a. This is the same kind of diagram as in the simplified example in Figure 9.1b. In total there are 70 cycles (graph-states may be members of more than one cycle), with an average cycle length of 35.8 steps. The two largest graph-states on separate

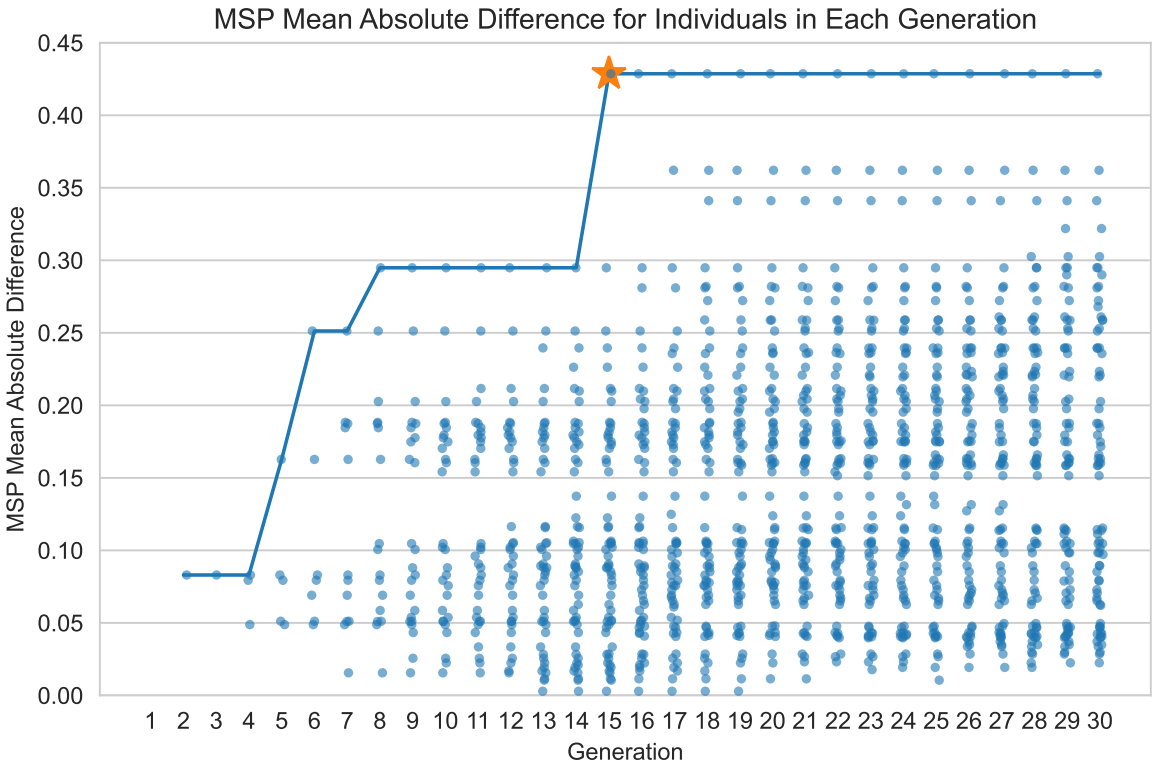
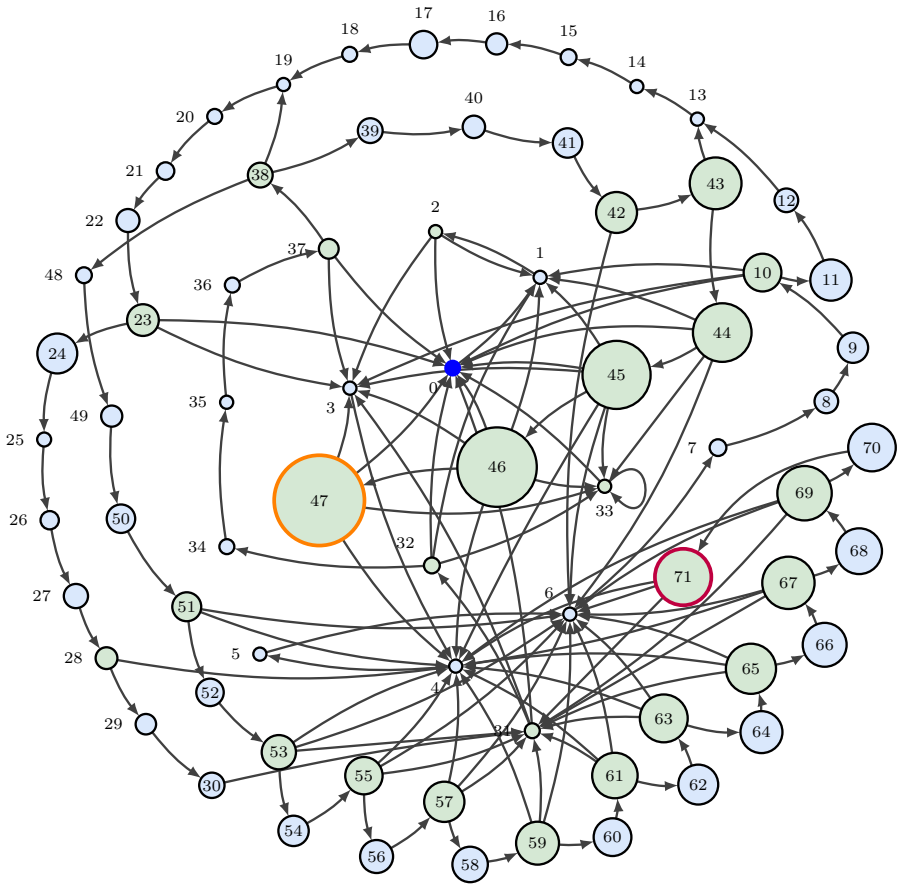


Figure 9.4: Maximising Cycles and Differentiation experiment. Just the TSP MAD is plotted for ease of comparison with Fig. 9.3, although three features are used in the multi-objective GA. The individual growth rule which generates maximum differentiation (TSP MAD) was found in generation 15 and is highlighted with a star. The state transition diagram for this individual is shown in Fig. 9.5

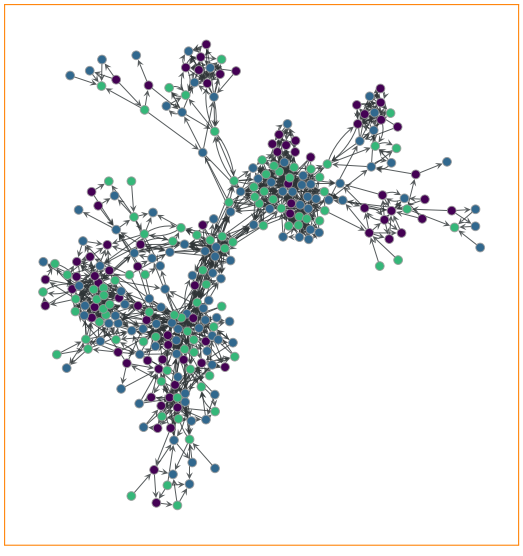
cycles are those numbered 47 and 71, and circled in orange and purple respectively, containing 292 and 114 nodes respectively. (Whilst graph-states 43–46 are bigger than graph-state 71, they are on the same cycle as 47 so are not considered a different "cell type".) The actual graph-states 47 and 71 are shown in Figures 9.5b & 9.5c, along with their very different TSPs in Figure 9.6.

9.3.3 Natural TSP Targets

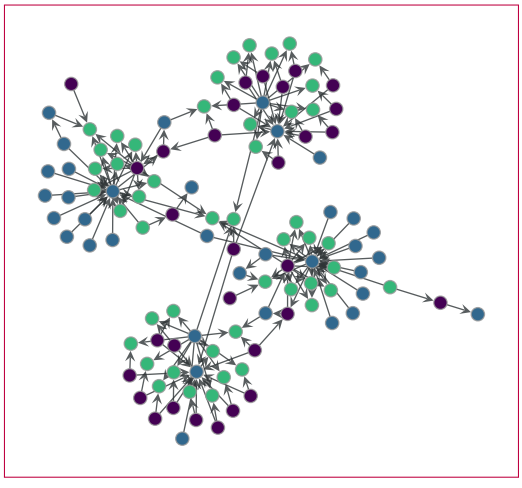
Having shown that it is possible to find single growth rules that produce separate graphs with dissimilar structures, we next evaluate whether a single growth rule can produce graphs which are structurally close to two or more target graphs. Chapter 7 used a GA to search for growth rules that generate graphs with TSPs close to those of two natural networks: the *E. Coli* GRN and the *C. Elegans* NN. However, that study conducted two separate searches: each trying to optimise for proximity to only one target. Since here we are concerned with differentiation, the present experiment uses multi-objective optimisation (again using NSGA-II) to find a single growth rule that produces two separate graphs each of which has a small TSP distance to its respective target. We once again use *E. Coli* GRN and the *C. Elegans* NN



(a) Graph-state transition diagram



(b) Graph-state 47: orange circle in (a)



(c) Graph-state 71: purple circle in (a)

Figure 9.5: (a) shows the graph-state transition diagram for the best individual growth rule found in Experiment 2 (highlighted with a star in Fig. 9.4). Each circle represents one graph-state and has an area proportional to the number of nodes in the graph. Arrows represent transitions between graph-states on each DGCA timestep. The starting state (single node seed graph) is highlighted in bright blue and labelled 0. Graph-states which divide are coloured green (as in Figs. 9.1 & 9.2). The two largest graph-states on different cycles are 47 and 71 and are circled in orange and purple respectively. Their actual graph states are shown in (b) and (c). Their two Triad Significance Profiles are shown in Figure 9.6.

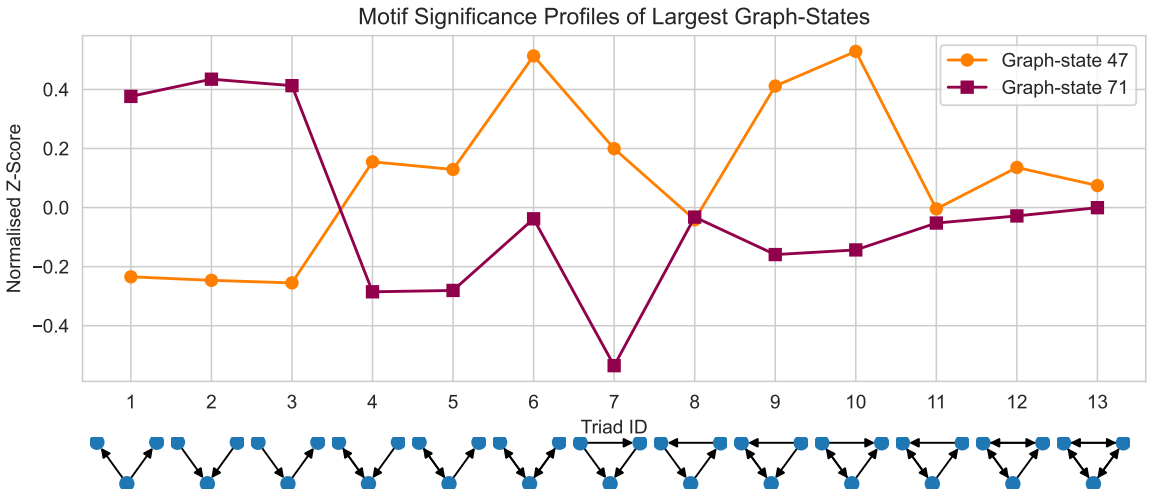


Figure 9.6: Triad Significance Profiles of graph-states 47 and 71 from Figure 9.5 showing the enrichment or diminuation of certain motifs compared to random graphs with the same degree distributions.

as target networks. This is a challenging goal as these networks have been identified by Milo et al. (2004) as belonging to different "families" because of their very different TSPs. If the DGCA system can use a single growth rule to develop two networks matching each of these profiles, this would be a good indication of the power of the the division and differentiation process described here.

As in the second experiment (Fig. 9.4), it was found that maximising the number of cycles and average cycle length sped up convergence. The GA used in this experiment therefore has *four* objectives: minimising the TSP error of the two largest graph-states (on different cycles) with respect to each of the two target TSPs, and maximising the number and length of cycles.

In order to prevent the evolution of a single graph-state with an TSP somewhere in between the two target TSPs, we allow each graph-state to be compared to only one or the other targets, not to both.

We again use NSGA-II with a population size of 100. We use the same size and connection density conditions as before for choosing "valid" graph-states for which to calculate the TSP.

The results of the experiment are shown in Figure 9.7. Note that in this experiment the aim is to *minimise* the MAD of the found TSPs versus the target TSPs, whereas in the first two experiments it was to *maximise* the mutual MAD of the TSPs of two graph-states produced by an individual growth rule.

Figure 9.9 shows the actual TSPs of graph-states produced by the best growth rule, which was found in generation 16 of the GA. The red lines indicate the generated TSPs and the blue lines indicate the target TSPs of the *E.Coli* GRN and *C.Elegans* NN. The TSP MADs versus each target are 0.06 and 0.10 respectively. Whilst this is a greater error than

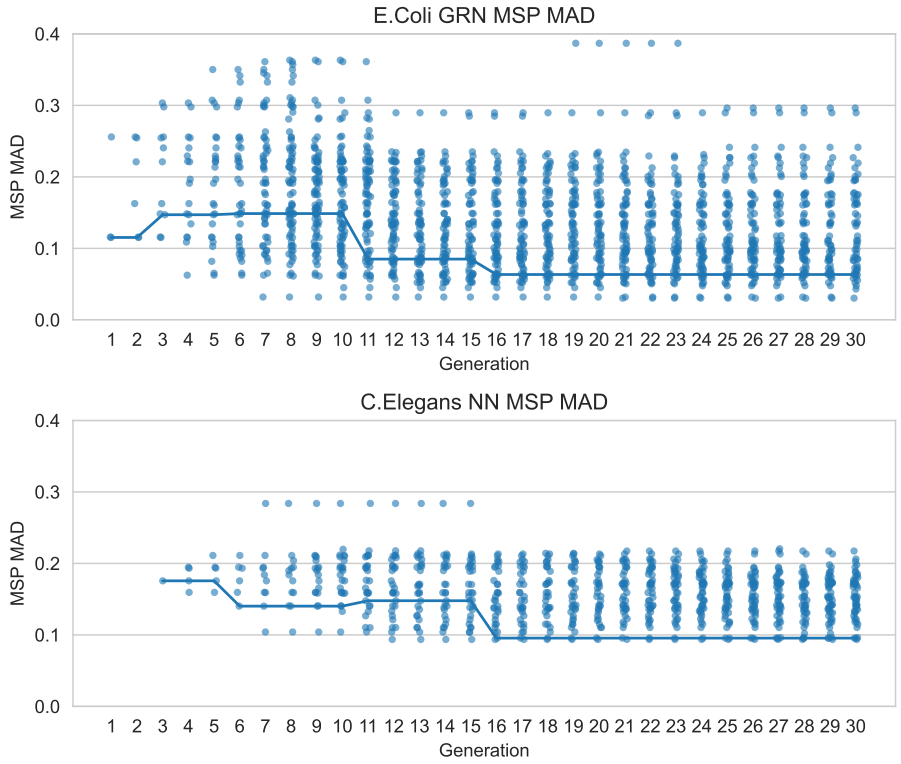


Figure 9.7: Swarm plots showing the TSP MAD for individuals in each generation vs. the E.Coli GRN TSP (upper chart) and the C.Elegans NN TSP (lower chart). The lines on these charts indicate the best individual in each generation, averaging across both MADs.

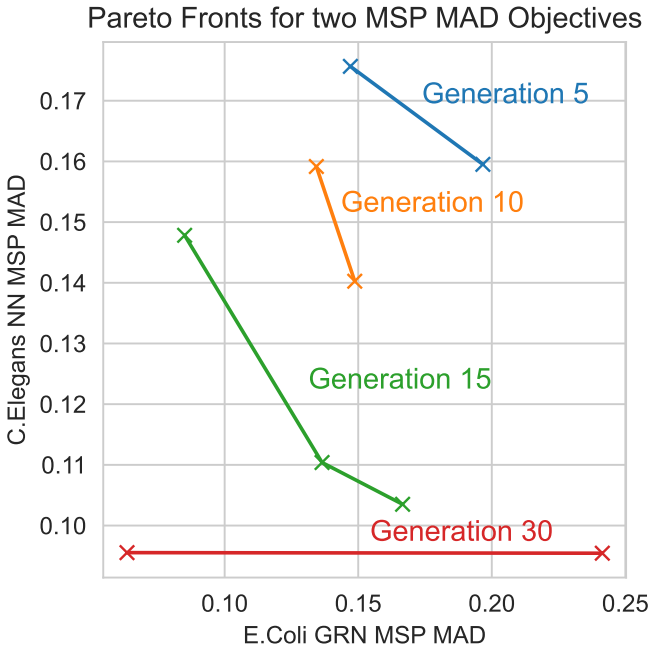


Figure 9.8: An alternative view of the results shown in Figure 9.7. The chart shows the Pareto fronts across the two TSP objectives for selected generations.

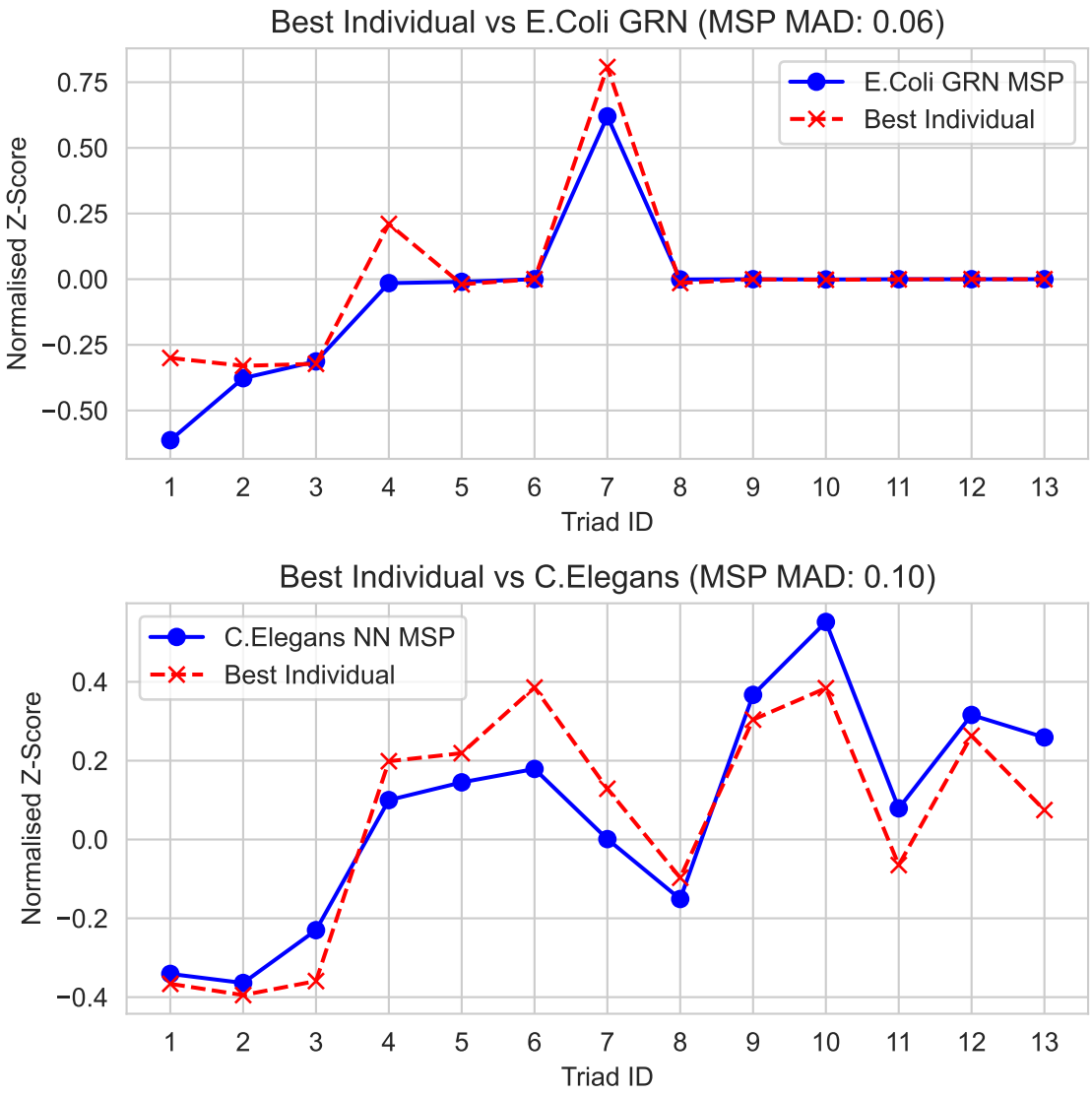


Figure 9.9: The actual TSPs of the two largest graph-states on separate cycles (generated by a single genome / growth rule). They are plotted against their most closely matching target TSPs the *E. Coli* GRN and *C. Elegans* NN respectively. See Fig. 9.6 for the meaning of the triad IDs along the x-axis.

was found in Chapter 7 when evolving separate growth rules for each target, it is nevertheless noteworthy that a single growth rule can generate two graphs with TSPs close to these two very different targets.

9.4 Conclusion

This work has illustrated behaviour of the DGCA system which is analogous to cell division and differentiation. The fact that graphs may divide in the DGCA system is used as an analogy of cell division. Each of the products of a division then follows its own trajectory

through the graph-state space, and may undergo further divisions in the future. The trajectories may lead to attractor cycles, which we use as an analogy of fully differentiated cell types.

Normally in a deterministic dynamical system such as DGCA, we would expect that given a starting state and an update rule, at most one attractor could be reached. However, each time a graph-state undergoes division, it effectively gives us several new "starting states" each of which may continue into a separate attractor. In a sense this is an extension of Waddington's epigenetic landscape metaphor: each marble can now divide (asymmetrically), and the products of this division may end up in different grooves in the landscape (ie. different basins of attraction).

The number and length of cycles are emergent properties of the system, but surprisingly we are able to evolve them. We also uncover an interesting interplay between the cycle dynamics and the structure of the graph-states produced, finding that greater differentiation of structure is possible when there are more and longer cycles.

The final experiment shows that a single growth rule can be evolved to produce two graph-states which are close to the TSPs of two very different target networks. This highlights that the system can "encode" *multiple* network topologies using a *single* DGCA growth rule: this could have applications in compressing network representations.

9.4.1 Future Work

Owing to the computational cost of calculating TSPs, this work has only examined two of the graph-states produced by each growth rule. Future work could examine more of the graph-states to evaluate whether a single DGCA rule can generate whole ensembles of structurally distinct graphs. In order to achieve this it may be desirable to explore other metrics for the difference between graphs, such as the graph edit distance, which may be cheaper to calculate than the TSP MAD.

Something which has not been explored in this work is the *rate* of production of different cell types. It may be that some attractor cycles are more frequently reached than others by the products of division. Since we are analogising attractor cycles as cell types, we could analyse the total numbers of each type of cell produced after running the system for a fixed time. This could lead to the emergence of common cells and rare cells, allowing the metaphor to be extended towards morphogenesis (although since the graphs are not spatially embedded this would not be a complete model of morphogenesis).

Chapter 10

Reservoir Computing with DGCA

10.1 Introduction

As demonstrated in previous chapters, DGCA can be used to grow graphs with a great variety of structures and growth dynamics. This chapter seeks to put those features of the system to computational use. The obvious way to use graphs as computational substrates is by interpreting them as neural networks. Throughout this thesis, DGCA has been used to generate directed graphs: this makes it easy to use them as ANNs. The graphs are not constrained to be acyclical, so in most cases will contain recurrent connections. This means they could be used as RNNs. However, training RNNs using backpropagation-through-time (BPTT) can be slow and computationally expensive. Furthermore in some cases BPTT can fail to converge as gradients become too small. The reservoir computing (RC) paradigm, in the original Echo State Network (ESN) form in which it was introduced by Jaeger (2007), overcomes some of these problems by using a randomly connected RNN as a "reservoir" whose weights remain untrained. Only the weights of a linear "readout layer" are trained through simple ridge regression, which is comparatively computationally inexpensive and fast.

Although the internal weights of the reservoir are set randomly, some reservoirs perform better than others. The question of the influence of network topology on the effectiveness of RC was highlighted as an important area for future research by Goudarzi and Teuscher (2016). This serves as the motivation for the work in this chapter. As shown in Chapter 7, DGCA is capable of producing networks with widely different microstructure, as measured by triad significance profile (TSP). In biological networks, it is hypothesised that far-from-random motif distributions may have been selected by evolution because of the computational advantages they bring (see Section 7.2.2). Similarly, it is hoped that growing networks using a DGCA-based evo-devo process may create good computational substrates for RC.

The experiment shown in this chapter uses an evo-devo approach to generating high quality ESNs using DGCA. The learned readout weights of each node are used to influence each

developmental step, blurring the boundaries between evolution, development and learning (phylogeny, ontogeny, and epigenesis).

Since the experiment in this chapter involves retraining the network after each developmental step, training speed is crucial. This makes the ESN approach a good fit for this work. Furthermore, the training of the readout layer gives each neuron a readout weight: this can be fed back to inform the DGCA growth process. This allows the developmental trajectory of the network to be influenced by the learning.

This chapter is structured as follows. It begins with a discussion of the interplay between evolution, development, and lifetime experience (phylogeny, ontogeny, and epigenesis): this forms the biological inspiration for the experiment in this chapter. This is followed by a brief introduction to ESNs and the time series prediction task which is used. The experimental setup is then described, including how graphs generated by DGCA are converted into ESNs in a deterministic fashion. The results of an evo-devo-learning experiment are then shown and discussed.

10.1.1 Biological Inspiration

In the evo-devo experiments shown in this thesis up to this point the development has been entirely determined by the genome. DGCA has been used as a closed deterministic dynamical system. However, during the development of biological organisms the growth process is *not* closed to outside influence. It may be subject to environmental influences, such as temperature, light levels, chemical concentrations and so on. It may also be influenced by spontaneous endogenous activity. One example of this is the spontaneous neural spiking which guides the development of the mammalian visual system. Shatz (1996) discovered that the ganglion cells in neonatal ferret retinas showed waves of spontaneous spiking activity passing across the retina in different directions. Ferrets were chosen for the experiment because they are born in a very immature state and their eyes remain sealed closed for $\sim$ 30 days. The retinal activity is therefore not caused by external sensory input, but is spontaneously generated by the cells. Although the electrical activity is endogenous rather than being the result of the cells being exposed to light we can nevertheless think of this as feedback from "experience" in the sense of the cells being exposed to the same modality of information that they will experience from sensory input. In the terminology of ANNs, the spontaneous electrical activity could be thought of as "synthetic training data". Shatz (1996) concludes:

> "Such a pattern of firing could help to refine the topographic map conveyed by ganglion cell axons to each eye-specific layer in the LGN. [...] Thus, even before the onset of vision, the retina spontaneously generates stereotyped patterns of correlated firing that are entirely appropriate to subserve the process of activity-dependent sorting of connections."

Electrical activity is not only used to guide neural growth and connectivity, but also influences synaptic pruning (Sakai, 2020). This pruning continues until adolescence in humans, resulting in adult brains having 15% fewer neurons than infant brains. The activity which influences development at this stage *does* depend on sensory input.

It is a mistake therefore to think of completely separate 'development' and 'learning' phases in neurogenesis, with deterministic, genetically encoded development occurring before birth and experience-dependant learning occurring after birth. In fact the structural development of the brain is influenced by synthetic or real 'sense data' both before and after birth (Hiesinger, 2021). More generally, both during embryogenesis and regeneration, environmental as well as endogenous physiological inputs can regulate development (Sullivan et al., 2016).

This provides the inspiration for the experimental setup used in this chapter, where the experience of learning influences each step of development of the network. A feedback loop is created between development and learning, with the network's learning at one point in time able to influence the next step of development, which in turn changes the network's ability to learn at the following timestep. The actual setup is explained in detail in Section 10.2, after a review of Echo State Networks.

10.1.2 Echo State Networks

We use a minimal implementation of an ESN as outlined in Lukoševičius (2012). Equations 10.1 to 10.3 are adapted from that source:

$$\mathbf{x}_{t+1} = (1-\alpha)\mathbf{x}_t + \alpha f(\mathbf{W}\cdot\mathbf{x}_t + \mathbf{W}_{in}\cdot[1;\mathbf{u}_t]) \tag{10.1}$$

$$\hat{\mathbf{y}}_{t+1} = \mathbf{W}_{out}\cdot[1;\mathbf{u}_t;\mathbf{x}_t] \tag{10.2}$$

where α is the leak rate parameter, f is a non-linear function, for which we use the hyperbolic tangent, $\mathbf{x}_t$ is the vector of internal reservoir states at time t, $\mathbf{u}_t$ is the vector of inputs at t (note that a bias term of 1 is appended to this vector). $\mathbf{W}$ is a matrix of internal reservoir weights, $\mathbf{W}_{in}$ is a matrix of input weights. The readout weights $\mathbf{W}_{out}$ are multiplied by the concatenation of the reservoir states, the input vector and the bias term to give the output $\hat{\mathbf{y}}_{t+1}$, which is a prediction of the next value of the time-series. A diagram of the implementation used is shown in Figure 10.1.

In order to train the output weights, ridge regression (regression with Tikhonov regularisation) is used:

$$\mathbf{W}_{out} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \tag{10.3}$$

where $\mathbf{X}$ is a matrix of collected reservoir states over the training period (the design matrix), λ is the regularisation constant and $\mathbf{y}$ is the target timeseries. The output weights matrix $\mathbf{W}_{out}$ has size $n \times d$ where d is the number of dimensions of the output data. In our case,

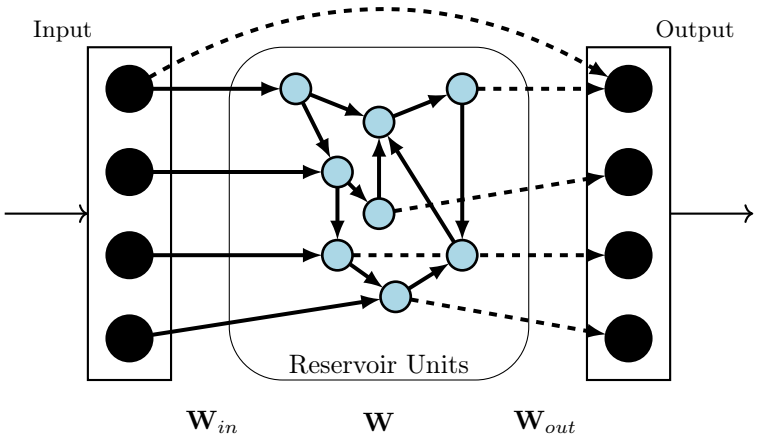


Figure 10.1: A typical ESN setup. The input is broadcast to all reservoir nodes using the $\mathbf{W}_{in}$ weights matrix, which is randomly intitialised. The internal reservoir weights $\mathbf{W}$ are also random and are not trained. The linear readout weights $\mathbf{W}_{out}$ which connect both the input units and the reservoir units to the output (dashed arrows) are trained using ridge regression.

both the input and the output data are 1-dimensional meaning that $\mathbf{W}_{in}$ and $\mathbf{W}_{out}$ have size $n \times 1$. Training a single readout layer via ridge regression is extremely fast and efficient compared to training the internal weights $\mathbf{W}$ using BPTT or other techniques.

In order for the ESN to work properly it must have the echo state property, effectively meaning it must have fading memory of inputs. This allows integration of information over time, but also guarantees that after some period the internal reservoir states will be entirely dependant on the input stream, and not the initial state of the system. A common heuristic for ensuring the echo state property is to scale the spectral radius of $\mathbf{W}$ (in other words the largest absolute eigenvalue) to be just below one (Yildiz et al., 2012). We scale the weights so that the spectral radius (ρ) is 0.9:

$$\mathbf{W} = \tilde{\mathbf{W}} \times \frac{0.9}{\rho(\tilde{\mathbf{W}})} \tag{10.4}$$

where $\tilde{\mathbf{W}}$ are the unscaled weights.

Although the ESN is only trained to predict the next value (i.e. one timestep in the future), multiple steps can be predicted by feeding this value back into the input successively. We refer to this as using the ESN in "free-running" mode. Obviously when this method is used to predict further into the future, errors will accumulate and the prediction will worsen. The degree of worsening depends on the Lyapunov exponent of the system being predicted. Chaotic systems have a positive Lyapunov exponent meaning that infinitesimally close trajectories will diverge exponentially.

10.1.3 Reservoir Quality

As outlined in the introduction to this chapter, different network topologies can have a significant impact on the RC performance (Jiang et al., 2008). Dale et al. (2019) characterises RC performance along three dimensions: memory capacity, kernel rank and generalisation rank. Memory capacity measures the networks ability to remember inputs from several timesteps in the past. Kernel rank is a measure of the reservoir's ability to distinguish between similar inputs (i.e. separability), while generalisation rank is a measure of its ability to generalise over different inputs and avoid over-fitting to noise (Legenstein & Maass, 2007).

There are some known associations between overall network properties and measures of reservoir capabilities: spectral radius influences the fading memory property, network sparsity influences kernel-rank. However, less is known about specific network structures and their applicability to specific tasks. Existing work in this area has included identifying network motifs, such as short loops, which can improve performance on certain tasks (Aceituno et al., 2020); there has also been work analysing specific network topologies such as ring, lattice and torus topologies (Dale et al., 2021), and connecting small sub-reservoirs (Dale, 2018; Qiao et al., 2017; Wringe et al., 2023). Whilst these sub-reservoir approaches can influence the large-scale structure of the network (eg. tightly connected sub-reservoirs with sparse interconnections), they do not directly tackle the question of the influence of network micro-structure.

10.1.4 Related Work

There has been much research on using evolutionary algorithms to find good reservoir topologies. Jiang et al. (2008) use the CMA-ES algorithm for evolving reservoir parameters for a motor control task. Chatzidimitriou and Mitkas (2010) adapt the Neuro-evolution of augmenting topologies (NEAT) approach introduced in Stanley and Miikkulainen (2002) to ESNs. They use mutation operators on matrices which include changing the density (by turning randomly selected weights "on" or "off"), changing the spectral radius by scaling the weights, and adding a new weight (expanding the matrix). By using the NEAT principle of historical markers, they are also able to implement a crossover operation which does not destroy evolved network structure. Whilst this process is somewhat similar to the approach used here in that the weights matrix can expand, it should be noted that they do not use a developmental approach, and evolution acts directly on the weights matrices.

Qiao et al. (2017) use a model of growing ESNs by iteratively expanding the weights matrix through the addition of blocks on the matrix diagonal, until the error is below some required threshold. In common with the approach used in this chapter, the readout weights are trained after each development step. However, these are not used to guide the development. Indeed the development is not a dynamical process as it does not depend on the present state of the system but simply involves the appending identically sized blocks

of random weights after each step. Dale (2018) also evolve hierarchical reservoirs, with the hyperparameters of each sub-reservoir and the interconnections between them adapted through evolution. Seoane (2019) take a more theoretical approach, exploring whether the RC paradigm is likely to have evolved in nature.

10.2 Experimental Setup

10.2.1 Mackey–Glass Oscillator

The Mackey–Glass delay differential equation was introduced in Mackey and Glass (1977) as a model of what the authors term "dynamical diseases". For example, the production of blood cells (haematopoiesis) is regulated by the current population of blood cells but there is a delay between the initiation of production of new cells in the bone marrow and their release into the blood stream. Such a system can be modelled by a delay differential equation. The paper showed that when the delay is relatively small the system enters a stable periodic attractor which is healthy behaviour. However if the delay becomes too large the system can become chaotic, which is pathological. Because of the ease of tuning the level of order or chaos in the system, it has become a popular benchmark for Reservoir Computing (Wringe et al., 2024). We use the version of the equation from (Glass & Mackey, 2010):

$$\frac{dx}{dt} = \beta \frac{x_{t-\tau}}{1 + x_{t-\tau}^n} - \gamma x_t, \quad \gamma, \beta, n > 0 \tag{10.5}$$

We use the following parameters which are commonly used in the Reservoir Computing literature (Goudarzi et al., 2015; Jaeger, 2001): $\beta = 0.2$, $\gamma = 0.1, n = 10$, $\tau = 17$. The system becomes increasingly chaotic as the time delay τ increases above 16.8. With $\tau = 17$ the system is therefore mildly chaotic. The time series data is generated using the Euler method with an initial value of $x_0 = 1.2$, and an integration timestep of 0.01, followed by downsampling so that the resulting series has a unit time interval (t). A sample of the series with these parameters is shown in Figure 10.2, along with a time embedding view which highlights the the chaotic nature of the attractor.

Following Jaeger (2001), before feeding the series to the ESN we recentre it by subtracting 1 and passing through the hyperbolic tangent function ($x \mapsto \tanh(x - 1)$)[1].

The ESN is trained on 2000 timesteps of the series (after an initial washout period of 100 steps). The trained network then has to generate the next 500 timesteps of the series. The output is fed into the input so that the ESN operates in "free-running" mode. We calculate the mean squared error (MSE) of the generated 500 steps compared with the actual next 500 steps of the series as:

[1]Note that other authors use different scalings: for example, Goudarzi et al. (2015) scales the Mackey–Glass series to the interval $[0, 0.5]$

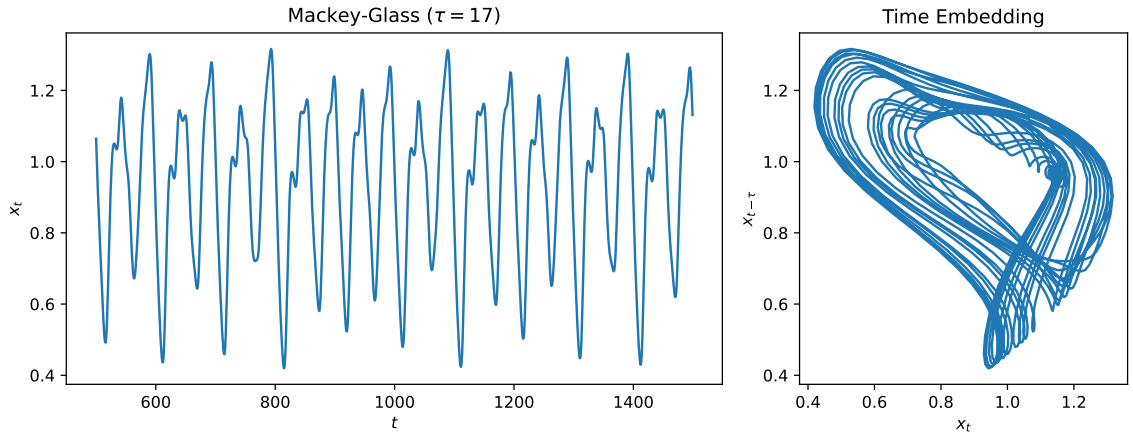


Figure 10.2: A sample of 1000 timesteps of the Mackey–Glass series with the parameters detailed above. The first 500 steps have been discarded. The right hand plot shows a time embedding view of the same data, with x_t plotted against $x_{t-\tau}$.

$$\mathrm{MSE}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{T}\sum_{i=1}^{T}(\hat{y}_i - y_i)^2 \tag{10.6}$$

where $\hat{\mathbf{y}}$ is the ESN-generated series and $\mathbf{y}$ is the actual series.

10.2.2 Conversion of DGCA graphs to ESNs

The graphs created by DGCA consist of vertices (V_g), directed edges (E_g) and a mapping from vertices to a finite set of labels which are referred to as "node states" ($\sigma_g : V_g \to \Sigma$). An ESN (or any neural network) uses *weighted* directed edges, and vertices without states (although during operation each node has an *activation*). To convert from a DGCA graph to an ESN, we therefore need a way to assign weights to the edges. One option would be to do this randomly. This would be a good approach if the aim was to construct and train an RNN. The edge weights would be adjusted during training, so a random assignment of edge weights (say from the uniform distribution $[-1, 1]$ would be a sensible starting point. However, in an ESN the edge weights are not trained, so the assignment of edge weights to the network may impact its performance. Conventionally, ESNs are constructed using random edge weights which are then scaled so that the weights matrix has a spectral radius less than one (see Section 10.1.2). However, for the evo-devo experiment conducted here it may be desirable for the ESN weights to be determined entirely by the genome, which is in this case the growth rule that generates the graph. Setting the weights deterministically allows the evolutionary process to be entirely responsible for performance, rather than having to contend with randomness. This can be achieved by using a fixed lookup table of weights based on the state of the starting node and ending node of each edge. Since a 3-state DGCA system is used, there are 9 [start,end] combinations, giving 9 distinct weights. These are

set to evenly span the interval $[-1, 1]$, as shown in Table 10.1. Because there are an even number of weights, this results in one option being set to zero (edges from nodes in state B to nodes in state B). Whilst this could be avoided by populating the lookup table with different values, it was decided to leave the possibility for an extant edge in the DGCA graph to be given a zero weight in the ESN. This means that certain edges in the graph produced by DGCA may effectively be "deleted" when it is converted to an ESN. This may be a useful ability for evolution to discover: an edge may have been useful during the developmental process but detrimental to the the ESN learning. Since the node states are a deterministic outcome of the DGCA growth process, evolution may be able to take advantage of this to improve the development and the learning.

		End		
		A	B	C
Start	A	−1	−0.75	−0.5
	B	−0.25	0	0.25
	C	0.5	0.75	1

Table 10.1: Lookup table for assigning weights to ESN $\mathbf{W}$ matrix based on the DGCA node state at the start and end of the edge.

As noted above, a common way to ensure that ESNs have the echo-state property is to scale the reservoir weights so that the spectral radius is less than one. Therefore after weights are assigned to the edges using the lookup table above, they are scaled to give a spectral radius of 0.9.

Most ESN implementations also use a randomly initialised input weights matrix ($\mathbf{W}_{in}$). However, we would also like these to be determined by the evo-devo process. Another lookup table is used for this: the input weight to each node is determined by that nodes state alone. For a 3-state system, the weights in Table 10.2 are used, which are again chosen to span the interval [-1,-1]. As before, one of these weights is 0 (input to nodes in state B). It is hoped that evolution may be able to take advantage of this: previous work has shown that a sparse input layer may improve ESN performance (Gallicchio, 2020).

State	Weight
A	−1
B	0
C	1

Table 10.2: Lookup table for assigning weights to ESN input layer $\mathbf{W}_{in}$ based on DGCA node states.

10.2.3 Development and Learning

A single graph generated by the DGCA system may be converted to an ESN and used for a prediction task as described above. As shown in previous chapters the growth rule (MLP

weights) of the DGCA can be evolved to produce graphs with particular properties. An obvious evolutionary approach to using DGCA for growing a high-performing ESN reservoir would simply be to use a population of growth rules to grow DGCA graphs for a fixed number of timesteps or until an attractor was reached. These graphs could then be assessed as ESNs and the fitness function of the evolutionary search would be their accuracy (low MSE) on the prediction task.

However, this procedure does not take full advantage of the developmental nature of DGCA. Because DGCA grows graphs over multiple timesteps there is an opportunity to provide feedback *during* the growth process. As noted in Section 10.1.1, this set-up is analogous to some types of biological development, in which environmental or internal feedback can shape the growth process.

In order to implement this we use the following procedure. After each developmental step, the graph is converted into ESN by applying weights to the edges as described above. The ESN is then trained on the Mackey–Glass time series prediction task. The training produces a vector of readout weights ($\mathbf{W}_{out}$). On the next development step, this vector is appended to the matrix of node neighbourhood information, effectively meaning that each node's developmental decisions are biased by its entry in $\mathbf{W}_{out}$. The MSE of the ESN in the prediction task is also appended but this time as a global bias which affects all nodes. Since values in $\mathbf{W}_{out}$ can be very large (> 100) and we do not wish this input to dominate the node state information, we scale $\mathbf{W}_{out}$ by dividing by its maximum absolute value (infinity norm). This ensures all entries are in the range $[-1, 1]$ before they are appended to the node information matrix.

Equation 6.18 introduced the node neighbourhood information aggregation function:

$$\mathbf{C} = \text{Agg}(\mathbf{A}, \mathbf{S}) \tag{10.7}$$

where $\mathbf{A}$ is the graph adjacency matrix, $\mathbf{S}$ is the node state matrix, with one row for each node, and Agg is the aggregation function (see Equation 6.15 for a concrete implementation). The output of this $\mathbf{C}$ is used as the input for both the state update and the action update MLPs (Equations 6.22 & 6.23), with each row representing the input for one node. In the present implementation, $\mathbf{W}_{out}$ is horizontally concatenated to $\mathbf{C}$. The MSE value is repeated n times to form column vector (denoted $\text{MSE} \cdot \mathbf{1}_n$) which is also concatenated:

$$\tilde{\mathbf{C}} = [\mathbf{C}, \mathbf{W}_{out}, \text{MSE} \cdot \mathbf{1}_n] \tag{10.8}$$

$\tilde{\mathbf{C}}$ is then used as the input to the state and action MLPs. This causes the graph's next development step to be influenced not only by the node states (selected from a finite discrete set Σ - in the present implementation a 3-state system is used), but also by the continuous variables of the per-node values of $\mathbf{W}_{out}$ and the global value of the task MSE. After the

development step, the same procedure is repeated, with the graph being used as an ESN on the same task, giving a new $\mathbf{W}_{out}$ and MSE. This process can be continued for a fixed number of developmental timesteps, or until the error goes below a certain threshold. In the experiment presented here, we run 50 steps of development, but terminate early if the graph grows above 500 nodes. This is in order to cut off the runaway growth behaviour identified in Chapter 4. The process is illustrated in Figure 10.3.

The intuition behind this approach is that each node's entry in $\mathbf{W}_{out}$ gives a sense of the importance of that node in the prediction of the ESN. Of course, there may be nodes which perform an important function inside the ESN, but nevertheless have a low readout weight, so it is a mistake to think of the readout weights as a form of "credit assignment" indicating the contribution of each node. However, it is hoped that by providing this information to the growth process, it may be possible for evolution to discover growth rules which can steer development in an advantageous direction. In the extreme, evolution could select MLP weights which totally ignore this extra information, meaning the system would behave just like the "closed" versions of DGCA which have been discussed up to this point, relying only on the node states to determine the developmental trajectory. At the other extreme, MLP weights could be selected such that *only* the node readout information was used, making development entirely dependant on the network's "experience" of use as an ESN. This highlights the benefit of also feeding the MLPs the MSE as a global bias term: it may allow evolution to select an approach which modulates between these two extremes based on the current level of the error.

For example, a possible strategy might be to use a more "experience-blind" approach to growth in the very early stages, when the network is small and therefore the error is likely to be high. Then when the network is medium sized (so the error is likely to be lower), more weight could be put on the node readout weights in guiding development. Finally when the error has become very small this might be interpreted by the growth rule as a signal that development is "complete" and no further changes should be made. There are parallels with the different ways "experience" is used to guide biological growth at different stages: the waves of electrical activity which help the organisation and patterning of the retinal cells occur only during one phase of development. [2]

We refer to the procedure here as "integrating feedback from learning during development" or simply *learning feedback*.

10.2.4 Timescales

Note that there are three different timescales in operation during this experiment:

1. The evolutionary timescale, indicated by the generation number.

[2]The present work does not conduct an investigation into the different possible *strategies* for evolution to harness the supplied information to tune the development process. The above discussion is simply to motivate the inclusion of the per-node $\mathbf{W}_{out}$ entries and the global MSE error.

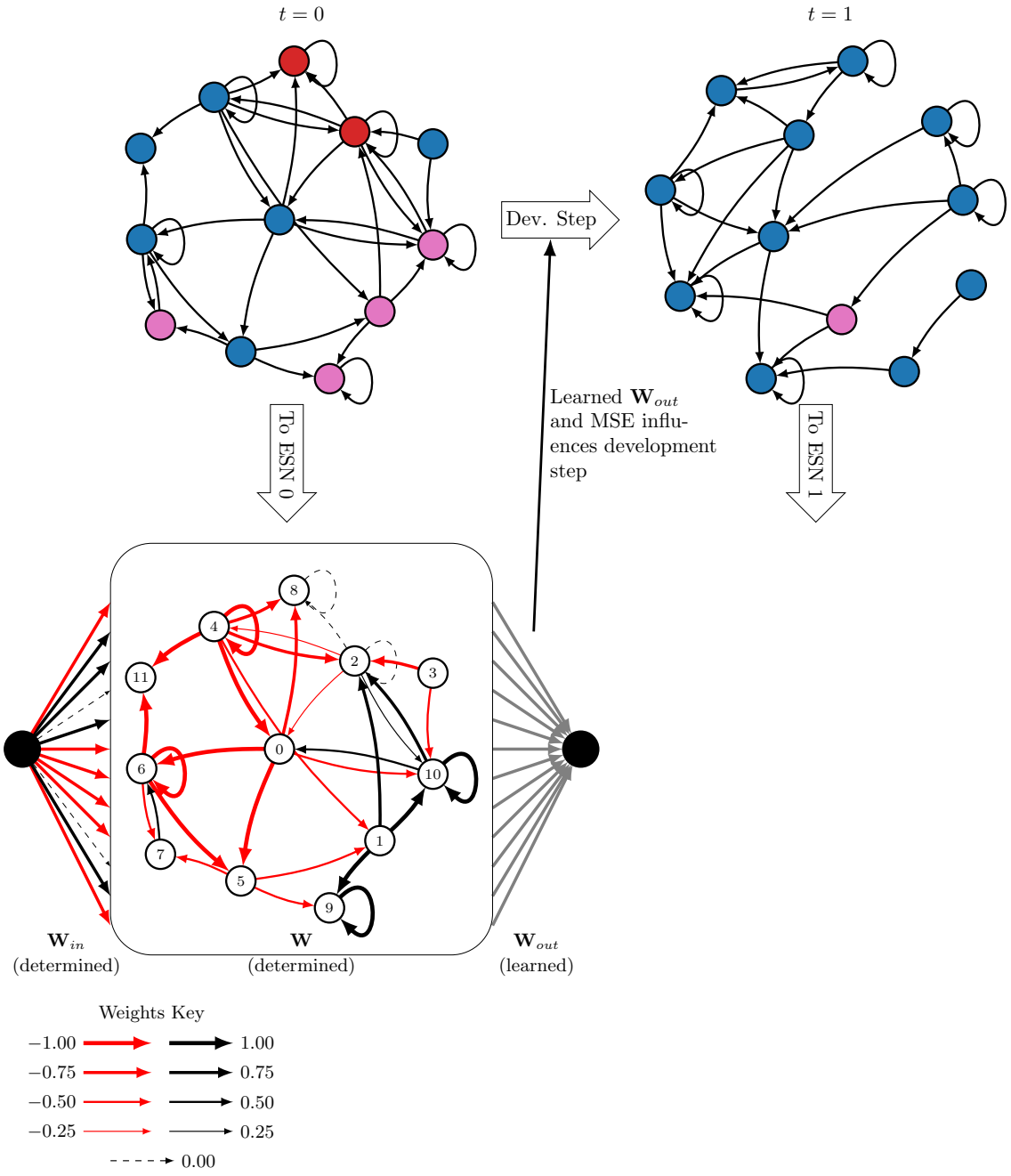


Figure 10.3: Development and learning with DGCA & ESN. The seed graph (DGCA state at $t = 0$) is shown at the top left. This is converted into an ESN by using the lookup Table 10.1 for the internal weights ($\mathbf{W}$) and Table 10.2 for the input weights ($\mathbf{W}_{in}$). $\mathbf{W}$ and $\mathbf{W}_{in}$ are therefore entirely determined from the DGCA graph state. (The correspondence between node colours and states is: blue=A, red=B, pink=C). The readout weights $\mathbf{W}_{out}$ are learned based on the task. The trained ESN is then used to predict the Mackey–Glass time series, and the error (MSE) is calculated. Both $\mathbf{W}_{out}$ (one value per node) and the MSE (same value at all nodes) are appended to each node's neighbourhood information vector, biasing the next DGCA development step (shown at the top of the diagram). The resultant DGCA graph-state (at $t = 1$) is then converted into a new ESN (not shown) and the process continues. The seed graph and DGCA update rule used here are randomly selected and for illustrative purposes only: it is not claimed that the development step shown improves ESN performance.

2. The developmental timescale, indicating each step of the developmental process.

3. The operational timescale meaning the timesteps of the Mackey–Glass series.

These correspond to phylogenetic, ontogenetic and epigenetic timescales. The timescales are strictly nested. Each evolutionary timestep gives rise to a new generation of individuals (growth rules). Each individual must then be developed for a number of timesteps. Each developmental timestep includes a full run of operational timesteps.

10.2.5 Seed graph selection

As noted in previous chapters, the developmental outcome is dependant both on the growth rule (the MLP weights) and the starting point: the seed graph. In order to evolve a growth rule which fulfils particular requirements, it is therefore beneficial to use the same seed graph throughout the evolutionary run. One growth rule may lead to good outcomes if applied to a particular seed graph but poor outcomes if applied to another; this could make evolution of a good growth rule very difficult. In previous chapters, a simple and small seed graph has been used: a ring of 8 nodes with DGCA v1 (Chapters 4 & 5) and a single node with DGCA v2 in subsequent chapters.

In the present experiment, the aim is to convert the graph into an ESN and train the readout weights after every developmental step, as described above. However, very small networks are unable to complete the task at all and it is not possible to train readout weights or calculate a global error. It was therefore decided to use a relatively large seed graph so that right from the first developmental step, $\mathbf{W}_{out}$ and the MSE could be calculated and fed back into the growth MLPs.

In order to choose this seed graph, a parameter sweep over random Erdős–Rényi graphs was conducted. The parameters of the Erdős–Rényi model are n the number of nodes and p the edge probability (also referred to as connection density). The parameter sweep was conducted over values of $n \in \{25, 50, 75, 100, 125, 150, 175, 200, 225, 250\}$ and values of $p \in \{0.01, 0.015, 0.02, 0.025, 0.03, 0.035, 0.04, 0.045, 0.05, 0.055\}$. For each parameter setting, 300 random graphs were constructed and used as ESNs on the Mackey–Glass prediction task. The median MSE over these trials is shown in the upper left plot of Figure 10.4. Note that not all trials converged so in some cases it was not possible to calculate the MSE (or it went to infinity). The median MSE is taken over those trials which did converge. The percentage of trials which converged is shown in the upper right plot of Figure 10.4. As can be seen in these two plots very small ($n < 100$) and very sparse ($p < 0.02$) networks perform poorly (larger MSEs, fewer trials converging). Note that (Goudarzi et al., 2015) uses a reservoir of 500 units for the Mackey–Glass prediction task: it is to be expected that using a much smaller reservoir gives poor results. These upper two plots report the results for graphs which are converted into ESNs by the random assignment of weights to edges. The weights are selected from the interval $(-1, 1)$. After the weights are assigned,

the spectral radius of **W** is rescaled to 0.9. However, as described in Section 10.2.2, when we convert a DGCA graph to an ESN, discrete weights are selected from the lookup table shown in Table 10.1. If a 3-state DGCA system is used, as in this experiment, this results in $\mathbf{W} \in \{-1, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1\}^{(n \times n)}$. After assignment the weights are also scaled to give a spectral radius of 0.9. The restriction of the reservoir weights to these discrete values reduces the performance of the ESN. The same results as before are shown in the lower two plots of Figure 10.4. These are the relevant results when considering what size and connection density of seed graph should be used. The actual data for this lower left plot are shown in Table 10.3.

		Connection Density (p)									
		0.5%	1.0%	1.5%	2.0%	2.5%	3.0%	3.5%	4.0%	4.5%	5.0%
Num Nodes (n)	25	85.5	51.6	60.9	69.4	63.9	68.6	70.4	59.1	56.5	55.3
	50	66.3	46.3	48.9	42.5	30.4	23.3	21.6	17.8	22.7	23.9
	75	56.2	42.4	26.4	16.5	17.5	18.7	16.8	19.2	18.9	19.5
	100	56.1	23.7	17.7	16.2	18.1	18.3	18.3	23.5	21.2	22.8
	125	44.4	16.3	17.0	15.5	17.1	17.1	23.7	23.6	24.8	24.0
	150	25.5	16.1	15.2	14.3	12.1	18.9	28.4	29.7	31.5	30.3
	175	23.1	15.7	12.9	13.1	14.7	25.7	28.4	34.0	33.2	28.9
	200	15.8	13.1	12.2	15.5	17.4	28.3	35.2	38.3	31.3	26.8
	225	17.8	11.8	13.5	18.2	26.5	36.0	39.9	35.2	30.3	23.4
	250	14.4	10.7	14.4	16.6	26.1	41.0	39.0	33.2	27.8	20.9

Table 10.3: MSE$\times 10^3$ for Mackey–Glass prediction task with different network initialisations and discrete weights (median MSE over 300 trials for each setting). This is the data displayed in the lower left plot of Fig. 10.4. The yellow cell indicates the chosen parameter values.

The aim is to select a seed graph that is as small as possible, but still converges when used as an ESN on the prediction task. It does not matter if the MSE is high, as it will be the task of the evolution and development process to improve this. As a result, the parameters $n = 50$ and $p = 0.025$ were chosen, close to the frontier of performant networks (highlighted with a red square in Figure 10.4). Out of the 300 Erdős–Rényi graphs constructed with these parameters, the one which gave a MSE closest to the median was selected. This graph and its corresponding ESN weights matrix is shown in Figure 10.5a. Its performance on the Mackey–Glass prediction task is shown in Figure 10.6; the MSE is 0.03. The generated signal for the first 200 timesteps is reasonably accurate but then it rapidly deteriorates.

10.2.6 Evolutionary goals

As in previous chapters, the "genomes" in this evolutionary experiment are the MLP weights which control the DGCA growth. In the present implementation this growth is not a closed deterministic process but can be influenced by the ESN learning experience as described in Section 10.2.3.

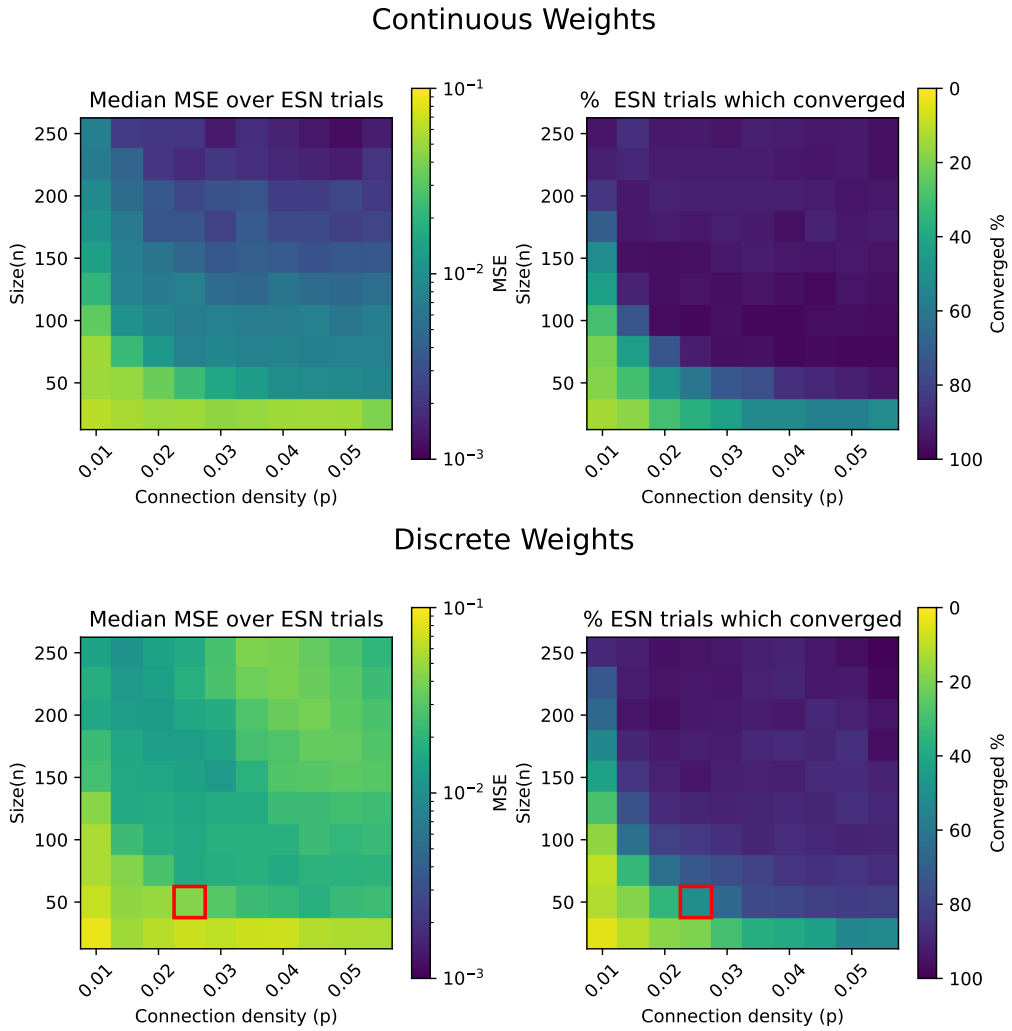
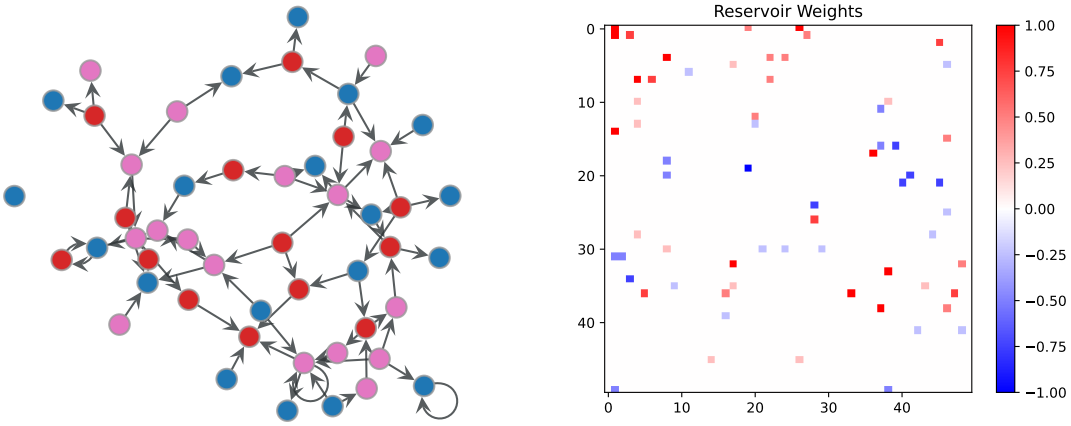


Figure 10.4: 100 random Erdős-–Rényi graphs were constructed for each size (n) and edge density (p). These were then used as ESNs for the Mackey–Glass generation task. The left plots show the median MSE over the 300 trials. The right plots show the percentage of trials which converged (with very small/sparse networks it was not possible to train the readout). The upper plots shows networks using weights in the continuous range $[-1, 1]$; the lower plots show those using the discrete weights shown in Table 10.1 for the reservoir weights and those in Table 10.2 for the input weights. The chosen values for $n = 50$ and $p = 0.025$ in the seed graph are highlighted with a red square. The median MSE for these values with the discrete weights was 0.030 over the 66% of the trials which converged. The actual data for the lower left plot (MSE with discrete weights) are shown in Table 10.3.



(a) Chosen seed graph (b) Corresponding $\mathbf{W}$

Figure 10.5: The chosen seed graph. The node colours in (a) indicate the 3 different node states. These are used to choose the 9 discrete edge weights shown in (b). Where there is no edge, the corresponding value in $\mathbf{W}$ is zero. Note that $\mathbf{W}$ is rescaled to give a spectral radius of 0.9 before being used as an ESN reservoir.

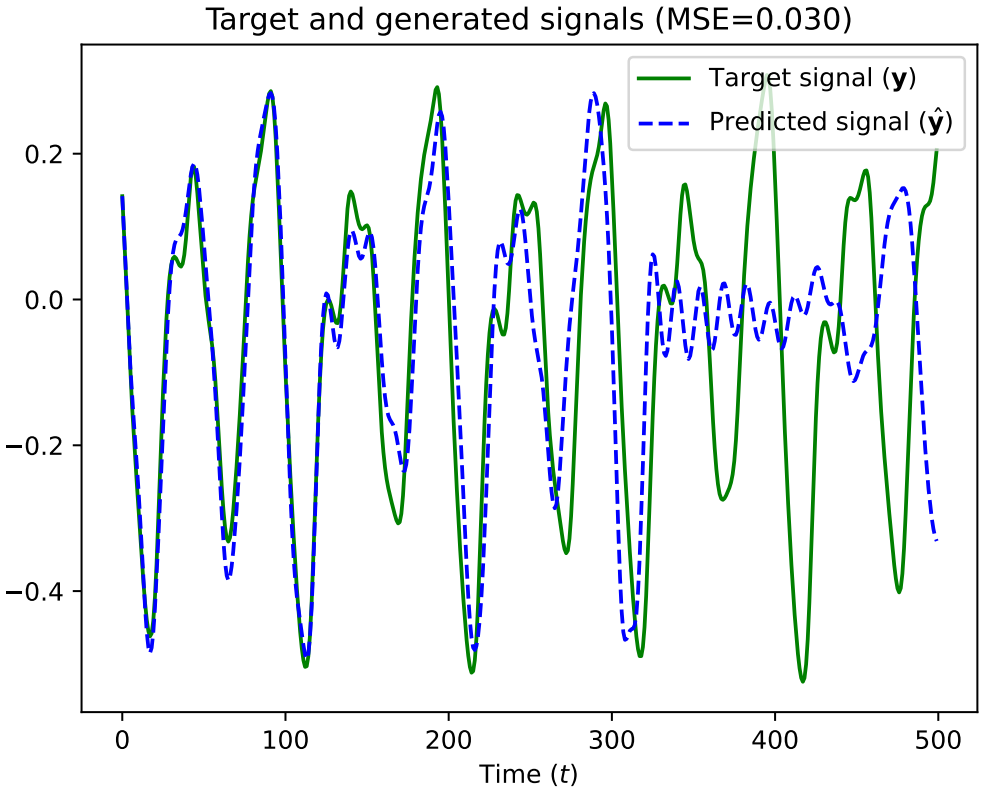


Figure 10.6: The seed graph shown in Fig. 10.5 is converted into an ESN and is trained to predict the Mackey–Glass series. As expected, the performance is poor. It is the task of the evo-devo process to improve this performance.

The obvious evolutionary goal is to reduce the MSE of the networks on the Mackey–Glass prediction task. However, it is relatively easy to reduce this by using much larger networks. As shown in Figure 10.4, using larger (random) networks improves performance. It would therefore be desirable to minimise the MSE whilst also minimising network size. This implies a multi-objective optimisation (MOO). As in Chapter 9 we use the Non-Dominated Sort Genetic Algorithm (NSGA2) of Deb et al. (2002) to implement this MOO. However, since this is an evo-devo process, each genome gives rise not to a single graph/ESN which gives a single MSE, but to a succession of graphs over the course of the developmental process. We must therefore decide which of these graph's MSE and size should be used as the scores for that genome. The obvious option would be to run each genome's developmental process for a fixed number of timesteps and then use the MSE and size of the final graph. However, preliminary experiments revealed that performance can vary unpredictably over the course of development, sometimes improving before deteriorating again, and sometimes improving in an oscillatory fashion: the error trending downwards but with spikes of poor performance along the way. This is to be expected in a developmental process. Not every developmental step will monotonically improve the organism: there may be intermediate steps which are necessary for restructuring but temporarily impair performance. Measuring performance after a fixed number of developmental timesteps therefore seems arbitrary. Instead it was decided to measure the *minimum* MSE generated by an ESN over the course of the developmental process. The size of the network is also measured at this point. The number of development steps taken to reach this minimum was also recorded, but is not a target for optimisation. The nested evo-devo and learning process described here is shown in Algorithm 3.

10.3 Experimental Results

The results of an evolutionary run using 50 individuals over 250 generations are shown in Figure 10.7. For each genome the minimum MSE achieved during development is reported (y-axis) as well as the network size at that point (marker size). The number of development steps taken to reach this minimum MSE is represented by the marker colour (note that this is not something which is optimised in the evolutionary process). For clarity, only individuals in the Pareto front of minimising network size and MSE for each generation are shown. Individuals are only shown in the first generation in which they appear, since Pareto optimal individuals often persist for multiple generations in NSGA2. In the first few generations, the error is reduced somewhat by using larger graphs ($\sim$ 100 nodes). To reduce the error further larger graphs ($\sim$ 300 nodes) are used by about generation 50. However there is a selection pressure against larger graphs in the MOO. By around generation 200, individuals are discovered which produce very low errors ($\sim 10^{-4}$) with relatively small graphs ($\sim$ 200 nodes). One such optimal individual is circled in green and is examined in more detail. This

Algorithm 3 Evolutionary-Developmental-Learning Process

```
Initialize population of DGCA rules P
Initialise seed graph g
for generation in E do                                          ▷ Evolution loop
    for p in P do
        for development step in D do                            ▷ Development loop
            g ← DGCA(g, p, W_out, MSE)                          ▷ Run one DGCA development step
            Convert DGCA to ESN
            Train ESN on Mackey–Glass series                    ▷ Calculate W_out
            Use trained ESN to predict further Mackey–Glass steps
            Calculate MSE of prediction
        end for
        Record best MSE over the course of development with growth rule p
        Record graph size at the point of best MSE.
    end for
    Create offspring population
    P ← Pareto optimal individuals (small MSE, small graph size) from parent and
offspring population.                                           ▷ NSGA2 algorithm
end for
```

individual was discovered in generation 187, produced a minimum MSE of 5.97×10^{-5}, and at that point had 207 nodes. This individual was selected for discussion rather than the point to the right of it, which has an only marginally smaller MSE (5.78×10^{-5}) and a larger number of nodes (275).

The development of the circled individual is detailed in Figure 10.8. For each developmental step the MSE of that graph used as an ESN on the Mackey–Glass prediction task is shown. The MSE trends downwards as the graph develops. The best MSE is achieved at 34 development steps (the point circled in green). The actual graph at that point is shown in Figure 10.9a, along with the weights matrix after it has been transformed into an ESN in Figure 10.10 (note this is before the weights are rescaled to give a spectral radius of 0.9). The network has a noticeably modular structure, with 5 tightly connected clusters, and fewer connections between the clusters. One of the clusters is not connected to the main component at all (along with a smaller disconnected group of four nodes). It seems doubtful that these detached components improve the ESN's prediction: they are effectively separate small ESNs which are being run in parallel to the main one. However, their readout weights are not set to zero, as shown in Figure 10.9b, suggesting they are making some useful contribution. As noted in Section 10.1.3, several studies have used modular structure (via various sub-reservoir architectures) to improve performance.

The actual performance of this graph when used as an ESN on the Mackey–Glass prediciton task is shown in Figure 10.11. It almost exactly matches the target Mackey–Glass signal for the 500 timesteps it generates. Figure 10.11 may be compared with prediction of the seed graph ESN in Figure 10.6 to see the improvement.

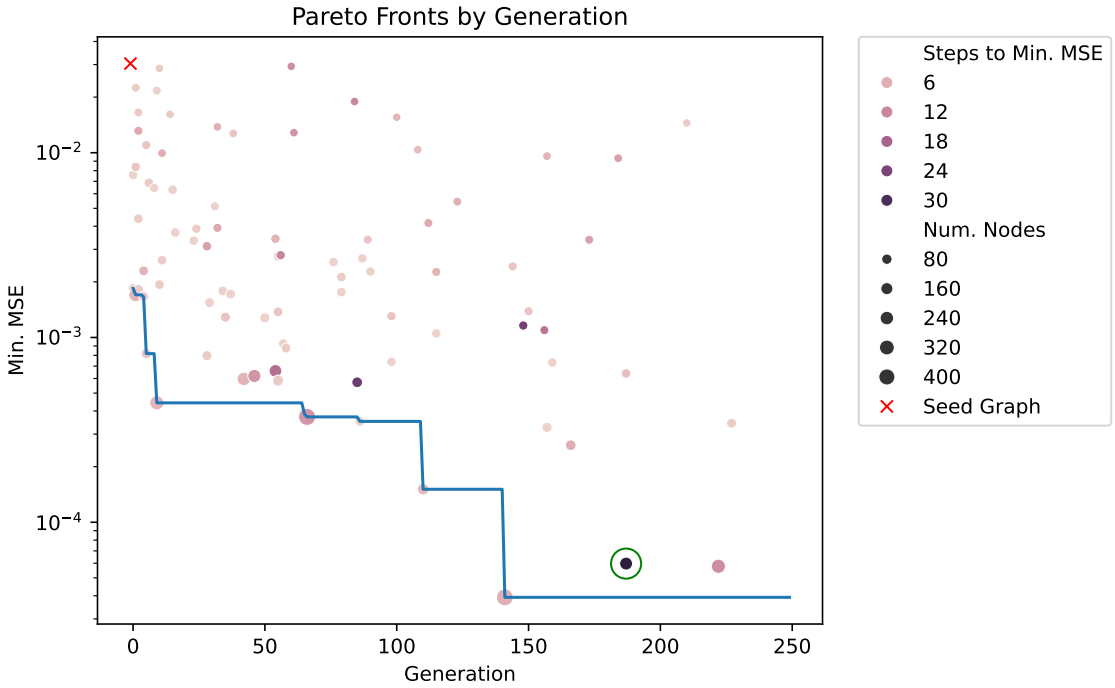


Figure 10.7: The points indicate Pareto optimal individuals in the first generation in which they appear. Point colour indicates the developmental steps taken by the individual to reach its minimum MSE, while point size indicated the size of the network at this minimum MSE point. The blue line indicates the minimum MSE discovered so far in evolution. The MSE of the seed graph is marked with a red cross in the upper left. One of the final optimal individuals which is chosen for further investigation is circled in green. Whilst this individual does not have the lowest MSE, it is on the Pareto front as it has a smaller number of nodes. The large number of developmental steps taken to reach its minimum MSE also makes it more interesting for review.

Note that the three different timescales referred to in Section 10.2.4 are shown on the different charts. The evolutionary timescale is shown on the x-axis of Figure 10.7. The developmental timescale is shown on the x-axis of Figure 10.8. The operational timescale is shown on the x-axis of Figure 10.11.

10.3.1 Comparison with Random Networks

One criticism of these results might be that the developmental process improves the performance of the ESN simply by growing the size of the graph. This is implied in Figure 10.8, where the performance improvement appears to be correlated with the growing size of the graph in the second plot. It is well known that ESNs with more nodes can perform better on a given task (as shown in the parameter sweep in Figure 10.4). This would suggest that all the growth rule needs to do is add more nodes, which is very simple behaviour. As shown in the random search in Chapter 4, the runaway growth behaviour is very easily produced by

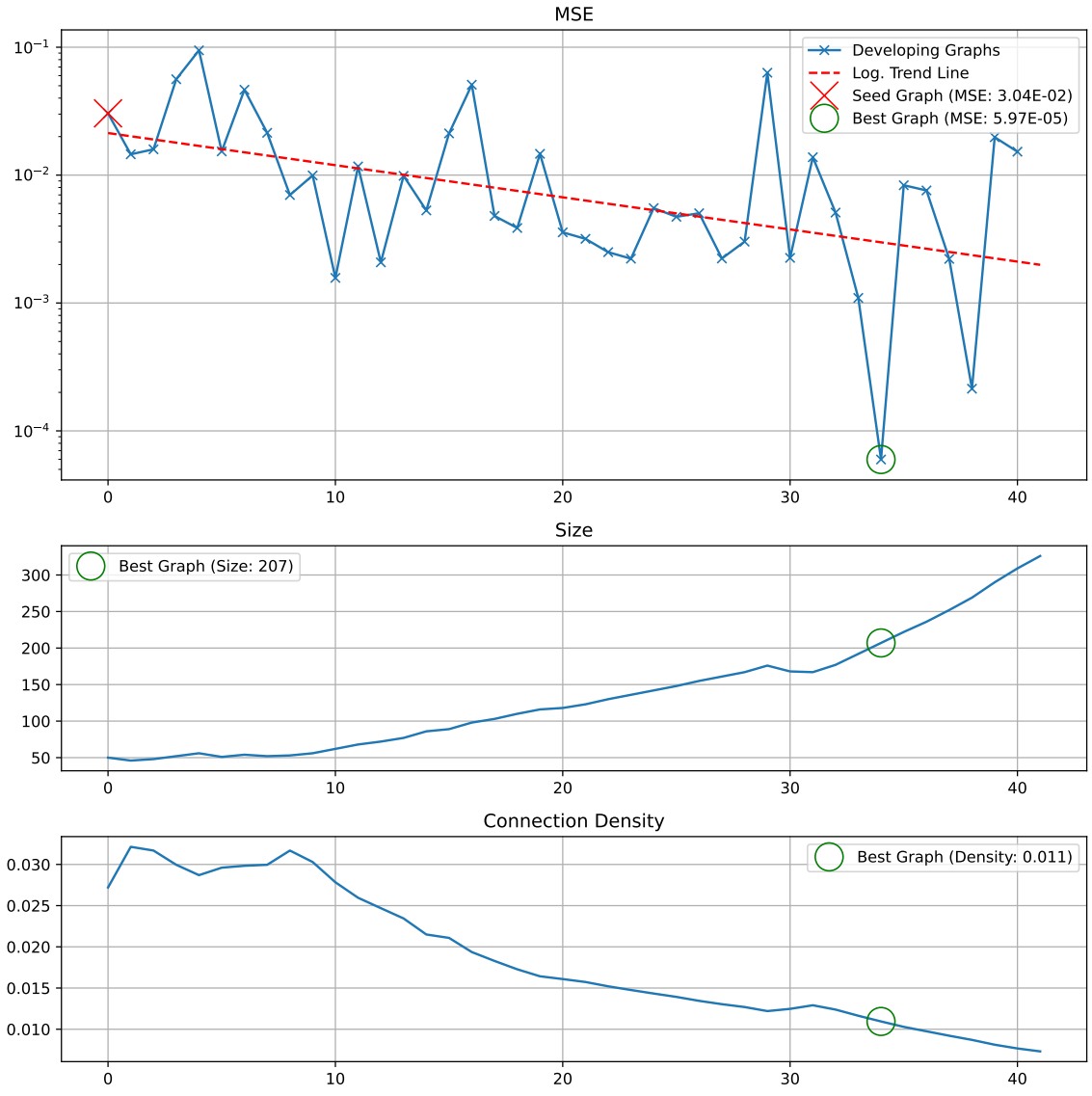
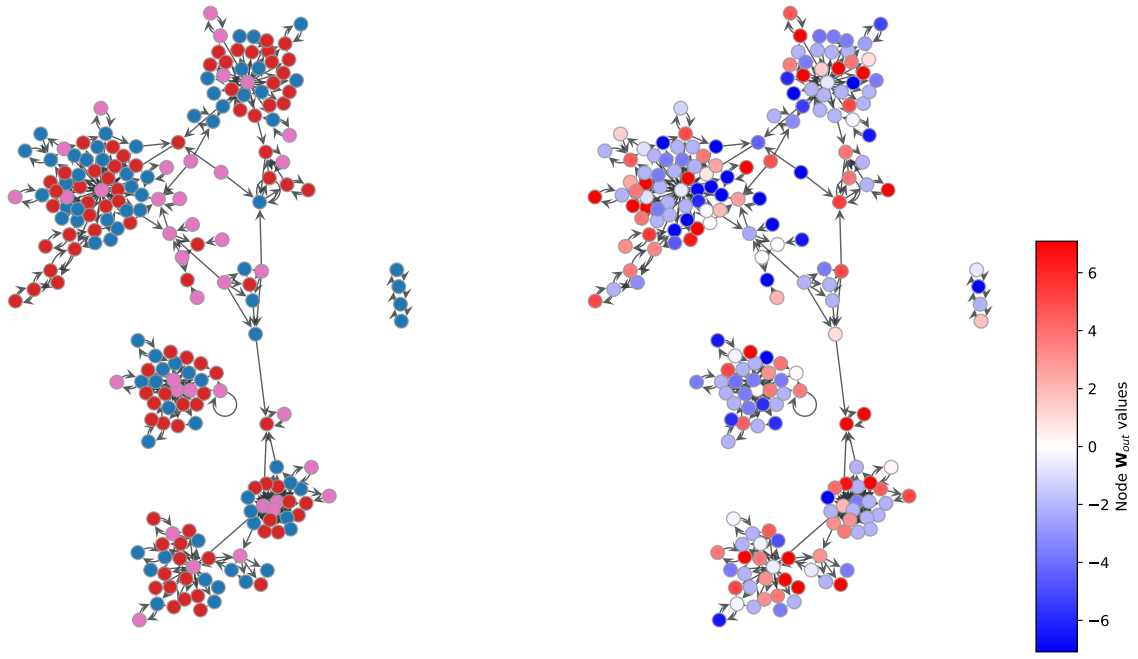


Figure 10.8: Development of the highlighted individual in Fig. 10.7



(a) Nodes coloured by state

(b) Nodes coloured by readout weight

Figure 10.9: The actual graph at the 34th development timestep. The nodes are coloured by state (red, blue and pink for the three different states) in (a), whilst in (b) they are coloured by readout weight.

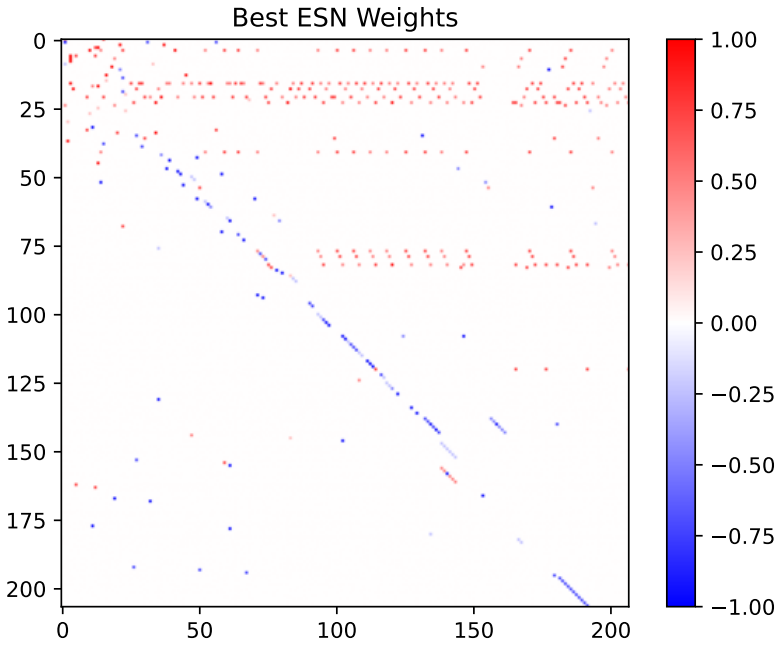


Figure 10.10: The edge weights of the graph shown in Fig. 10.9 when it is transformed into an ESN (ie. $\mathbf{W}$). Note that only the 9 discrete weights shown on the colour bar ticks are used (from Table 10.1).

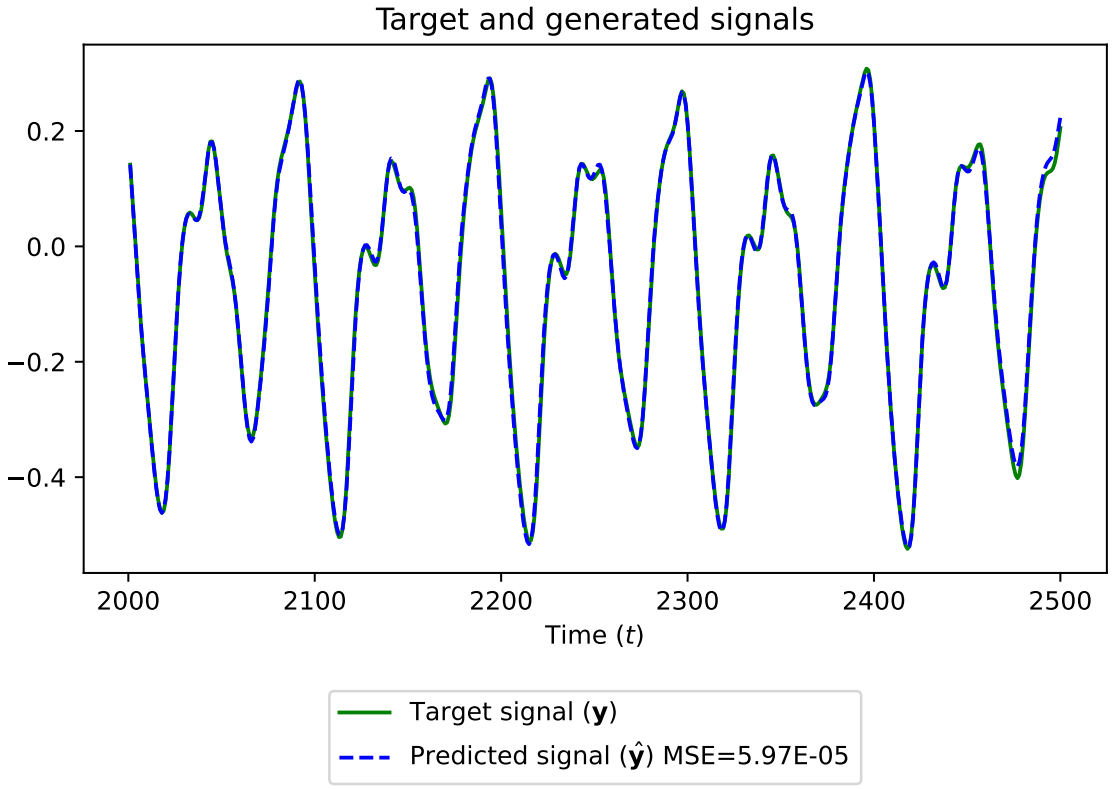


Figure 10.11: The target signal for timesteps 2000-2500 of the Mackey–Glass series (after the washout and training periods), and the free-running prediction of the developed ESN whose structure and weights are shown in Figs 10.9 & 10.10. The MSE of the generated signal is 5.97×10^{-5}.

the DGCA system.[3] It would therefore seem unnecessary to perform an evolutionary search to discover this behaviour, undermining the case for an evo-devo process. Furthermore, if most of the increase in performance over the course of development is simply due to the addition of more nodes, it suggests that there may not be anything special about the specific structure created by the developmental process. Chapter 7 showed that DGCA is capable of producing specific graph microstructure, and the motivation for the work in this chapter is that the developmental process might networks with structure which is particularly suited to a given ESN task. If the development process is improving the task performance *merely* by growing bigger networks, this would be a less interesting result.

The experiment was set up to try and avoid this, by having one of the objectives of the MOO as minimising graph size. Development was also terminated early if the graph grew above 500 nodes. Furthermore, the developed graph *does* seem to have an unusual modular structure as a result of the developmental process (Figure 10.9) and does not have the appearance of a randomly constructed graph. However, to test this criticism more quantita-

[3] Although DGCA v1 was used in Chapter 4, whereas DGCA v2 is used here, it is also true that DGCA v2 frequently produces runaway growth.

tively, 300 random Erdős–Rényi graphs with the same size (n) and edge probability (p) as the best developed graph were constructed. They were used as ESNs on the same Mackey–Glass prediction task. To make this a fair comparison, the random ESNs were assigned edge weights ($\mathbf{W}$) and input weights ($\mathbf{W}_{in}$) from the discrete sets in Tables 10.1 & 10.2 respectively (with $\mathbf{W}$ then rescaled to give a spectral radius of 0.9). The distribution of MSEs of these random ESNs is shown in the right hand box-plot of Figure 10.12. As shown the developed network (with an MSE of 5.97×10^{-5}) is an order of magnitude better than the best of the random graphs (MSE of 2.35×10^{-4}), and several orders better than the median of the random graphs (MSE of 1.48×10^{-2}). The left hand box-plot shows the distribution of randomised network MSEs with the same n and p values as the seed graph. The seed graph itself has an MSE of 0.03, close to the median of this distribution - it was deliberately selected for this reason as explained in Section 10.2.5. These results show that the best developed network did not only give a low MSE in absolute terms, but also relative to random graphs with the same size and density, indicating that the specific structure created by the developmental process was indeed valuable.

10.3.2 Importance of Feedback During Development

In order to test the importance of the feedback from the learning experience during development (ie. the fact that the DGCA MLPs were supplied with $\mathbf{W}_{out}$ and MSE values of the ESN after each developmental step), we run the same growth rule (highlighted in Figure 10.7) without this feedback. Since we are using the same growth rule, the MLPs have input layers which are sized to receive the $\mathbf{W}_{out}$ and MSE values concatenated to the node neighbourhood information vectors (as shown in Equation 10.8). In order to supply the correct sized input vectors, we simply replace the $\mathbf{W}_{out}$ and MSE values with vectors of zeros:

$$\tilde{\mathbf{C}} = [\mathbf{C}, \mathbf{0}_n, \mathbf{0}_n] \tag{10.9}$$

The results of running the developmental process in this way, without feedback from the learning experience at each step, are shown in Figure 10.13. These charts can be compared with those in Figure 10.8. As shown, without the learning feedback the MSE does not trend downwards, and the best MSE is much higher than when feedback was used. This confirms that in this growth rule found by the evolution the feedback from learning did indeed play an important role.

It is also interesting that growth was much faster, and the graph breached the 500 node size threshold after only 27 development steps. This suggests that in the case of this particular growth rule, the feedback from learning played an inhibitory role with respect to graph growth. Of course, in other cases evolution may find many different ways to use the feedback from learning to guide development, as discussed in Section 10.2.3.

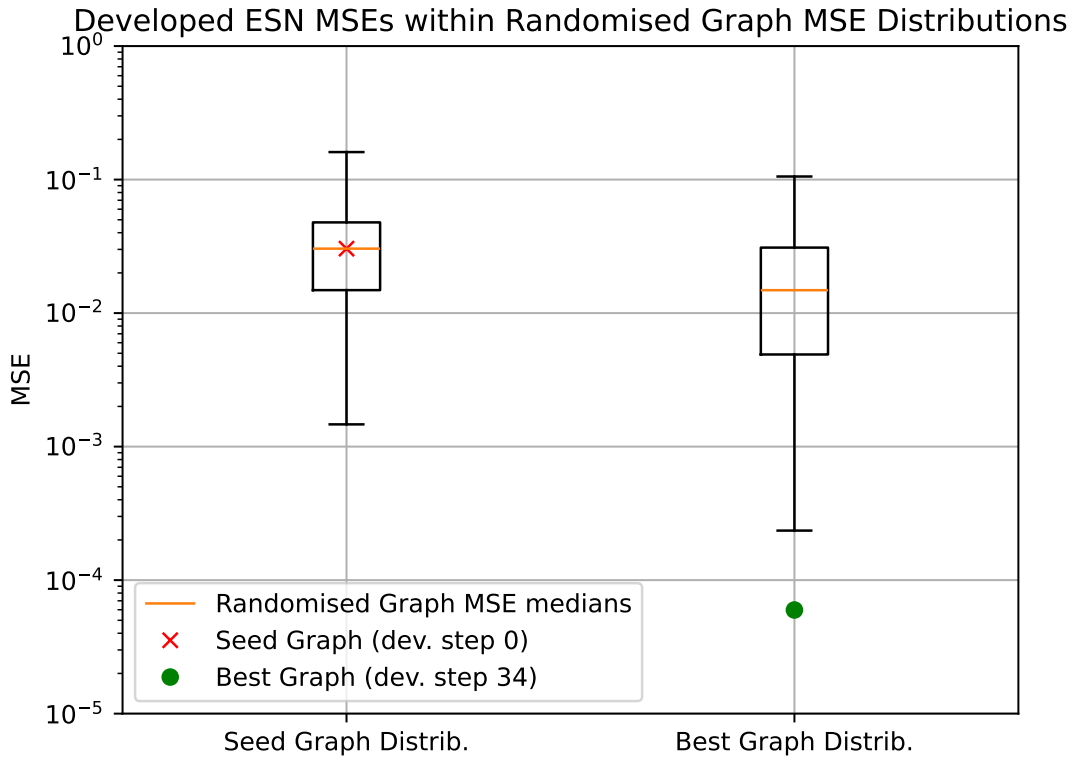


Figure 10.12: The MSE distributions of 300 random Erdős–Rényi graphs with the same number of nodes (n) and edge density (p) as the seed graph and best developed graph. The boxplots show the minimum, lower quartile, median, upper quartile and maximum MSE (MSEs >100 were discarded before these statistics were calculated to remove ESNs which did not converge). The seed graph's actual MSE is shown as a red cross: as intended, it has an MSE close to the median of the ensemble of random graphs. The MSE of the best developed graph is shown as a green circle. It is lower than any MSE generated by the 300 random graphs with the same n and p values, indicating that the specific structure created by the developmental process was useful.

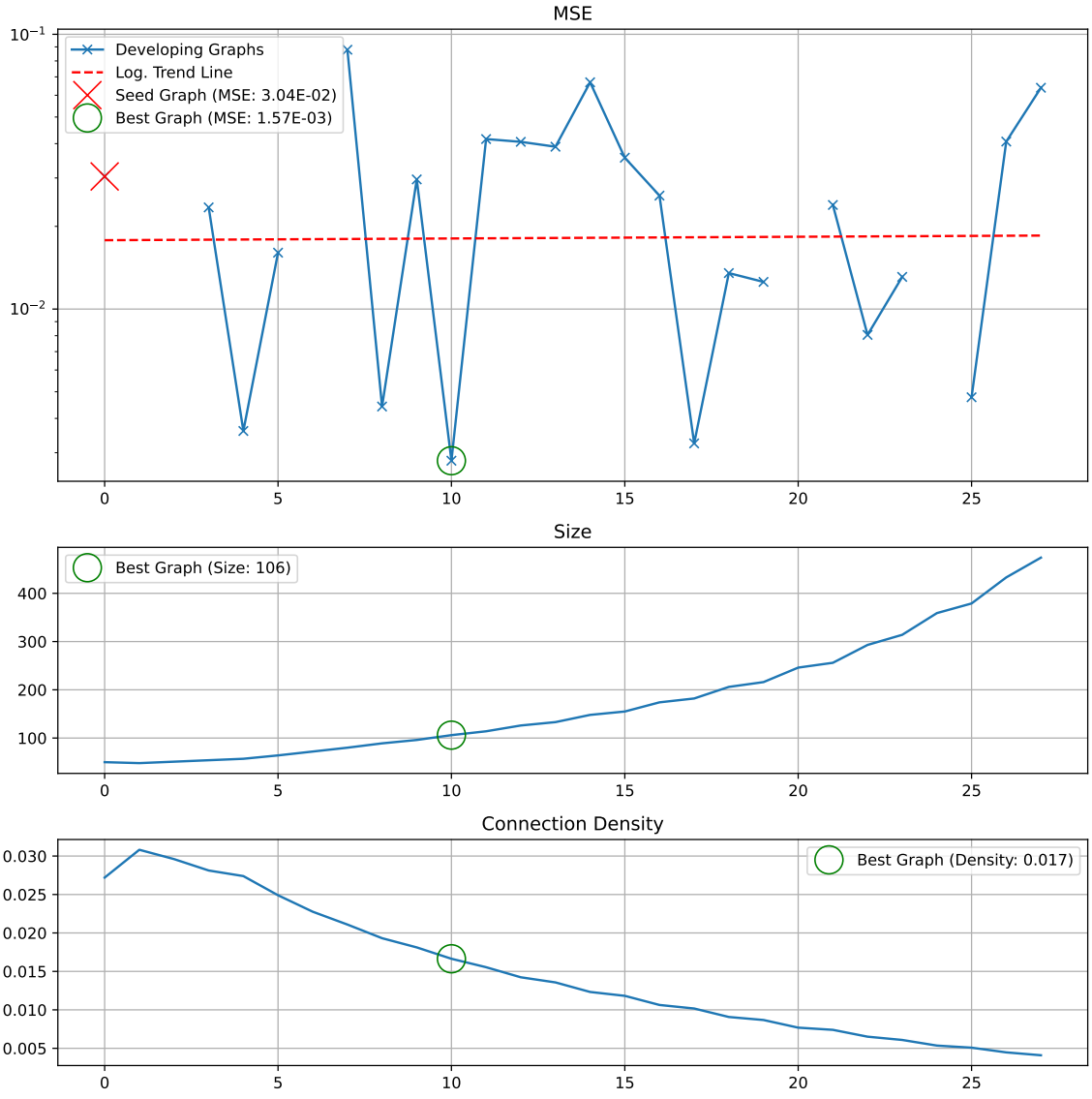


Figure 10.13: The same data as shown in Fig. 10.8 for the same growth rule but this time not providing feedback about the $\mathbf{W}_{out}$ and MSE values after each developmental step (the growth rule is fed zeros instead of these values).

10.4 Conclusion and Future Work

The experiment shown in this chapter is preliminary work showing in principle how DGCA can be used with an evo-devo process to create high quality reservoirs for RC. More work needs to be done to investigate the usefulness of biasing the development process with the learned readout weights an error. On a simple task like predicting the Mackey–Glass series, it is likely that evolution could find a growth rule which generated a good reservoir structure *without* this feedback from the learning process. The feedback during development might be more relevant if a single growth rule was being used to grow reservoirs for two different tasks. The feedback (trained output weights and error) from each task might be different, influencing the development to take different courses in each case. Finding a growth rule (genome) which could respond appropriately to different environmental cues (feedback from the learning experience) would similar to meta-learning. Strictly speaking, it would not be "learning to learn" but "learning to develop to learn". Certain benchmark tasks in the RC literature such as the multiple superimposed oscillators task can be parametrised differently to create a family of tasks with similar properties (Wringe et al., 2024). It would be interesting to investigate whether a single growth rule could be found which could produce high performing ESNs for different versions of the task, by growing differently in response to the feedback from learning.

Chapter 11

Conclusion

11.1 Summary of Findings

This thesis introduces an extension of Graph CA, in which nodes can be added and removed and the graph structure can develop as part of the same dynamical process which updates the node states. Various design choices are made, guided by making the model as simple and expressive as possible. The resulting system exhibits rich behaviour, both in terms of the growth dynamics it produces, and in terms of the wide variety of graph structures it can produce. This behaviour is evolvable: the "genome" is represented by the weights of the small internal neural network. Despite the nonlinear mapping between these weights and the outcome (in terms of dynamical behaviour and graph structure), it does seem possible to evolve the weights to produce the desired outcome.

Chapter 4 introduces DGCA v1 and identifies different classes of dynamical behaviour, with the most interesting being the halting growth class, in which development naturally comes to a halt as the system enters an attractor. Chapter 5 conducts a random search and a novelty search over the dimensions of transient length, attractor cycle length and maximum size reached. Novelty search is able to push the dynamical behaviour into more extreme areas, showing the evolvability of the system. Several example growth behaviours are then examined, highlighting the richness and variety of the dynamics produced by the system.

Chapter 6 introduces DGCA v2, an improved version of the system giving more flexibility in the way newly created nodes are connected to their local neighbourhood. Amongst other things, this allows the replication of whole subgraphs (with or without modification), when groups connected nodes which independently decide to divide. Chapter 7 puts DGCA v2 to the test by evaluating the graph microstructure it can produce using an analysis of motifs. An evolutionary experiment is conducted to evolve growth rules which produce graphs with the same motif distributions as certain biological networks from different families. This shows that the graph structures produced by DGCA are evolvable.

Chapter 8 investigates a property discovered in Chapter 5, that disconnected graph com-

ponents may have independent dynamics, each entering separate attractors. This chapter introduces a new formalism for describing this behaviour, and shows some examples, highlighting that the graph state transition diagram can itself be a complex graph at a higher level of organisation. Comparisons are drawn with symbolic grammars: if we consider each graph state to be a symbol, then the graph state transition diagram can be regarded as a symbolic grammar which emerges from sub-symbolic dynamics. Chapter 9 uses the new formalism in an analogy of cell division and differentiation. The phenomenon of graphs splitting is analogised as cell division, and the development trajectories of graph components are analogised as cell differentiation. An evolutionary experiment is conducted showing that a single growth rule can produce graph components with very different microstructure.

Chapter 10 conducts a preliminary investigation into the use of DGCA to generate computational substrates under the reservoir computing paradigm. Since DGCA is a developmental system, it makes sense to integrate the learning and development processes. Node-level feedback from the learning experience is used to bias the development steps. The results of this evo-devo-learning experiment are encouraging and point towards computational use cases for the DGCA system.

11.2 Contributions

The research questions set out in Chapter 3 are recapitulated here:

1. Can we implement a version of Graph CA with dynamical structure?
2. Can we evolve the dynamics of development *and* the graph structures which result?
3. Can the developmental dynamics produce analogues of self-reproduction and other biological phenomena?
4. How do the growth dynamics impact the structures that are produced? Can multiple different structures be produced by a single growth rule?
5. Can we use an evo-devo process to produce a useful computational substrate?

As outlined above, the thesis investigates all of these questions. Chapters 4 introduces a minimal working model of Graph CA with dynamical structure, whilst Chapter 6 refines the model and describes it more formally.

Several of the chapters address the question of evolvability, with Chapter 5 focussing on the evolvability of the growth dynamics using an open-ended novelty search, and Chapter 7 using goal-directed evolution to generate graphs with particular microstructure. Chapter 9 combines the two, using a multi-objective evolutionary process to maximise the number of component attractor cycles and the degree of structural difference between individuals. In doing so this chapter also addresses question 4, on the interaction between the growth

dynamics and the graph structures, showing that a single growth rule can indeed produce multiple different graph structures.

Question 3 on biological analogies is addressed in various places, starting in Chapter 4, where the halting growth behaviour is highlighted as an analogue of naturally terminating biological growth. Chapter 8 also discusses how the graph division behaviour can be viewed as a type of self-reproduction. Other biological metaphors are also explored, such as the cell division and differentiation analogy in Chapter 9.

The final question is addressed in Chapter 10. Chapter 7 shows that DGCA can be used to produce graphs with varying microstructure, evaluated using an analysis of network motifs. It is hypothesised that network motifs in biological networks may have been selected and conserved by evolution due to their computational properties (implementing memory, conditional feedback etc). If this is the case then the structural expressivity of DGCA could make it a useful algorithm for generating graphs with particular computational characteristics. Chapter 10 conducts preliminary work in investigating this, showing that the graphs generated by DGCA do show promising computational abilities when used as ESNs.

11.3 Discussion

Some difficulties were encountered in devising a simple yet expressive formalism for graph development based on CA. The first version of DGCA, used in Chapters 4 & 5 uses a single SLP to choose whether to change a node state or divide or remove the node. This has the unintended consequence that in systems with more node states, structural modification options were less likely to be chosen. This in turn means that such systems are more likely to enter an attractor. Furthermore the expressivity of the structural modifications is limited by the fact that duplicate nodes always duplicate their parents' edges. The improved version of the system introduced in Chapter 6, aims to remedy these problems: firstly by using separate SLPs to perform the node state update and the node action (division, removal or no action); and secondly by specifying more fully which edges a duplicated node should have, out of a set of 'allowable' local edges. It was shown that version 2 of DGCA can in theory produce *any* graph structure from a single-node seed graph given a correctly parametrised growth rule. Whilst the description of DGCA v2 is somewhat complex, the maths behind the state update and structural update process remains relatively simple. In particular, using Kronecker tensor products to implement the structural update makes this step concise and easy to program using standard linear algebra libraries (for example using the `numpy` Python library).

Using a matrix representation of the graph throughout the development process (both the adjacency matrix and the node state matrix) makes it very fast to run on modern computer hardware. It can be accelerated further by the use of GPUs, which are well suited to linear algebra. This speed is noteworthy, since many graph grammar approaches to graph

development are very computationally intensive and slow. Matching the left-hand side of a graph grammar rule involves solving the subgraph isomorphism problem, which is NP-complete. As previously discussed, graph grammars are also often nondeterministic. By using a CA-inspired approach, in which decisions are made at the node level based on local neighbourhood information, the subgraph matching problem is sidestepped, allowing speed and determinacy.

Other motivations for using a CA-based model to implement graph development are detailed in Section 1.1. They include the biological similarity of an approach in which information is shared locally between cells each following the same "program". The complexity which can emerge from simple update rules and local communication, which is well-established in CA research, is again confirmed here. This thesis focuses on evaluating the expressiveness of the DGCA model both in terms of the structures and the dynamics it gives rise to. As demonstrated, the DGCA model is highly expressive in both respects, and is also amenable to evolution.

11.4 Future Work

11.4.1 Robustness and Perturbation

It was noted in Chapter 3 that dynamical systems can exhibit a range of sensitivities to perturbation. In some cases, the system is robust to perturbation, quickly recovering its original trajectory through state space, whilst in other cases it is highly sensitive, and can even be pushed into different basins of attraction. It would be interesting to investigate both these cases with respect to the dynamical development of DGCA. Graph states can be perturbed by changing one or more node states, or adding or removing edges. On the one hand, an experiment could be conducted to evolve growth rules which were robust to some level of perturbation. On the other hand, it would be interesting to find growth rules where development could be switched between two attractors by a small perturbation. This latter example could have applications in growing networks which are sensitive to environmental feedback.

11.4.2 Larger Scale Structure

Building on the work presented in Chapter 7, it would be interesting to assess the ability of DGCA to create networks with larger network motifs ($k > 3$) and larger-scale modular structure. In particular, the ability of the system to reproduce network structures whilst also introducing variation suggests it could present an interesting analogue of structures in biological networks. Both metabolic networks (Ravasz et al., 2002), and neural networks (Bullmore & Sporns, 2009) exhibit modular structure. More generically, the body plan of many organisms exhibits repeated structure with variation, for example in the fingers of a

hand (Gould, 2002).

Measuring the presence of "modules", or subgraphs with similar but not necessarily identical structure, may require the application of novel graph theoretic techniques. Whilst there are several algorithms for detecting communities in networks (eg. Newman (2004)), it is more difficult to detect repeated structure. Exact subgraph matching is a classic problem in graph-theory. In recent years, several approaches have emerged for detecting approximate subgraph matches (Boury et al., 2023; Liu et al., 2019). By using these methods it may be possible to analyse DGCA's ability to create graphs with repeated but varying structure which is typical in biology.

11.4.3 Differentiable Development

The experiments reported in this thesis have used evolution to adapt the MLP weights to produce desired behaviour. Given the highly nonlinear mapping between the MLP weights (genotype) and the graph structures produced by the developmental process (phenotype), evolution seems like the most appropriate method. However, it would be desirable to be able to invert this mapping. In other words, *given* a final desired graph structure, what MLP weights would produce the right developmental steps to produce it from a given seed graph? This problem seems intractable as there may be multiple different developmental paths from a particular seed graph to a particular final graph. Furthermore, each possible path may involve a different number of steps.

On the other hand, this problem is somewhat analogous to training a neural network via backpropagation. The graph structure updates in DGCA are implemented using Kronecker tensor products and the output of MLPs (see Equation 6.40). MLPs can be trained using backpropagation. Tensor products are also differentiable, suggesting it may be possible to calculate a gradient back through all development steps. As with training a RNN using backpropagation-through-time, this would require "unfolding" all developmental steps, and calculating a gradient through each of them. This may mean that given a final graph structure, the whole development process could be differentiated, allowing the calculation of the MLP weights which would produce the necessary developmental steps to get there.[1]

This is a speculative suggestion and may not be possible with the current mathematical representation of DGCA. In particular, whilst the addition of nodes is implemented using tensor products, their removal (Equation 6.42) is not, meaning this part of the graph structure update is not currently differentiable.

If it were possible to differentiate the development of a specific graph structure, it could allow DGCA to be used as a compression algorithm, allowing a specific large graph to be represented by a very small growth rule.

[1]This would be somewhat similar to the gradient based Neural Development Programs introduced in Najarro et al. (2023), although those are constrained to develop feedforward networks.

11.4.4 Endogenous Evolution

In Chapter 9, the graph splitting into multiple disconnected components was analogised as cell division, with each component then pursuing its own developmental trajectory in an analogy of cell differentiation. In nature, cell division can also be the occasion of genetic mutation. It would be interesting to investigate the effects of mutating the MLP weights used by each disconnected component on graph division events. In other words, when a small graph component is split off the main body of the graph, that component's growth rule would be slightly altered. This would allow a form of endogenously driven evolution, in which growth rules gradually change over time as a consequence of reproduction. Rules which are more successful at reproduction via graph division would end up as a larger proportion of the population, and would also have more chances to mutate and improve further. This would be somewhat similar to the "Evoloop" extension of Langton's loops, in which a self-reproducing (grid) CA pattern can reproduce itself and evolve over time (Sayama, 1999), except that the individuals would consist of free-floating graphs rather than patterns of cells on a grid.[2]

11.4.5 Computational Substrates

The work shown in Chapter 10 on using graphs developed with DGCA as ESNs gave preliminary indications that this could be an interesting line of research. In particular, using the learning experience of the ESN (as indicated by they readout weights) to influence further DGCA development appeared promising. There are many further experiments which could be conducted in this framework, including testing whether the same genome can be used to grow reservoirs for different tasks given different "environmental feedback" from the learning experience. It would also be interesting to connect this work with the experiments on motifs in Chapter 7 to evaluate whether graphs with particular microstructure (measured by motif significance profile) gave better performance on particular tasks. Investigation on the links between graph structure and RC metrics such as memory capacity, kernel rank and generalisation rank would also be interesting, but this is a broader topic which moves away from DGCA research.

[2]Note however that in Evoloop, as in the Universal Constructor of von Neumann (1967), the genome exists as a "tape" of cell states *within* the CA. The CA rule is then analogised as the laws of physics or chemistry which "execute" this genome. In DGCA, we have analogised the parametrisation of the update rule itself as the genome.

Appendix A

Appendix to Chapter 4

A.1 Growth trajectories with random seed graphs

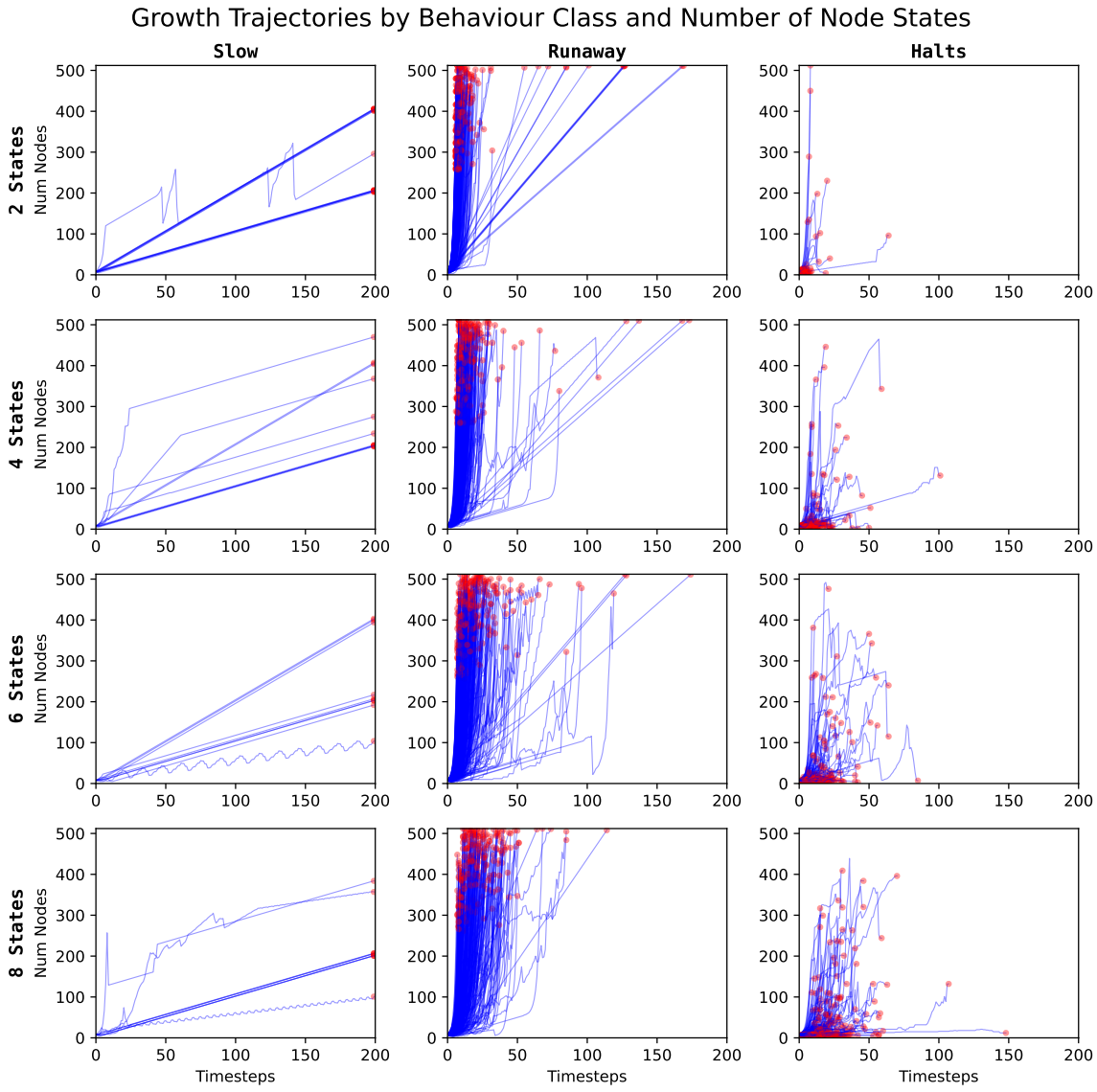


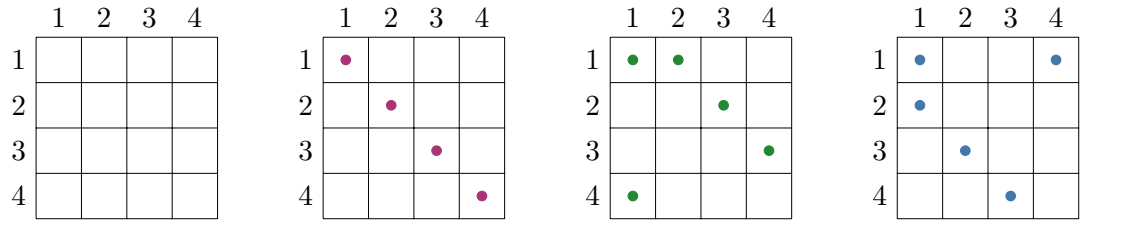
Figure A.1: Growth trajectories using random seed graphs, for 2, 4, 6 & 8 state systems, divided into three behaviour classes. See the caption to Fig. 4.7 for explanation.

Appendix B

Appendix to Chapter 6

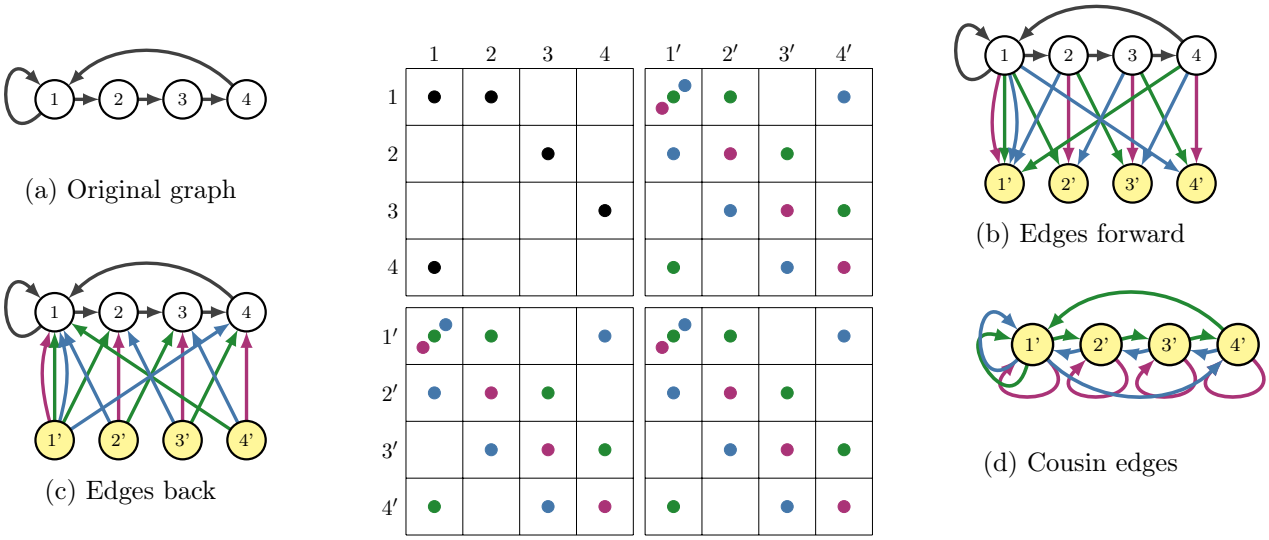
B.1 Further diagrams for the structural update

This section provides further diagrams explaining the structural update step in DGCA, using the same example graph as in Section 6.5.1.



(a) The empty matrix $\mathbf{0}$ (b) The identity matrix $\mathbf{I}$ (c) The original adjacency matrix $\mathbf{A}$ (d) The transpose of the original adjacency matrix $\mathbf{T} = \mathbf{A}^T$

Figure B.1: An alternative view of Equation 6.44. The four choices of matrix which can be used in each quadrant. Squares containing dots indicate ones in the adjacency matrix; squares without dots indicate zeros.



(a) Original graph

(b) Edges forward

(c) Edges back

(d) Cousin edges

Figure B.2: All possible edges that could be created if all nodes of the original graph divided (the intermediate $\mathbf{A}^+$ matrix). The upper left quadrant of the adjacency matrix shows the original graph (a). The upper right quadrant shows the edges forwards from the original nodes to the new nodes (b). In this group each new node can choose to have edges directly from its parent (red), from its parents incomers (green) or from its parents outgoers (blue). The lower left quadrant shows the edges back from the new nodes to the original nodes (c), where again each new node can have edges from one of the four matrices. The lower right quadrant shows edges between new nodes (d).

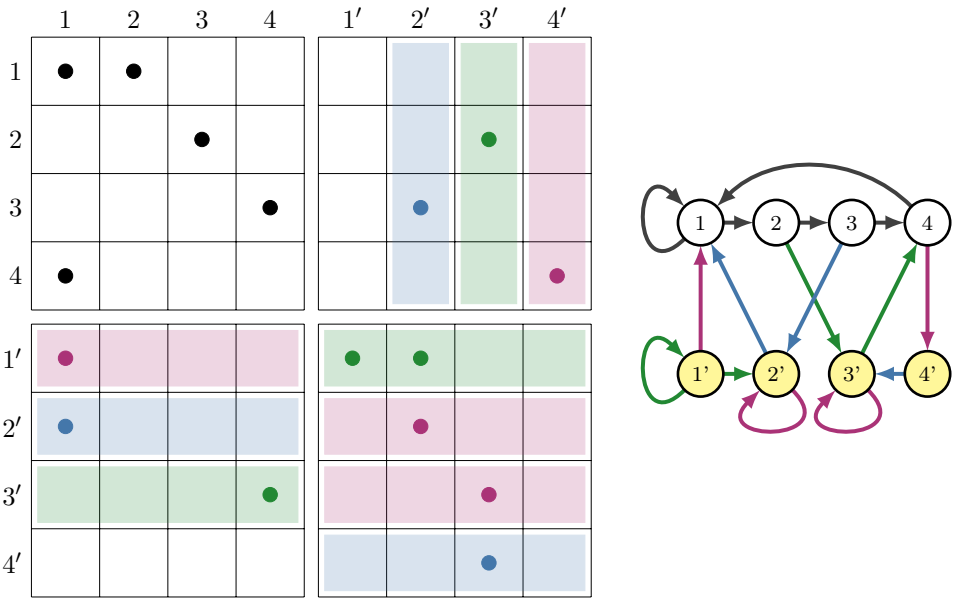


Figure B.3: An alternative view of Equation 6.50. Of all the possibilites in Fig. B.2, this shows one possible choice of edges to be instatiated to give $\mathbf{A}^+$. Note that nodes $2'$ and $3'$ have self loops because they have chosen rows from the $\mathbf{I}$ matrix (shown in red), but node $1'$ has a selfloop because it has chosen its row from the $\mathbf{A}$ matrix, which also gives it the edge $1' \rightarrow 2'$ (shown in green). The next step is the deletion of nodes 3, $3'$ and $4'$ in converting $\mathbf{A}^+$ to $\mathbf{A}^-$ (shown in Equation 6.54).

B.2 Minimal Python Implementation

A minimal python implementation of one graph update step using DGCA v2, as described in Chapter 6 is shown below.

```
import numpy as np
rng = np.random.default_rng(42)

# Set hyperparameters
num_states = 3
mlp_state_layer_sizes = [3*num_states, num_states]
mlp_action_layer_sizes = [3*num_states, 15]

# Set parameters (weights and biases)
wgts_S = [rng.uniform(low=-1, high=1, size=(i,o)) for \
          i,o in zip(mlp_state_layer_sizes[:-1],mlp_state_layer_sizes[1:])]
biases_S = [rng.uniform(low=-1, high=1, size=(1,s)) for \
              s in mlp_state_layer_sizes[1:]]
wgts_A = [rng.uniform(low=-1, high=1, size=(i,o)) for \
          i,o in zip(mlp_action_layer_sizes[:-1],mlp_action_layer_sizes[1:])]
biases_A = [rng.uniform(low=-1, high=1, size=(1,s)) for \
              s in mlp_action_layer_sizes[1:]]

# With these hyperparameters, the model has 180 parameters.
tot_params = np.sum([w.size for w in wgts_S]) \
    + np.sum([b.size for b in biases_S]) \
    + np.sum([w.size for w in wgts_A]) \
    + np.sum([b.size for b in biases_A])
print(f'The model has {tot_params} parameters')

# Helper functions
def run_mlp(in_vec, wgts, biases):
    h = in_vec
    for w,b in zip(wgts, biases):
        h = np.tanh((h @ w) + b)
    return h

def one_hot(matrix):
    out = matrix == np.max(matrix, axis=1, keepdims=True)
    return out.astype(int)
```

```
# Setup seed graph
A = np.array([
    [1,1,0,0],
    [0,0,1,0],
    [0,0,0,1],
    [1,0,0,0]
])
S = np.array([
    [1,0,0],
    [0,1,0],
    [0,0,1],
    [1,0,0]
])

# Calculate neighbourhood information
C = np.hstack([S, A.T @ S, A @ S])
# Normalise
C = C / np.clip(np.max(np.abs(C), axis=1, keepdims=True), 1, np.inf)

# Run MLPs
Y_S = run_mlp(C, wgts_S, biases_S)
Y_A = run_mlp(C, wgts_A, biases_A)

S_plus = np.vstack([one_hot(Y_S), S])
action = one_hot(Y_A[:,:3])
remove, noaction, divide = action.T
keep = np.hstack([np.logical_not(remove), divide]).astype(bool)

F = one_hot(Y_A[:,3:7])
B = one_hot(Y_A[:,7:11])
N = one_hot(Y_A[:,11:])

Qm = np.array([[1,0],[0,0]])
Qf = np.array([[0,1],[0,0]])
Qb = np.array([[0,0],[1,0]])
Qn = np.array([[0,0],[0,1]])

A_plus = np.kron(Qm, A) \
```

```
        + np.kron(Qf,
              np.diag(F[:,1]) +
              A @ np.diag(F[:,2]) +
              A.T @ np.diag(F[:,3])) \
        + np.kron(Qb,
              np.diag(B[:,1]) +
              A @ np.diag(B[:,2]) +
              A.T @ np.diag(B[:,3])) \
        + np.kron(Qn,
              np.diag(N[:,1]) +
              A @ np.diag(N[:,2]) +
              A.T @ np.diag(N[:,3]))

# The new A and S matrices defining the graph after one development step.
S_minus = S_plus[keep]
A_minus = A_plus[keep,:][:,keep]
```

Appendix C

Appendix to Chapter 7

C.1 Subgraph counting with the RAND-ESU algorithm

The figures in this appendix give a fuller account of the RAND-ESU subgraph counting algorithm used in calculating the triad significance profile (TSP), and parameter sweep which was used to choose the RAND-ESU sampling rate, as outlined in Section 7.2.3. The algorithm was introduced in Wernicke (2006).

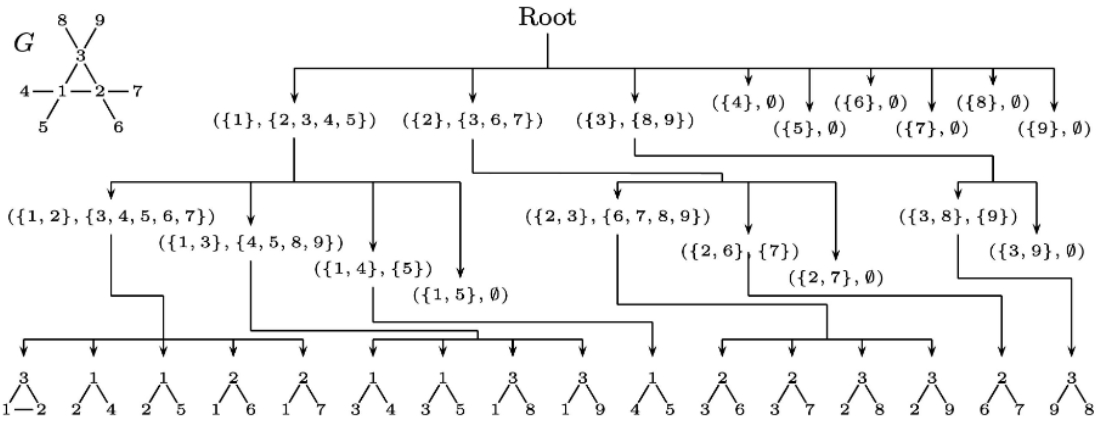


Figure C.1: Figure reproduced from Wernicke (2006) Fig. 4, showing the ESU (enumerate subgraphs) tree for the graph shown in the upper left. The tree illustrates the process of enumerating all connected three-node subgraphs by iterating through connected nodes. For example $\{\{1\}, \{2, 3, 4, 5\}\}$ inticates that starting at node 1, the connected nodes are $\{2, 3, 4, 5\}$. At the next level down each branch adds one of these nodes to the subgraph. For example, the leftmost branch indicates that if node 2 is chosen, then the adjacent nodes to $\{1, 2\}$ are $\{3, 4, 5, 6, 7, 8\}$. The RAND-ESU algorithm does not perform the full enumeration but uses a random sample at each level of the tree. Since we are considering three-node subgraphs, the tree has three levels, and a separate sampling rate can be chosen for each. The overall sampling rate is the product of the rates used at each level.

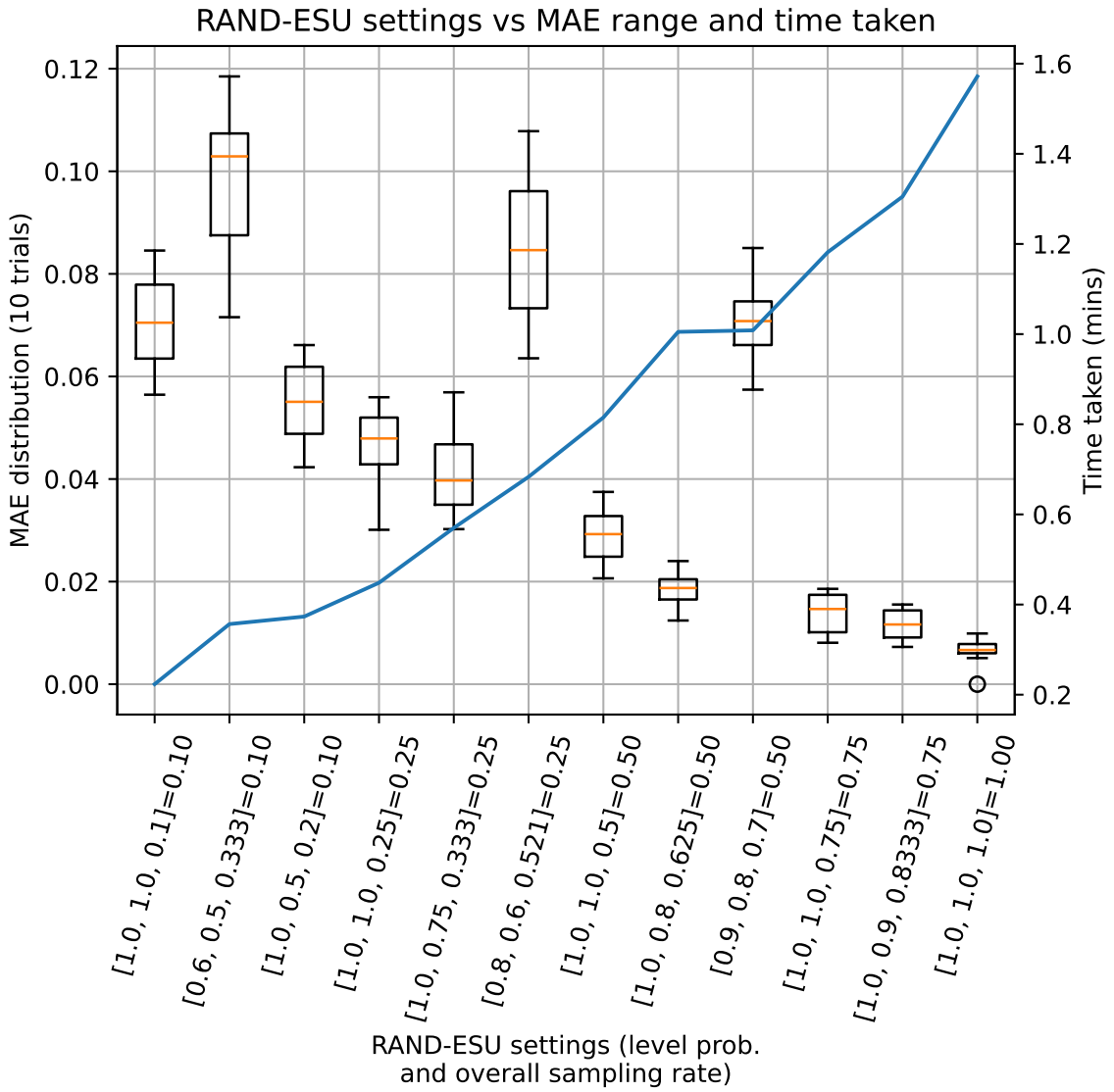


Figure C.2: Results of a parameter sweep of sampling rates at each level of the ESU tree (see Fig. C.1). The x-axis labels show the level sampling rate ant the overall sampling rate. For example level sampling rates of $1.0, 0.5, 0.2$ give an overall sampling rate of 0.1 ($= 1.0 \times 0.5 \times 0.2$). For each sampling rate the TSP is calculated and compared to the true TSP (using full enumeration of subgraphs, ie. a sampling rate of 1). The Mean Absolute Error (MAE) of these calculated TSP compared to the true TSP is calculated. The box plots show the distribution of MAEs for each setting of RAND-ESU sampling rates over 10 trials. Lower MAEs indicate a result closer to the true TSP. The time taken to calculate the TSP using the different RAND-ESU sampling settings is shown by the blue line (right-hand axis). The sampling rates $[1.0, 1.0, 0.5]$ were chosen for the experiments in Chapter 7 as they give an acceptable trade off between TSP accuracy (median MAE of 0.03 vs true TSP) and calculation time.

C.2 TSPs of Preferential Attachment Networks

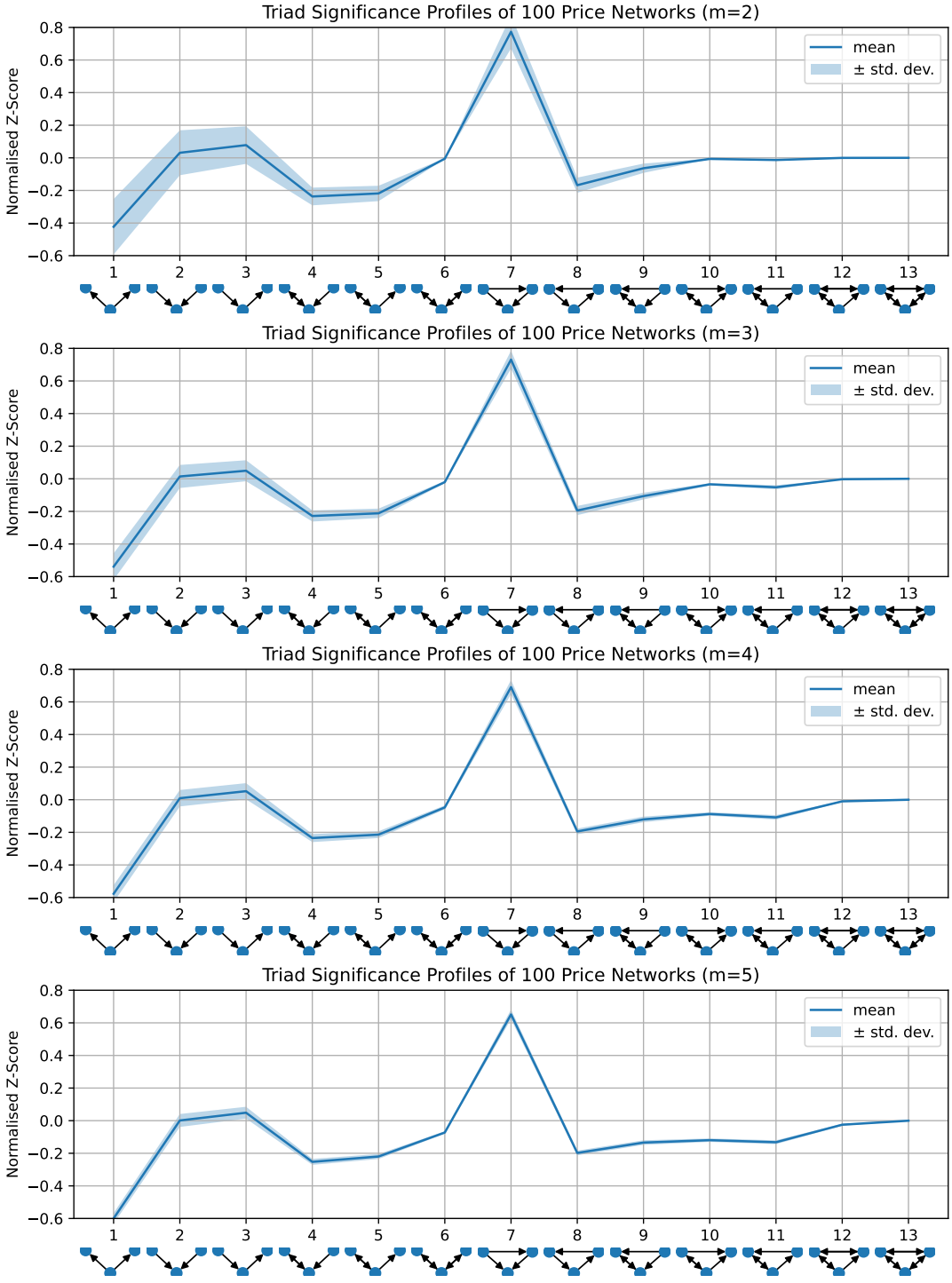


Figure C.3: Each plot show the mean TSP of 100 preferential attachment networks using the Price model (Price, 1976), with values of the m parameter, which specifies the out degree of newly added vertices, ranging from 2 to 5. The ± 1 standard deviation value range is shown using shading. A shown, the range of triad significance profiles of graphs constructed using Price preferential attachment is very limited, even with differing values of the m parameter.

Bibliography

Aceituno, P. V., Yan, G., & Liu, Y.-Y. (2020). Tailoring echo state networks for optimal learning. *iScience*, *23*(9), 101440. https://doi.org/10.1016/j.isci.2020.101440

Adami, C. (1998). *Introduction to artificial life*. Springer New York. https://doi.org/10.1007/978-1-4612-1650-6

Albert, R., & Barabási, A.-L. (2002). Statistical mechanics of complex networks. *Reviews of Modern Physics*, *74*(1), 47–97. https://doi.org/10.1103/RevModPhys.74.47

Alon, U. (2007). *An introduction to systems biology: Design principles of biological circuits*. Chapman & Hall/CRC.

Atkinson, T., Plump, D., & Stepney, S. (2018). Evolving graphs by graph programming. In M. Castelli, L. Sekanina, M. Zhang, S. Cagnoni, & P. García-Sánchez (Eds.), *Genetic programming* (pp. 35–51). Springer International Publishing. https://doi.org/10.1007/978-3-319-77553-1_3

Atkinson, T., Plump, D., & Stepney, S. (2021). Evolving graphs with semantic neutral drift. *Natural Computing*, *20*(1), 127–143. https://doi.org/10.1007/s11047-019-09772-4

Babai, L., & Kucera, L. (1979). Canonical labelling of graphs in linear average time [ISSN: 0272-5428]. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, 39–46. https://doi.org/10.1109/SFCS.1979.8

Baetens, J., Van der Meeren, W., & De Baets, B. (2017). On the dynamics of stochastic elementary cellular automata [Number: 1-2]. *JOURNAL OF CELLULAR AUTOMATA*, *12*(1), 63–80. Retrieved October 4, 2023, from http://hdl.handle.net/1854/LU-8197320

Banzhaf, W., & Yamamoto, L. (2015, July 3). *Artificial chemistries*. The MIT Press. https://doi.org/10.7551/mitpress/9780262029438.001.0001

Beckage, B., Kauffman, S., Gross, L. J., Zia, A., & Koliba, C. (2013). More complex complexity: Exploring the nature of computational irreducibility across physical, biological, and human social systems. In H. Zenil (Ed.), *Irreducibility and computational*

equivalence: 10 years after wolfram's a new kind of science (pp. 79–88). Springer. https://doi.org/10.1007/978-3-642-35482-3_7

Beer, R. D., McShaffrey, C., & Gaul, T. M. (2024). Deriving the intrinsic viability constraint of an emergent individual from first principles. https://doi.org/10.1162/isal_a_00739

Beggs, J. M., & Plenz, D. (2003). Neuronal avalanches in neocortical circuits [Publisher: Society for Neuroscience Section: Behavioral/Systems/Cognitive]. *Journal of Neuroscience*, *23*(35), 11167–11177. https://doi.org/10.1523/JNEUROSCI.23-35-11167.2003

Belousov, B. P. (1959). Периодически действующая реакция и ее механизм [A periodically acting reaction and its mechanism]. *Сборник рефератов по радиационной медицине*, (147).

Bennett, M. (2020). An attempt at a unified theory of the neocortical microcircuit in sensory cortex. *Frontiers in Neural Circuits*, *14*, 40. https://doi.org/10.3389/fncir.2020.00040

Boers, E. J. W., & Sprinkhuizen-Kuyper, I. G. (2001). Combined biological metaphors. In M. J. Patel, V. Honavar, & K. Balakrishnan (Eds.), *Advances in the evolutionary synthesis of intelligent agents* (pp. 153–183). MIT Press.

Boury, T., Ponty, Y., & Reinharz, V. (2023). Automatic exploration of the natural variability of RNA non-canonical geometric patterns with a parameterized sampling technique. https://doi.org/10.4230/LIPIcs.WABI.2023.20

Brackston, R. D., Lakatos, E., & Stumpf, M. P. H. (2018). Transition state characteristics during cell differentiation. *PLoS Computational Biology*, *14*(9), e1006405. https://doi.org/10.1371/journal.pcbi.1006405

Bray, D. (2009, May 26). *Wetware: A computer in every living cell*. Yale University Press.

Bruna, J., Zaremba, W., Szlam, A., & LeCun, Y. (2013, December 21). *Spectral networks and locally connected networks on graphs* [arXiv.org]. Retrieved June 20, 2025, from https://arxiv.org/abs/1312.6203v3

Bullmore, E., & Sporns, O. (2009). Complex brain networks: Graph theoretical analysis of structural and functional systems. *Nature Reviews. Neuroscience*, *10*(3), 186–198. https://doi.org/10.1038/nrn2575

Cai, J.-Y., Fürer, M., & Immerman, N. (1992). An optimal lower bound on the number of variables for graph identification. *Combinatorica*, *12*(4), 389–410. https://doi.org/10.1007/BF01305232

Carlini, N. (2020, April 1). *Digital logic gates on conway's game of life - part 1*. Retrieved August 10, 2025, from https://nicholas.carlini.com/writing/2020/digital-logic-game-of-life.html

Carroll, S. B., Carroll, J. W., Klaiss, J. P., & Olds, L. M. (2011). *Endless forms most beautiful: The new science of evo devo and the making of the animal kingdom*. Quercus.

Chan, B. W.-C. (2019). Lenia: Biology of artificial life. *Complex Systems*, *28*(3), 251–286. https://doi.org/10.25088/ComplexSystems.28.3.251

Chatzidimitriou, K. C., & Mitkas, P. A. (2010). A NEAT way for evolving echo state networks. *ECAI 2010*, 909–914. https://doi.org/10.3233/978-1-60750-606-5-909

Clune, J., Stanley, K. O., Pennock, R. T., & Ofria, C. (2011). On the performance of indirect encoding across the continuum of regularity. *IEEE Transactions on Evolutionary Computation*, *15*(3), 346–367. https://doi.org/10.1109/TEVC.2010.2104157

Cook, M. (2004). Universality in elementary cellular automata. *Complex Systems*.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Dale, M. (2018). Neuroevolution of hierarchical reservoir computers. https://doi.org/10.1145/3205455.3205520

Dale, M., Miller, J., Stepney, S., & Trefzer, M. (2019). A substrate-independent framework to characterize reservoir computers. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *475*, 20180723. https://doi.org/10.1098/rspa.2018.0723

Dale, M., O'Keefe, S., Sebald, A., Stepney, S., & Trefzer, M. A. (2021). Reservoir computing quality: Connectivity and topology. *Natural Computing*, *20*(2), 205–216. https://doi.org/10.1007/s11047-020-09823-1

Dawkins, R. (2003, October 3). The evolution of evolvability. In S. Kumar & P. J. Bentley (Eds.), *On growth, form and computers* (Illustrated edition, pp. 239–255). Academic Press.

Deb, K., Pratap, A., Agarwal, S., & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, *6*(2), 182–197. https://doi.org/10.1109/4235.996017

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems*, *29*. Retrieved June 20, 2025, from https://papers.nips.cc/paper_files/paper/2016/hash/04df4d434d481c5bb723be1b6df1ee65-Abstract.html

Deutsch, D. (1998). *The fabric of reality: Towards a theory of everything*. Penguin.

Dittrich, P., & Di Fenizio, P. S. (2007). Chemical organisation theory. *Bulletin of Mathematical Biology*, *69*(4), 1199–1231. https://doi.org/10.1007/s11538-006-9130-8

Durstewitz, D., Koppe, G., & Thurm, M. I. (2023). Reconstructing computational system dynamics from neural data with recurrent neural networks [Publisher: Nature Publishing Group]. *Nature Reviews Neuroscience*, *24*(11), 693–710. https://doi.org/10.1038/s41583-023-00740-7

Dye, C. A., Lee, J. K., Atkinson, R. C., Brewster, R., Han, P. L., & Bellen, H. J. (1998). The drosophila sanpodo gene controls sibling cell fate and encodes a tropomodulin homolog, an actin/tropomyosin-associated protein. *Development (Cambridge, England)*, *125*(10), 1845–1856. https://doi.org/10.1242/dev.125.10.1845

Eigen, M., & Schuster, P. (1977). A principle of natural self-organization. *Naturwissenschaften*, *64*(11), 541–565. https://doi.org/10.1007/BF00450633

Eom, Y.-H., Lee, S., & Jeong, H. (2006). Exploring local structural organization of metabolic networks using subgraph patterns. *Journal of Theoretical Biology*, *241*(4), 823–829. https://doi.org/10.1016/j.jtbi.2006.01.018

Erdős, P., & Rényi, A. (1959). On random graphs. i. *Publicationes Mathematicae Debrecen*, *6*(3), 290–297. https://doi.org/10.5486/PMD.1959.6.3-4.12

Erdős, P., Harary, F., & Tutte, W. T. (1965). On the dimension of a graph. *Mathematika*, *12*(2), 118–122. https://doi.org/10.1112/S0025579300005222

Evans, K. M. (2003). Larger than life: Threshold-range scaling of life's coherent structures. *Physica D: Nonlinear Phenomena*, *183*(1), 45–67. https://doi.org/10.1016/S0167-2789(03)00155-6

Floreano, D., & Mattiussi, C. (2008, August 22). *Bio-inspired artificial intelligence: Theories, methods, and technologies*. MIT Press.

Gallicchio, C. (2020, June 4). Sparsity in reservoir computing neural networks. https://doi.org/10.48550/arXiv.2006.02957

Gardner, M. (1970). The fantastic combinations of john conway's new solitaire game 'life'. *Mathematical Games, Scientific American*, *223*(4).

Gardner, M. (2001). *The colossal book of mathematics: Classic puzzles, paradoxes, and problems: Number theory, algebra, geometry, probability, topology, game theory, infinity, and other topics of recreational mathematics* (1st ed). Norton.

Gilpin, W. (2019). Cellular automata as convolutional neural networks. *Physical Review E*, *100*(3), 032402. https://doi.org/10.1103/PhysRevE.100.032402

Glass, L., & Mackey, M. (2010). Mackey-glass equation. *Scholarpedia*, *5*(3), 6908. https://doi.org/10.4249/scholarpedia.6908

Goudarzi, A., Shabani, A., & Stefanovic, D. (2015). Exploring transfer function nonlinearity in echo state networks. *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*, 1–8. https://doi.org/10.1109/CISDA.2015.7208637

Goudarzi, A., & Teuscher, C. (2016). Reservoir computing: Quo vadis? *Proceedings of the 3rd ACM International Conference on Nanoscale Computing and Communication*, 1–6. https://doi.org/10.1145/2967446.2967448

Gould, S. J. (2002). *Ontogeny and phylogeny*. The Belknap Press of Harvard Univ. Press.

Grattarola, D., Livi, L., & Alippi, C. (2021, October 27). Learning graph cellular automata. https://doi.org/10.48550/arXiv.2110.14237

Grohe, M. (2021). The logic of graph neural networks. *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 1–17. https://doi.org/10.1109/LICS52264.2021.9470677

Gruau, F. (1994). *Neural network synthesis using cellular encoding and the genetic algorithm* [Doctoral dissertation, Ecole Normale Superieure de Lyon]. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.5939

Gruau, F., & Whitley, D. (1993). Adding learning to the cellular development of neural networks: Evolution and the baldwin effect. *Evolutionary Computation*, *1*(3), 213–233. https://doi.org/10.1162/evco.1993.1.3.213

Gruau, F., Whitley, D., & Pyeatt, L. (1996). A comparison between cellular encoding and direct encoding for genetic neural networks. *Proceedings of the 1st annual conference on genetic programming*, 81–89.

Hall, B. K. (2003). Unlocking the black box between genotype and phenotype: Cell condensations as morphogenetic (modular) units. *Biology and Philosophy*, *18*(2), 219–247. https://doi.org/10.1023/A:1023984018531

Harary, F. (1969). *Graph theory*. Addison-Wesley Publishing Company.

Harvey, I. (2011). The microbial genetic algorithm [Series Title: Lecture Notes in Computer Science]. In G. Kampis, I. Karsai, & E. Szathmáry (Eds.), *Advances in artificial life. darwin meets von neumann* (pp. 126–133, Vol. 5778). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-21314-4_16

Hawkins, J. (2021). *A thousand brains: A new theory of intelligence* (First edition). Basic Books.

Heiney, K., Ramstad, O. H., Sandvig, I., Sandvig, A., & Nichele, S. (2019). Assessment and manipulation of the computational capacity of in vitro neuronal networks through criticality in neuronal avalanches. *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, 247–254. https://doi.org/10.1109/SSCI44817.2019.9002693

Heiney, K., Tufte, G., & Nichele, S. (2020). On artificial life and emergent computation in physical substrates. *arXiv:2009.04518 [cs, q-bio]*. Retrieved December 7, 2021, from http://arxiv.org/abs/2009.04518

Hickinbotham, S., Clark, E., Nellis, A., Stepney, S., Clarke, T., & Young, P. (2016). Maximizing the adjacent possible in automata chemistries. *Artificial Life*, *22*(1), 49–75. https://doi.org/10.1162/ARTL_a_00180

Hickinbotham, S., Stepney, S., & Hogeweg, P. (2021). Nothing in evolution makes sense except in the light of parasitism: Evolution of complex replication strategies [Publisher: Royal Society]. *Royal Society Open Science*, *8*(8), 210441. https://doi.org/10.1098/rsos.210441

Hiesinger, P. R. (2021). *The self-assembling brain*. Retrieved October 19, 2021, from https://press.princeton.edu/books/hardcover/9780691181226/the-self-assembling-brain

Hintze, A., Hiesinger, P. R., & Schossau, J. (2020). Developmental neuronal networks as models to study the evolution of biological intelligence. *ALife Workshop on the development of artificial neural networks*, 5. https://www.irit.fr/devonn/2020/07/13/hintze.html

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Holland, P. W., Laskey, K. B., & Leinhardt, S. (1983). Stochastic blockmodels: First steps. *Social Networks*, *5*(2), 109–137. https://doi.org/10.1016/0378-8733(83)90021-7

Hsieh, S.-M., Hsu, C.-C., & Hsu, L.-F. (2006). Efficient method to perform isomorphism testing of labeled graphs. In M. L. Gavrilova, O. Gervasi, V. Kumar, C. J. K. Tan, D. Taniar, A. Laganá, Y. Mun, & H. Choo (Eds.), *Computational science and its applications - ICCSA 2006* (pp. 422–431). Springer. https://doi.org/10.1007/11751649_46

Ilachinski, A., & Halpern, P. (1987). Structurally dynamic cellular automata by andrew ilachinski and paul halpern. *Complex Systems*, *1*(3), 503–527. https://www.complex-systems.com/abstracts/v01_i03_a07/

Inselberg, A. (1985). The plane with parallel coordinates. *The Visual Computer*, *1*(2), 69–91. https://doi.org/10.1007/BF01898350

Jaeger, H. (2001). The" echo state" approach to analysing and training recurrent neural networks-with an erratum note'. *Bonn, Germany: German National Research Center for Information Technology GMD Technical Report*, *148*.

Jaeger, H. (2007). Echo state network. *Scholarpedia*, *2*(9), 2330. https://doi.org/10.4249/scholarpedia.2330

Jiang, F., Berry, H., & Schoenauer, M. (2008). Supervised and evolutionary learning of echo state networks. Retrieved June 27, 2022, from https://hal.inria.fr/inria-00337235

Johns, C. (2023, April 30). On the rules of perfection: The simplest means and the richest effects. In C. Johns (Ed.), *Leibniz's discourse on metaphysics: A new translation and commentary*. Edinburgh University Press. https://doi.org/10.3366/edinburgh/9781474457774.003.0007

Kagaya, K., Kubota, T., & Nakajima, K. (2022, September 19). Self-organized criticality for dendritic readiness potential. https://doi.org/10.48550/arXiv.2209.09075

Kashtan, N., & Alon, U. (2005). Spontaneous evolution of modularity and network motifs [Publisher: Proceedings of the National Academy of Sciences]. *Proceedings of the National Academy of Sciences*, *102*(39), 13773–13778. https://doi.org/10.1073/pnas.0503610102

Kauffman, S. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets [Publisher: Elsevier]. *Journal of theoretical biology*, *22*(3), 437–467.

Kauffman, S. (1996, November 21). *At home in the universe: The search for the laws of self-organization and complexity* (Reprint edition). Oxford University Press.

Kitano, H. (1990). Designing neural networks using genetic algorithms with graph generation system. *Complex Syst.*

Kniemeyer, O., Barczik, G., Hemmerling, R., & Kurth, W. (2008). Relational growth grammars – a parallel graph transformation approach with applications in biology and architecture. *Applications of Graph Transformations with Industrial Relevance*, 152–167. https://doi.org/10.1007/978-3-540-89020-1_12

Kniemeyer, O., Buck-Sorlin, G. H., & Kurth, W. (2004). A graph grammar approach to artificial life [Conference Name: Artificial Life]. *Artificial Life*, *10*(4), 413–431. https://doi.org/10.1162/1064546041766451

Kodjabachian, J., & Meyer, J.-A. (1998). Evolution and development of neural controllers for locomotion, gradient-following, and obstacle-avoidance in artificial insects [Conference Name: IEEE Transactions on Neural Networks]. *IEEE Transactions on Neural Networks*, *9*(5), 796–812. https://doi.org/10.1109/72.712153

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*. MIT Press.

Krastev, M. S. (2018, August). *Spiky RBN: A sub-symbolic artificial chemistry* [Doctoral dissertation, University of York]. Retrieved July 18, 2022, from https://etheses.whiterose.ac.uk/24115/

Langton, C. G. (1990). Computation at the edge of chaos: Phase transitions and emergent computation. *Physica D: Nonlinear Phenomena*, *42*(1), 12–37. https://doi.org/10.1016/0167-2789(90)90064-V

Langton, C. G. (1984). Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, *10*(1), 135–144. https://doi.org/10.1016/0167-2789(84)90256-2

Langton, C. G. (1989). *Artificial life: Proceedings of an interdisciplinary workshop on the synthesis and simulation of living systems*. Addison-Wesley Longman Publishing Co., Inc.

Langton, C. G. (1996, May 9). Artificial life. In *The philosophy of artificial life* (pp. 39–94). Oxford University Press.

Legenstein, R., & Maass, W. (2007). Edge of chaos and prediction of computational performance for neural circuit models. *Neural Networks*, *20*(3), 323–334. https://doi.org/10.1016/j.neunet.2007.04.017

Lehman, J., & Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, *19*(2), 189–223. https://doi.org/10.1162/EVCO_a_00025

Leier, A., Kuo, P. D., & Banzhaf, W. (2007). Analysis of preferential network motif generation in an artificial regulatory network model created by duplication and divergence [Publisher: World Scientific Publishing Co.]. *Advances in Complex Systems*, *10*(2), 155–172. https://doi.org/10.1142/S0219525907000994

Levy, S. (1993). *Artificial life: A report from the frontier where computers meet biology*. Vintage Books.

Lindenmayer, A. (1968). Mathematical models for cellular interactions in development II. simple and branching filaments with two-sided inputs. *Journal of Theoretical Biology*, *18*(3), 300–315. https://doi.org/10.1016/0022-5193(68)90080-5

Liu, L., Du, B., xu, J., & Tong, H. (2019). G-finder: Approximate attributed subgraph matching. *2019 IEEE International Conference on Big Data (Big Data)*, 513–522. https://doi.org/10.1109/BigData47090.2019.9006525

Lobo, D., Vico, F. J., & Dassow, J. (2011). Graph grammars with string-regulated rewriting. *Theoretical Computer Science*, *412*(43), 6101–6111. https://doi.org/10.1016/j.tcs.2011.07.004

Lones, M. A., Turner, A. P., Fuente, L. A., Stepney, S., Caves, L. S. D., & Tyrrell, A. M. (2013). Biochemical connectionism. *Natural Computing*, *12*(4), 453–472. https://doi.org/10.1007/s11047-013-9400-y

Lui, J. C., & Baron, J. (2011). Mechanisms limiting body growth in mammals. *Endocrine Reviews*, *32*(3), 422–440. https://doi.org/10.1210/er.2011-0001

Luke, S. (2000). Evolving graphs and networks with edge encoding: Preliminary report.

Lukoševičius, M. (2012). A practical guide to applying echo state networks [Series Title: Lecture Notes in Computer Science]. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade* (pp. 659–686, Vol. 7700). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8_36

Luo, L. (2016). *Principles of neurobiology*. Garland Science Taylor & Francis Group.

Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, *14*(11), 2531–2560. https://doi.org/10.1162/089976602760407955

Mackey, M. C., & Glass, L. (1977). Oscillation and chaos in physiological control systems [Publisher: American Association for the Advancement of Science]. *Science*, *197*(4300), 287–289. https://doi.org/10.1126/science.267326

Mariot, L. (2024, May 5). Insights gained after a decade of cellular automata-based cryptography. https://doi.org/10.48550/arXiv.2405.02875

Marr, C., & Huett, M.-T. (2009). Outer-totalistic cellular automata on graphs. *Physics Letters A*, *373*(5), 546–549. https://doi.org/10.1016/j.physleta.2008.12.013

Martin, O., Odlyzko, A. M., & Wolfram, S. (1984). Algebraic properties of cellular automata. *Communications in Mathematical Physics*, *93*(2), 219–258. https://doi.org/10.1007/BF01223745

Matsuzaki, F. (2000). Asymmetric division of *Drosophila* neural stem cells: A basis for neural diversity. *Current Opinion in Neurobiology*, *10*(1), 38–44. https://doi.org/10.1016/S0959-4388(99)00052-5

Maturana, H. R., & Varela, F. J. (1980). *Autopoiesis and cognition: The realization of the living* (Vol. 42). Springer Netherlands. https://doi.org/10.1007/978-94-009-8947-4

McCormack, J. (2008). Evolutionary l-systems. In P. F. Hingston, L. C. Barone, & Z. Michalewicz (Eds.), *Design by evolution: Advances in evolutionary design* (pp. 169–196). Springer. https://doi.org/10.1007/978-3-540-74111-4_10

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, *5*(4), 115–133. https://doi.org/10.1007/BF02478259

Michel, O. (2007, décembre). *There's Plenty of Room for Unconventional Programming Languages or Declarative Simulations of Dynamical Systems (with a Dynamical Structure)* [thèse de doct., Université d'Évry Val-d'Essonne].

Miller, J. F. (2003, June 6). Cartesian genetic programming. In J. F. Miller (Ed.), *Cartesian genetic programming* (Vol. 43). https://doi.org/10.1007/978-3-642-17310-3

Miller, J. F. (2004). Evolving a self-repairing, self-regulating, french flag organism. In K. Deb (Ed.), *Genetic and evolutionary computation – GECCO 2004* (pp. 129–139). Springer. https://doi.org/10.1007/978-3-540-24854-5_12

Miller, J. F. (2021). IMPROBED: Multiple problem-solving brain via evolved developmental programs. *Artificial Life*, 1–36. https://doi.org/10.1162/artl_a_00346

Miller, J. F., & Smith, S. (2006). Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, *10*(2), 167–174. https://doi.org/10.1109/TEVC.2006.871253

Miller, J. F., & Thomson, P. (2003). A developmental method for growing graphs and circuits. In A. M. Tyrrell, P. C. Haddow, & J. Torresen (Eds.), *Evolvable systems: From biology to hardware* (pp. 93–104). Springer. https://doi.org/10.1007/3-540-36553-2_9

Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., & Alon, U. (2002). Network motifs: Simple building blocks of complex networks [Publisher: American Association for the Advancement of Science]. *Science*, *298*(5594), 824–827. https://doi.org/10.1126/science.298.5594.824

Milo, R., Itzkovitz, S., Kashtan, N., Levitt, R., Shen-Orr, S., Ayzenshtat, I., Sheffer, M., & Alon, U. (2004). Superfamilies of evolved and designed networks. *Science*, *303*(5663), 1538–1542. https://doi.org/10.1126/science.1089167

Minsky, M. (1952, January). *A neural-analogue calculator based upon a probability model of reinforcement*. Harvard University Psychological Laboratories. Cambridge, MA.

Mirsky, L. (1971). *Transversal theory: An account of some aspects of combinatorial mathematics*. Academic Press.

Mordvintsev, A., Randazzo, E., & Fouts, C. (2022). Growing isotropic neural cellular automata. https://doi.org/10.1162/isal_a_00552

Mordvintsev, A., Randazzo, E., Niklasson, E., & Levin, M. (2020). Growing neural cellular automata. *Distill*, *5*(2), e23. https://doi.org/10.23915/distill.00023

Mountcastle, V. B. (1997). The columnar organization of the neocortex. *Brain*, *120*(4), 701–722. https://doi.org/10.1093/brain/120.4.701

Najarro, E., Sudhakaran, S., Glanois, C., & Risi, S. (2022). HyperNCA: Growing developmental networks with neural cellular automata. *arXiv:2204.11674 [cs]*. Retrieved May 3, 2022, from http://arxiv.org/abs/2204.11674

Najarro, E., Sudhakaran, S., & Risi, S. (2023). Towards self-assembling artificial neural networks through neural developmental programs. *Proceedings of the 2023 Artificial Life Conference*. https://doi.org/10.1162/isal_a_00697

Newman, M. E. J. (2004). Fast algorithm for detecting community structure in networks [Publisher: American Physical Society]. *Physical Review E*, *69*(6), 066133. https://doi.org/10.1103/PhysRevE.69.066133

Nichele, S., & Molund, A. (2017). Deep learning with cellular automaton-based reservoir computing. *Complex Systems*, *26*(4), 319–340. https://doi.org/10.25088/ComplexSystems.26.4.319

Nichele, S., Ose, M. B., Risi, S., & Tufte, G. (2017). CA-NEAT: Evolved compositional pattern producing networks for cellular automata morphogenesis and replication. *IEEE Transactions on Cognitive and Developmental Systems*. https://doi.org/10.1109/TCDS.2017.2737082

O'Neill, M., & Ryan, C. (2003, January 1). *Grammatical evolution: Evolutionary automatic programming in an arbitrary language* (Vol. 4). https://doi.org/10.1007/978-1-4615-0447-4

Owens, N., & Stepney, S. (2010). The game of life rules on penrose tilings: Still life and oscillators. In A. Adamatzky (Ed.), *Game of life cellular automata* (pp. 331–378). Springer London. https://doi.org/10.1007/978-1-84996-217-9_18

Peixoto, T. P. (2014a). The graph-tool python library. *figshare*. https://doi.org/10.6084/m9.figshare.1164194

Peixoto, T. P. (2014b). Hierarchical block structures and high-resolution model selection in large networks [Publisher: American Physical Society]. *Physical Review X*, *4*(1), 011047. https://doi.org/10.1103/PhysRevX.4.011047

Peixoto, T. P. (2019). Bayesian stochastic blockmodeling [Section: 11 _eprint: https://onlinelibrary.wiley.com/do
In *Advances in network clustering and blockmodeling* (pp. 289–332). John Wiley & Sons, Ltd. https://doi.org/10.1002/9781119483298.ch11

Plump, D. (2017). From imperative to rule-based graph programs. *Journal of Logical and Algebraic Methods in Programming*, *88*, 154–173. https://doi.org/10.1016/j.jlamp.2016.12.001

Pontes-Filho, S., Lind, P., Yazidi, A., Zhang, J., Hammer, H., Mello, G. B. M., Sandvig, I., Tufte, G., & Nichele, S. (2020). A neuro-inspired general framework for the evolution of stochastic dynamical systems: Cellular automata, random boolean networks and echo state networks towards criticality. *Cognitive Neurodynamics*, *14*(5), 657–674. https://doi.org/10.1007/s11571-020-09600-x

Postel, M., Karam, A., Pézeron, G., Schneider-Maunoury, S., & Clément, F. (2019). A multi-scale mathematical model of cell dynamics during neurogenesis in the mouse cerebral cortex. *BMC Bioinformatics*, *20*(1), 470. https://doi.org/10.1186/s12859-019-3018-8

Price, D. J. d. S. (1965). Networks of scientific papers [Publisher: American Association for the Advancement of Science]. *Science*, *149*(3683), 510–515. https://doi.org/10.1126/science.149.3683.510

Price, D. J. d. S. (1976). A general theory of bibliometric and other cumulative advantage processes [_eprint: https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/asi.4630270505]. *Journal of the American Society for Information Science*, *27*(5), 292–306. https://doi.org/10.1002/asi.4630270505

Prill, R. J., Iglesias, P. A., & Levchenko, A. (2005). Dynamic properties of network motifs contribute to biological network organization [Publisher: Public Library of Science]. *PLOS Biology*, *3*(11), e343. https://doi.org/10.1371/journal.pbio.0030343

Prusinkiewicz, P., Hanan, J., & Měch, R. (2000). An l-system-based plant modeling language. In M. Nagl, A. Schürr, & M. Münch (Eds.), *Applications of graph transformations with industrial relevance* (pp. 395–410). Springer. https://doi.org/10.1007/3-540-45104-8_31

Qiao, J., Li, F., Han, H., & Li, W. (2017). Growing echo-state network with multiple subreservoirs [Conference Name: IEEE Transactions on Neural Networks and Learning Systems]. *IEEE Transactions on Neural Networks and Learning Systems*, *28*(2), 391–404. https://doi.org/10.1109/TNNLS.2016.2514275

Ravasz, E., Somera, A. L., Mongru, D. A., Oltvai, Z. N., & Barabási, A.-L. (2002). Hierarchical organization of modularity in metabolic networks. *Science*, *297*(5586), 1551–1555. https://doi.org/10.1126/science.1073374

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 25–34. https://doi.org/10.1145/37401.37406

Rosen, R. (1991). *Life itself: A comprehensive inquiry into the nature, origin, and fabrication of life*. Columbia Univ. Press.

Rozenberg, G. (Ed.). (1997). *Handbook of graph grammars and computing by graph transformation* (Vols. 3). World Scientific.

Ryan, K., Lu, Z., & Meinertzhagen, I. A. (2016). The CNS connectome of a tadpole larva of ciona intestinalis (l.) highlights sidedness in the brain of a chordate sibling (E. Marder, Ed.) [Publisher: eLife Sciences Publications, Ltd]. *eLife*, *5*, e16962. https://doi.org/10.7554/eLife.16962

Sakai, J. (2020). Core concept: How synaptic pruning shapes neural wiring during development and, possibly, in disease [Publisher: National Academy of Sciences Section: Core Concepts]. *Proceedings of the National Academy of Sciences*, *117*(28), 16096–16099. https://doi.org/10.1073/pnas.2010281117

Sayama, H. (1999). Toward the realization of an evolving ecosystem on cellular automata. *Proceedings of the Fourth International Symposium on Artificial Life and Robotics (AROB 4th '99)*, 254–257. https://www.researchgate.net/publication/2577056_Toward_the_Realization_of_an_Evolving_Ecosystem_on_Cellular_Automata

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model [Conference Name: IEEE Transactions on Neural Networks]. *IEEE Transactions on Neural Networks*, *20*(1), 61–80. https://doi.org/10.1109/TNN.2008.2005605

Schiff, J. L. (2011, October 18). *Cellular automata: A discrete view of the world* [Google-Books-ID: uXJC2C2sRbIC]. John Wiley & Sons.

Seoane, L. F. (2019). Evolutionary aspects of reservoir computing [Publisher: Royal Society]. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *374*(1774), 20180377. https://doi.org/10.1098/rstb.2018.0377

Shatz, C. J. (1996). Emergence of order in visual system development. *Proceedings of the National Academy of Sciences of the United States of America*, *93*(2), 602–608. Retrieved January 18, 2022, from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC40098/

Shellman, E. R., Chen, Y., Lin, X., Burant, C. F., & Schnell, S. (2014). Metabolic network motifs can provide novel insights into evolution: The evolutionary origin of eukaryotic organelles as a case study. *Computational biology and chemistry*, *53PB*, 242–250. https://doi.org/10.1016/j.compbiolchem.2014.09.006

Shin, J. K. (2016). Application of cellular automata for a generative art system [Publisher: [Leonardo, The MIT Press]]. *Leonardo*, *49*(5), 431–396. Retrieved August 10, 2025, from https://www.jstor.org/stable/24916800

Siddiqi, A., & Lucas, S. (1998). A comparison of matrix rewriting versus direct encoding for evolving neural networks. *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, 392–397. https://doi.org/10.1109/ICEC.1998.699787

Sipper, M. (1999). The emergence of cellular computing [Conference Name: Computer]. *Computer*, *32*(7), 18–26. https://doi.org/10.1109/2.774914

Stanley, K. O., D'Ambrosio, D., & Gauci, J. (2009). A hypercube-based indirect encoding for evolving large-scale neural networks, 39.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *10*(2), 99–127. https://doi.org/10.1162/106365602320169811

Sullivan, K. G., Emmons-Bell, M., & Levin, M. (2016). Physiological inputs regulate species-specific anatomy during embryogenesis and regeneration [Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/19420889.2016.1192733]. *Communicative & Integrative Biology*, *9*(4), e1192733. https://doi.org/10.1080/19420889.2016.1192733

Tavares, J., Kreutzer, C., & Fedor, A. (2015). Neuro-cellular automata: Connecting cellular automata, neural networks and evolution.

Teuscher, C. (2022). Revisiting the edge of chaos: Again? *Biosystems*, *218*, 104693. https://doi.org/10.1016/j.biosystems.2022.104693

Tomita, K., Kurokawa, H., & Murata, S. (2002). Graph automata: Natural expression of self-reproduction. *Physica D: Nonlinear Phenomena*, *171*(4), 197–210. https://doi.org/10.1016/S0167-2789(02)00601-2

Turing, A. M. (1948). Intelligent machinery. Retrieved February 26, 2021, from https://weightagnostic.github.io/papers/turing1948.pdf

Turing, A. M. (1952). The chemical basis of morphogenesis [Publisher: The Royal Society]. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, *237*(641), 37–72. Retrieved March 27, 2020, from https://www.jstor.org/stable/92463

Ulam, S. (1950). Random processes and transformations. *Proceedings of the International Conference of Mathematicians*, *2*, 264–275. https://www.mathunion.org/fileadmin/ICM/Proceedings/ICM1950.2/ICM1950.2.ocr.pdf

Uragami, D., Gunji, Y.-P., & Department of Intermedia Art and Science School of Fundamental Science and Engineering Waseda University. (2022). Universal criticality in reservoir computing using asynchronous. *Complex Systems*, *31*(1), 103–121. https://doi.org/10.25088/ComplexSystems.31.1.103

Valentim, C. A., Rabi, J. A., & David, S. A. (2023). Cellular-automaton model for tumor growth dynamics: Virtualization of different scenarios. *Computers in Biology and Medicine*, *153*, 106481. https://doi.org/10.1016/j.compbiomed.2022.106481

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł. u., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*. Retrieved July 9, 2025, from https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

von Neumann, J. (1967, April 1). *Theory of self-reproducing automata* (A. W. Burks, Ed.). University of Illinois Press.

Waddington, C. H. (1957). *The strategy of the genes. a discussion of some aspects of theoretical biology. with an appendix by h. kacser.* London: George Allen & Unwin, Ltd.

Walter, A., Wu, S., Tyrrell, A. M., McDaid, L., McElholm, M., Sumithran, N. T., Harkin, J., & Trefzer, M. A. (2023). Artificial neural microcircuits for use in neuromorphic system design. https://doi.org/10.1162/isal_a_00581

Watts, D. J., & Strogatz, S. H. (1998). Collective dynamics of 'small-world' networks [Publisher: Nature Publishing Group]. *Nature*, *393*(6684), 440–442. https://doi.org/10.1038/30918

Weisfeiler, B., & Leman, A. (1968). The reduction of a graph to canonical form and the algebra which appears therein [English translation by G. Ryabov]. *NTI*. https://www.iti.zcu.cz/wl2018/pdf/wl_paper_translation.pdf

Wenemoser, D., & Reddien, P. W. (2010). Planarian regeneration involves distinct stem cell responses to wounds and tissue absence. *Developmental Biology*, *344*(2), 979–991. https://doi.org/10.1016/j.ydbio.2010.06.017

Wernicke, S. (2006). Efficient detection of network motifs [Conference Name: IEEE/ACM Transactions on Computational Biology and Bioinformatics]. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *3*(4), 347–359. https://doi.org/10.1109/TCBB.2006.51

White, J. G., Southgate, E., Thomson, J. N., & Brenner, S. (1997). The structure of the nervous system of the nematode caenorhabditis elegans [Publisher: Royal Society]. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, *314*(1165), 1–340. https://doi.org/10.1098/rstb.1986.0056

Wills, P., & Meyer, F. G. (2020). Metrics for graph comparison: A practitioner's guide. *PLOS ONE*, *15*(2), e0228728. https://doi.org/10.1371/journal.pone.0228728

Wolf-Gladrow, D. A. (2000). *Lattice gas cellular automata and lattice boltzmann models* (Vol. 1725). Springer. https://doi.org/10.1007/b72010

Wolfram, S. (2002, May 14). *A new kind of science* (1st edition). Wolfram Media.

Wolfram, S. (2020, June 19). *A project to find the fundamental theory of physics*. Wolfram Media, Inc.

Wolfram, S. (2025). Towards a computational formalization for foundations of medicine. *Stephen Wolfram Writings*. Retrieved July 28, 2025, from https://writings.stephenwolfram.com/2025/02/towards-a-computational-formalization-for-foundations-of-medicine/

Wolpert, L. (1969). Positional information and the spatial pattern of cellular differentiation. *Journal of Theoretical Biology*, *25*(1), 1–47. https://doi.org/10.1016/S0022-5193(69)80016-0

Wringe, C., Stepney, S., & Trefzer, M. A. (2023). Modelling and evaluating restricted ESNs. In D. Genova & J. Kari (Eds.), *Unconventional computation and natural computation* (pp. 186–201). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-34034-5_13

Wringe, C., Trefzer, M., & Stepney, S. (2024, May 10). Reservoir computing benchmarks: A review, a taxonomy, some best practices. https://doi.org/10.48550/arXiv.2405.06561

Wuensche, A., Lesser, M., & Lesser, M. J. (1992, September 20). *Global dynamics of cellular automata: An atlas of basin of attraction fields of one-dimensional cellular automata*. Andrew Wuensche.

Wulff, N., & Hertz, J. A. (1992). Learning cellular automaton dynamics with neural networks. *Advances in Neural Information Processing Systems*, *5*. Retrieved November 10, 2022, from https://papers.nips.cc/paper/1992/hash/d6c651ddcd97183b2e40bc464231c962-Abstract.html

Yaskin, V. A. (2011). Seasonal changes in hippocampus size and spatial behavior in mammals and birds. *Biology Bulletin Reviews*, *1*(3), 279. https://doi.org/10.1134/S2079086411030108

Yildiz, I. B., Jaeger, H., & Kiebel, S. J. (2012). Re-visiting the echo state property. *Neural Networks: The Official Journal of the International Neural Network Society*, *35*, 1–9. https://doi.org/10.1016/j.neunet.2012.07.005

Zhabotinsky, A. M. (1964). Периодический процесс окисления малоновой кислоты растворе [The periodical process of oxidation of malonic acid solution]. *Биофизика*, (9).

Zhang, L., He, C., Lai, Y., Wang, Y., Kang, L., Liu, A., Lan, C., Su, H., Gao, Y., Li, Z., Yang, F., Li, Q., Mao, H., Chen, D., Chen, W., Kaufmann, K., & Yan, W. (2023). Asymmetric gene expression and cell-type-specific regulatory networks in the root of bread wheat revealed by single-cell multiomics analysis. *Genome Biology*, *24*(1), 65. https://doi.org/10.1186/s13059-023-02908-x

Zhao, L., Beverlin, B., Netoff, T., & Nykamp, D. (2011). Synchronization from second order network connectivity statistics. *Frontiers in Computational Neuroscience*, *5*. Retrieved February 25, 2024, from https://www.frontiersin.org/articles/10.3389/fncom.2011.00028

Zupanc, G. K. H., Zupanc, F. B., & Sipahi, R. (2019). Stochastic cellular automata model of tumorous neurosphere growth: Roles of developmental maturity and cell death. *Journal of Theoretical Biology*, *467*, 100–110. https://doi.org/10.1016/j.jtbi.2019.01.028