Proceedings Paper:

Waldegrave, Riversdale, Stepney, Susan orcid.org/0000-0003-3146-5401 and Trefzer, Martin A orcid.org/0000-0002-6196-6832 (2023) Developmental Graph Cellular Automata. In: ALIFE 2023: Ghost in the Machine: Proceedings of the 2023 Artificial Life Conference:. ALife 2023, 24-28 Jul 2023, University of Hokkaido. ACM , JPN

https://doi.org/10.1162/isal_a_00658

Developmental Graph Cellular Automata

Riversdale Waldegrave[1], Susan Stepney[1] and Martin A. Trefzer[2]

[1]Department of Computer Science, University of York, UK
[2]School of Physics, Engineering and Technology, University of York, UK
rraw500@york.ac.uk

Abstract

We present a system for growing graphs which can be thought of as an extension of the update rules used by Cellular Automata. As in Neural Cellular Automata, these rules are encoded in the real-valued weight matrix of a neural network. This should make the system easy to evolve, allowing it to be used as an evolutionary-developmental method of creating graph structures for use as recurrent neural networks or substrates in Reservoir Computing. Here we conduct a random search experiment and characterise five different classes of behaviour of the system. The most interesting of these is when the graph grows for a number of timesteps before naturally coming to a halt as it enters an attractor. This behaviour is seen more frequently than might be expected and contrasts with most developmental systems in which growth must be stopped by external intervention. There are clear parallels with biological morphogenetic processes where growth naturally comes to a halt.

Introduction

Graphs appear in many places in Artificial Life, from Artificial Neural Networks (ANNs) to Kauffman's Random Boolean Networks (RBNs) (Kauffman, 1969). They are particularly useful for representing spatially discrete dynamical systems. Such systems have interesting computational properties. For example, several simple updating rules in Wolfram's one-dimensional Elementary Cellular Automata (ECA) (Wolfram, 2002) have been shown to create dynamics which can be harnessed for Turing complete computation (Rule 110, for example) (Cook, 2004). 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 recurrent neural networks represented as graphs (Jaeger, 2001). The links between these various types of spatially discrete dynamical systems become clear when they are all represented as graph structures (Pontes-Filho et al., 2020).

These uses of graph-structured objects for computing generally use static graphs. However, there is a separate field of research focused on developing or growing graphs. Linking these two areas, and harnessing the dynamics of developmental graphs for computation, is an intriguing area for research. The biological metaphors often applied to ANNs or dynamical systems usually lack a description of how these computational structures are formed in the first place. This point is made by Hiesinger (2021), who laments that ANNs took their inspiration from neuroscience at a time when it was believed that neurons in the brain were connected randomly. In fact this is far from the case and infant brains are by no means blank slates comparable to randomly initialised ANNs.

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. For example, HyperNEAT (Stanley et al., 2009) uses a small evolved neural network which is then "queried" to discover the connection weight between every pair of nodes in a larger network. This results in ANNs with structural regularities that can usefully mirror geometric properties in the problem domain (for example, the layout of a checkerboard, or relative pixel locations in an image).

The system introduced here encodes developmental rules for graphs in the weights of a small "internal" neural network which determines each cell's behaviour. Many developmental approaches do not naturally terminate and have to be arbitrarily halted at some point (eg. Hintze et al. (2020)). 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.

The paper is organised as follows. We first review related work in the field of Cellular Automata (CAs) and highlight some approaches for growing graphs. We then introduce the model of *Developmental Graph Cellular Automata* which uses CA-like updating rules on irregular graphs, which update not only node states, but also the graph structure by duplicating and removing nodes. This is followed by a section detailing the results of an experiment which shows that this system can frequently reach an attractor state in which growth comes to a halt naturally. We conclude with a brief discussion of some biological metaphors that could be applied to the system, and outline future work in using evolution to search for interesting behaviour.

Related Work

The present work takes inspiration from two separate fields. Firstly, recent work in Neural and Graph Cellular Automata (NCA and GCA respectively) has highlighted that CAs can be viewed as discrete dynamical systems on graphs with a lattice topology, in which the update rules may be represented as arbitrary functions. Secondly, the field of rewriting rules in general, and graph rewriting in particular provides various different approaches to encoding developmental systems.

Neural and Graph Cellular Automata

The idea of replacing the update rule of a CA with a neural network was introduced by Wulff and Hertz (1992). It was extended to incorporate evolution by Tavares et al. (2015) who coined the term *Neural Cellular Automata* (NCA). Gilpin (2019) further highlighted the equivalence between CAs and Convolutional Neural Networks, whilst Mordvintsev et al. (2020, 2022) trained the neural networks to produce growing patterns of cells on the CA grid. In those papers the term "growing" refers to a growing pattern within the fixed space of the CA grid; when we talk about "growth" in the present work, we mean nodes being added to the graph, ie. the "space" itself expanding and changing topology. This has more in common with the way Wolfram's hypergraphs (Wolfram, 2020) frame graph rewriting as a reconfiguration of space itself.

Random Boolean Networks (RBNs) (Kauffman, 1996) 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 CAs, RBNs are based on graphs with random connectivity rather than regular lattices. The original RBNs are still regular in graph-theoretic terms as each node has the same in-degree. Again unlike CAs, 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.

One approach to bridging the gap between RBNs and CAs is Graph Cellular Automata (GCA) (Marr and Huett, 2009), which use graphs with random connectivity and uniform degree like RBNs, and use the same update function at every node like CAs. Unlike RBNs they are not constrained to two-state (Boolean) systems. Grattarola et al. (2021) have linked GCA and NCA to create Graph Neural Cellular Automata. These are CAs structured as graphs and use a neural network for their update rule. These share some similarities with the present system, but do not allow the addition or removal of nodes, so cannot be used to grow.

Najarro et al. (2022) combine NCAs with the idea of HyperNEAT. The CA grid acts like the "substrate" in HyperNEAT in that the value of each cell is interpreted as a weight in the final larger neural network, which again has a predefined size and structure.

Growing Graphs

The graph topology of all the systems mentioned above remains fixed after initialisation. By contrast, there are other systems that focus on structural changes. L-Systems are a simple example of this type of system. They consist of string rewriting rules in which all replacements are made in parallel (Lindenmayer, 1968). They 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. A novel use of L-Systems with relevance to the present work is for the development of graph structures to be used as neural networks (Boers and Sprinkhuizen-Kuyper, 2001).

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 ECAs, 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 we effectively have a CA with dynamic structure, where the number of cells can grow or shrink (although the topology always remains linear).

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 is the GP2 graph programming language (Plump, 2012). Another approach is Wolfram's hypergraph rewriting system (Wolfram, 2020). One feature of such systems 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

system	connectivity	update	structure
Cellular Automaton	lattice	hom	fixed
Random Boolean Network	regular graph	het	fixed
Graph Cellular Automaton	regular graph	hom	fixed
Context-sensitive L-System	row	hom	dynamic
present system	irregular graph	hom	dynamic

Table 1: Examples of spatially discrete dynamical systems with different types of connectivity, update function (homogeneous, hom, and heterogeneous, het) and structure.

compared with CAs and L-Systems in which the rules are always applied in parallel. Graph rewriting systems generally use rules that replace one subgraph with another, and so the state changes take place at the level of the subgraph rather than the individual node. This too contrasts with CAs, in which the updating rules use only information local to each node.

The DGCA Model

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 (the parameter K in Kauffman's formalism (Kauffman, 1969)). This means that 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. 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 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, 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.

One solution to the problem of variable and unordered inputs is to use the counts of nodes in each state as an input to the transition function. CA update rules of this type are called 'totalistic'; those that separately take into account the counts of neighbour cell states *and* the state of the central cell are called 'outer-totalistic'. The Game of Life CA (Gardner, 1970) is in the class of outer-totalistic rules. It is difficult to use totalistic rules on CAs structured as irregular graphs, however, since the counts of neighbour cells in each state can be unbounded. Owens and Stepney (2010) use outer-totalistic rules on Penrose tilings, which are irregular graphs, but these have an upper bound to the number of cell neighbours.

Marr and Huett (2009) present a formalism for applying outer-totalistic rules to irregular graph CAs by calculating the *proportion* of neighbouring cells in each state: they divide the counts of neighbours in each state by the degree of the central node. Since they are concerned with 2-state systems ($\Sigma = \{0, 1\}$), 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. However, it is not possible to extend this to systems with more than two states, since the flipping operation would be undefined. The system described here presents an alternative solution which can be used to represent outer-totalistic rules on irregular graph CAs.

One of the aims of the present system is to use the generated graphs as RNNs or an RC substrate, so we use directed graphs. During the development process, the edge directions do not indicate the flow of information into node transition functions, but rather are used to provide 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). One feature of this design decision is that an ECA can be fully represented in the proposed system, since the left and right neighbour of a cell/node can be represented by an incoming and an outgoing edge. This is discussed further in the next section.

Preprocessing Step

Consider a system of N nodes where each node can be in one of S states.

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:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{S} \tag{1}$$

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:

$$\mathbf{C}_{\mathbf{out}} = \mathbf{A} \cdot \mathbf{S} \tag{2}$$

$$\mathbf{C}_{\mathbf{in}} = \mathbf{A}^T \cdot \mathbf{S} \tag{3}$$

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 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 $\mathbf{A}$:

$$\mathbf{D}_{ij} = \begin{cases} \deg(v_i), & \text{if } i = j \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

$$\mathbf{L} = \mathbf{D} - \mathbf{A} \tag{5}$$

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

$$\mathbf{F} = \mathbf{L} \cdot \mathbf{S} \tag{6}$$

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 minus 1.

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

$$\mathbf{L}_{\mathbf{out}} = \mathbf{D}_{\mathbf{out}} - \mathbf{A} \tag{7}$$

$$\mathbf{L}_{\mathbf{in}} = \mathbf{D}_{\mathbf{in}} - \mathbf{A}^T \tag{8}$$

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 $\mathbf{F}_{\mathbf{out}}$ and $\mathbf{F}_{\mathbf{in}}$:

$$\mathbf{F}_{\mathbf{out}} = \mathbf{L}_{\mathbf{out}} \cdot \mathbf{S} \tag{9}$$

$$\mathbf{F}_{\mathbf{in}} = \mathbf{L}_{\mathbf{in}} \cdot \mathbf{S} \tag{10}$$

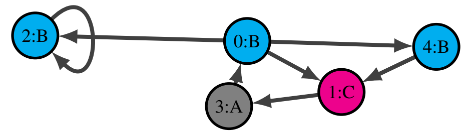

Figure 1: An example network. Nodes are numbered 0–4 and each is in the the state A, B or C (also indicated by the node colour). The adjacency matrix, matrix of node states, and graph Laplacians for this network are given in equations 11 and 13.

Together, $\mathbf{F}_{\mathbf{out}}$ and $\mathbf{F}_{\mathbf{in}}$ 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 1. The corresponding adjacency matrix $\mathbf{A}$, state matrix $\mathbf{S}$, degree matrices, "out Laplacian", "in Laplacian" and filtered states are:

$$\mathbf{A} = \begin{bmatrix} 0&1&1&0&1 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \end{bmatrix} \quad \mathbf{S} = \begin{bmatrix} 0&1&0 \\ 0&0&1 \\ 0&1&0 \\ 1&0&0 \\ 0&1&0 \end{bmatrix} \tag{11}$$

$$\mathbf{D}_{\mathbf{out}} = \begin{bmatrix} 3&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \end{bmatrix} \quad \mathbf{D}_{\mathbf{in}} = \begin{bmatrix} 1&0&0&0&0 \\ 0&2&0&0&0 \\ 0&0&2&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \end{bmatrix} \tag{12}$$

$$\mathbf{L}_{\mathbf{out}} = \begin{bmatrix} 3&-1&-1&0&-1 \\ 0&1&0&-1&0 \\ 0&0&0&0&0 \\ -1&0&0&1&0 \\ 0&-1&0&0&1 \end{bmatrix} \quad \mathbf{L}_{\mathbf{in}} = \begin{bmatrix} 1&0&0&-1&0 \\ -1&2&0&0&-1 \\ -1&0&1&0&0 \\ 0&-1&0&1&0 \\ -1&0&0&0&1 \end{bmatrix} \tag{13}$$

$$\mathbf{F}_{\mathbf{out}} = \begin{bmatrix} 0&1&-1 \\ -1&0&1 \\ 0&0&0 \\ 1&-1&0 \\ 0&1&-1 \end{bmatrix} \quad \mathbf{F}_{\mathbf{in}} = \begin{bmatrix} -1&1&0 \\ 0&-2&2 \\ 0&0&0 \\ 1&0&-1 \\ 0&0&0 \end{bmatrix} \tag{14}$$

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 $\mathbf{G}$ 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.

$$\mathbf{G}_{(N\times(2S+1))} = [\mathbf{F_{in}}_{(N\times S)}, \mathbf{F_{out}}_{(N\times S)}, \mathbf{1}_{(N\times 1)}] \quad (15)$$

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 1, the matrix $\mathbf{G}$ is:

$$\mathbf{G} = \begin{bmatrix} -1 & 1 & 0 & 0 & 1 & -1 & 1 \\ 0 & -2 & 2 & -1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & -1 & 1 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \quad (16)$$

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

$$\mathbf{O}_{(N\times(S+2))} = \mathbf{G}_{(N\times(2S+1))} \cdot \mathbf{W}_{((2S+1)\times(S+2))} \quad (17)$$

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:

$$\mathbf{W} = \begin{bmatrix} -0.2 & -0.7 & -0.3 & 0.1 & -0.8 \\ -0.1 & -0.1 & 0.4 & 0.4 & -0.3 \\ 0.3 & -0.8 & 0.0 & 0.5 & -1. \\ 0.9 & -0.4 & -0.6 & 0.9 & 0.9 \\ 0.9 & -0.6 & 1. & -0.4 & 0.5 \\ 0.1 & 0.7 & 0.8 & 0.7 & 0.3 \\ 0.2 & 0.3 & -0.6 & 0.1 & 0.7 \end{bmatrix} \quad (18)$$

Multiplying this by the $\mathbf{G}$ matrix in equation 16 gives the output matrix (rounded to one decimal place):

$$\mathbf{O} = \begin{bmatrix} 1.1 & -0.4 & 0.3 & -0.7 & 1.4 \\ 0.2 & 0.0 & 0.0 & 0.1 & -1.3 \\ 0.2 & 0.3 & -0.6 & 0.1 & 0.7 \\ -0.3 & 0.6 & -2.5 & 1.0 & 1.3 \\ 1. & -1.0 & -0.4 & -1.0 & 0.9 \end{bmatrix} \quad (19)$$

Taking the index of the maximum value in each row gives $[5, 1, 5, 5, 1]$. This is the final output of the transition function and indicates what 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 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.

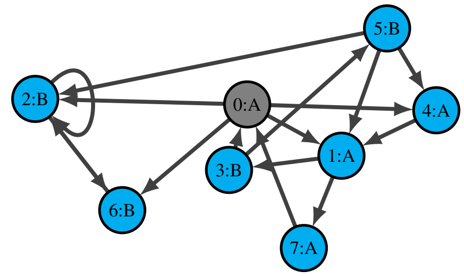

Figure 2: The graph in Figure 1 after one development step, using the SLP weights matrix in Equation 18 as explained in the text.

Comparison with Neural Cellular Automata

The procedure described above has much in common with the way a neural cellular automaton (NCA) works (Wulff and Hertz, 1992; Tavares et al., 2015; Mordvintsev et al., 2020), with a few important differences. NCAs are fixed-topology grid based CAs 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 for the reasons outlined above, 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.

GNCAs (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. An experiment was conducted there, 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. 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. Since part of the purpose of NCAs 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 DGCAs as simply as possible, hence the use of SLPs. However, future work will experiment with the use of MLPs to power the update behaviour at each timestep.

Experiment

The long term aim is to use DCGAs to grow graphs with suitable properties for different applications. The system is well suited to exploration using an evolutionary algorithm, since the update rule is entirely specified by the real-valued weight matrix of the SLP, **W**.

Before embarking on full-blown search, however, it is useful to perform an initial investigation to explore and characterise some of the different behaviours that are possible in this system. For this initial investigation, we use random search over **W**.

Detecting Attractors

Conventional CAs with a finite number of cells have a finite state space and therefore ultimately always reach an attractor, since 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 say for sure whether growth will stop for a given ruleset. 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. This is a non-trivial task. 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.

Five classes of growth behaviour

We identify five classes of behaviour the system can display in terms of its growth. These are based simply on the total number of nodes in the graph.

- **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.

- **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.

- **Static Networks** do not grow at all and remain fixed at the size of the seed network. 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 reach an attractor in which the number of nodes remains static: 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.

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 "static" 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. When the system displays this type of behaviour it is not doing anything more than a conventional CA can do.

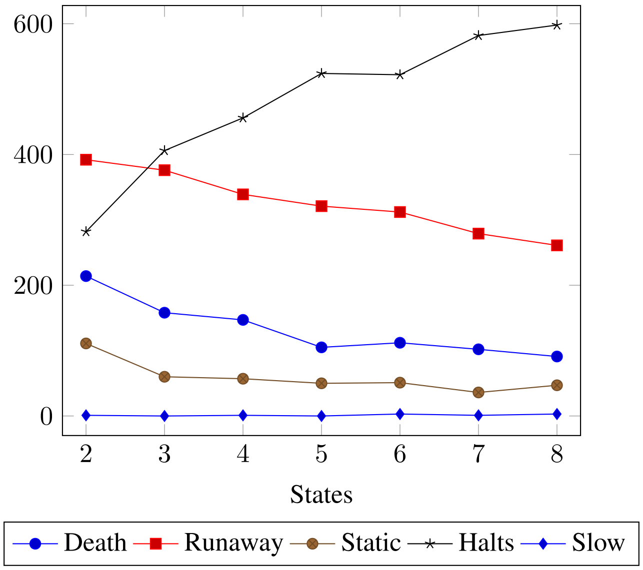

Figure 3: Chart showing the number of occurrences of each of the five classes of behaviour for 1000 trials with each number of states in the system. As the number of states in the system increases, runaway growth behaviour is seen less, and halting growth behaviour is seen more.

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). Each seed graph has the same structure: a ring of 8 nodes with edges connecting them in one direction. The states of nodes in the seed graph are randomly initialised. We run each of these seed graphs for a maximum of 256 timesteps. Runaway growth is defined as the graph growing to greater than 256 nodes within this time. In most cases, within the 256 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 (fewer than 5 out of 1000 trials), the system shows neither runaway growth nor reaches an attractor within the 256 timesteps, in which case we class the behaviour as "slow growth". The results are shown in Figure 3.

For a 2-state system, the runaway behaviour is the most common and is observed in 392 out of the 1000 trials. The halting growth behaviour is the next most common with 282 occurrences. This alone is a surprising result, given that the state space is unbounded (at each timestep the system can move to an entirely different state space). In spite of 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 5-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 256 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, in a 100 node graph with two states the number of possible overall states of the system is $2^{100} \approx 10^{30}$, whereas in a 100 node graph with five 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.

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. 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. Newly created nodes are added at the 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 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 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 5. The graph size remains fairly small throughout. After a transient of 45 steps, a 7-step attractor cycle is found.

Conclusion

We have introduced an extension of the concept of Graph Cellular Automata which allows the structure of the graph to develop over time. The system displays not only the *dynamics* of nodes changing state, but also the *meta-dynamics*

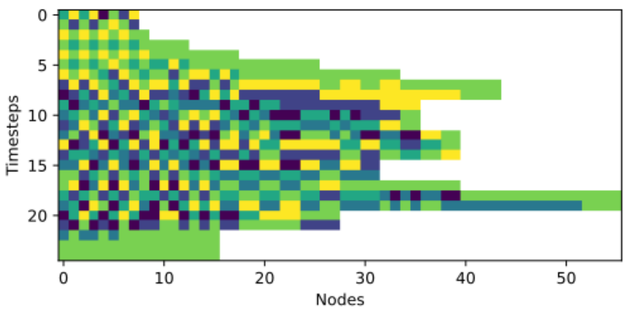

Figure 4: An example space-time diagram for a 6-state system in which growth halts. Timesteps proceed down the y-axis. At each timestep, the state of every node is concatenated into a vector, shown as a line containing six different possible colours. This line gets longer as the graph grows. It can be seen that the graph grows and shrinks over time before reaching a stable state. The final (bottom) two lines are the same, indicating that the system has reached a point attractor.

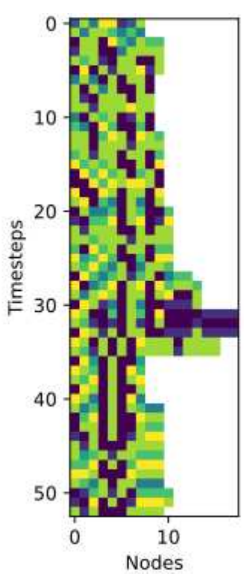

Figure 5: This system has a transient length of 45 steps followed by a 7-step attractor cycle (the line at timestep 52 is the same as that at timestep 46).

of the structure of the graph itself changing. The experiment 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.

Future Work

A key future aim is to use evolutionary search on the weights of the SLP that defines the update rule to find interesting behaviours. What constitutes an interesting behaviour is an open question, but an initial aim is to use novelty search to see what the system is capable of. Subsequent work will examine whether it is possible to evolve rules which generate graphs with particular properties.

Replacing the SLP with a MLP would enable a wider set of update rules to be used by the system, at the expense of having more parameters to evolve.

The graphs which are developed could also be used as recurrent neural networks (RNN) themselves once they have stopped developing. One goal for the evolutionary process could therefore be to create graphs that perform well on a particular task. The Reservoir Computing paradigm could be useful in this, as it allows a fixed weight RNN to be used for tasks by training a simple readout layer. An extension of this goal might be to push the system into different attractors during the growth process by perturbing some of the states. Each different attractor might result in a network which is suited to a different task. Thus a single ruleset (ie. SLP weights) and seed network could be thought of as analogous to a stem cell which could be differentiated to become suitable for a wide range of tasks during its growth process.

Source Code

Source code for the DGCA system and the experiment described here here can be found at `https://github.com/rvrsdl/alife2023_dgca`.

References

Boers, E. J. W. and Sprinkhuizen-Kuyper, I. G. (2001). Combined Biological Metaphors. In Patel, M. J., Honavar, V., and Balakrishnan, K., editors, *Advances in the Evolutionary Synthesis of Intelligent Agents*, pages 153–183. MIT Press.

Cook, M. (2004). Universality in Elementary Cellular Automata. *Complex Systems*.

Gardner, M. (1970). Mathematical Games. *Scientific American*, 223(4):120–123.

Gilpin, W. (2019). Cellular automata as convolutional neural networks. *Physical Review E*, 100(3).

Grattarola, D., Livi, L., and Alippi, C. (2021). Learning Graph Cellular Automata. *arXiv:2110.14237 [cs]*.

Hiesinger, P. R. (2021). *The Self-Assembling Brain*. Princeton University Press.

Hintze, A., Hiesinger, P. R., and Schossau, J. (2020). Developmental neuronal networks as models to study the evolution of biological intelligence. *ALife Workshop on the development of artificial neural networks*.

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(34).

Kauffman, S. (1996). *At Home in the Universe: The Search for the Laws of Self-Organization and Complexity*. Oxford University Press.

Kauffman, S. A. (1969). Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of Theoretical Biology*, 22(3):437–467.

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.

Maass, W., Natschläger, T., and Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560.

Marr, C. and Huett, M.-T. (2009). Outer-totalistic cellular automata on graphs. *Physics Letters A*, 373(5):546–549.

Mordvintsev, A., Randazzo, E., and Fouts, C. (2022). Growing Isotropic Neural Cellular Automata. In *ALife 2022*. MIT Press.

Mordvintsev, A., Randazzo, E., Niklasson, E., and Levin, M. (2020). Growing Neural Cellular Automata. *Distill*, 5(2):e23.

Najarro, E., Sudhakaran, S., Glanois, C., and Risi, S. (2022). HyperNCA: Growing Developmental Networks with Neural Cellular Automata. *arXiv:2204.11674 [cs]*.

Owens, N. and Stepney, S. (2010). The Game of Life Rules on Penrose Tilings: Still Life and Oscillators. In Adamatzky, A., editor, *Game of Life Cellular Automata*, pages 331–378. Springer London, London.

Plump, D. (2012). The Design of GP 2. *ETPCS*, 82:1–16.

Pontes-Filho, S., Lind, P., Yazidi, A., Zhang, J., Hammer, H., Mello, G. B. M., Sandvig, I., Tufte, G., and 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.

Rozenberg, G., editor (1997). *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore ; New Jersey.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The Graph Neural Network Model. *IEEE TNN*, 20(1):61–80.

Stanley, K. O., D'Ambrosio, D., and Gauci, J. (2009). A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks.

Tavares, J., Kreutzer, C., and Fedor, A. (2015). Neuro-Cellular Automata: Connecting Cellular Automata, Neural Networks and Evolution. In *Complex Systems Summer School*.

Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.

Wolfram, S. (2020). *A Project to Find the Fundamental Theory of Physics*. Wolfram Media.

Wulff, N. and Hertz, J. A. (1992). Learning Cellular Automaton Dynamics with Neural Networks. In *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann.