Constructing neural networks with pre-specified dynamics

Camilo J. Mininni[1,2], B. Silvano Zanutto[1,2]

June 19, 2023

Abstract

A main goal in neuroscience is to understand the computations carried out by neural populations that give animals their cognitive capabilities. Neural network models allow to formulate explicit hypothesis regarding the algorithms instantiated in the dynamic of a neural population, its firing statistics, and the underlying connectivity. Neural networks can be defined by a small set of parameters, carefully chosen to obtain specific functions, or by a large set of free parameters, fitted with optimization algorithms that minimize a given loss function. In this work we alternatively propose a method to make a detailed adjustment of the networks dynamic and firing statistics to better answer questions that link dynamics, structure and function. Our algorithm – termed generalized Firing-to-Parameter (gFTP) – provides a way to construct binary recurrent neural networks whose dynamic strictly follows a user pre-specified transition graph that details the transitions between population firing states triggered by sequences of stimulus presentations. Our main contribution is a procedure that detects when a transition graph is not realizable in terms of a neural network, and makes the necessary modifications in order to obtain a new transition graph that is realizable and preserves all the information encoded in the transitions of the original graph. With a realizable transition graph, gFTP assigns values to the network firing states associated with each vertex in the graph, and finds the synaptic weight matrices by solving a set of linear separation problems. We test gFTP performance by constructing networks with random dynamics, continuous attractor-like dynamics that encode position in 2-dimensional space, and discrete attractor dynamics. We discuss several ways in which gFTP can be used to evaluate degeneracy in the space of dynamics that instantiate the same function, how network dynamic constrains firing statistics, and the interdependencies between dynamics, neural representation and connectivity.

Key words: neural networks, brain dynamics, model fitting.

[1]Instituto de Biología y Medicina Experimental, Consejo Nacional de Investigaciones Científicas y Técnicas, Argentina. [2]Instituto de Ingeniería Biomédica, Universidad de Buenos Aires, Argentina. Correspondence to: cmininni@fi.uba.ar

1 Introduction

Neural network models are invaluable tools in neuroscience, since they allow to make explicit hypothesis relating behaviour with neurophysiology. Networks can be constructed from the bottom up, taking experimental evidence to define the neuron input-output mapping and their connectivity. Then, the emergent properties of the system can be studied, expecting that they recapitulate experimental observations not used as model hypothesis. Conversely, a normative approach is also possible, in which we start from the function the network is proposed to have, together with a few assumptions about network connectivity and neuron activation dynamics, and then we fit the free parameters of the model to obtain the best task performance, as measured by a loss function. This approach has increasingly gained momentum due to advances in deep learning that have provided researchers with new tools and hardware to fit increasingly complex models to increasingly complex tasks [1]. The models obtained have reproduced experimental observations regarding the nature of neural coding, distributed representation and population dynamics [2, 3, 4]. Modeling approaches of this kind stem from a paradigm that proposes to understand neural computation as algorithms instantiated in the low dimensional dynamic of large neural populations [5, 6]. With neural recording technology allowing more simultaneous measurements [7] fitting neural networks with many neurons, capable of complex dynamics is going to be more and more necessary. Nonetheless, network optimization is a complex endeavour. The way fitted networks solve the task, and how they represent information depend on the choosing of all hyperparameters, making it difficult to draw conclusions about how each hypothesis impacts on network behaviour. Moreover, a fitting algorithm may converge preferentially to a specific family of solutions, giving no information about how many other networks there are that can solve the task in qualitatively different ways. Neither the inability to achieve low error in a given task rules out the proposed hypotheses as candidates to explain a phenomenon. Furthermore, there are many synaptic weight configurations that lead to the same network output, making it difficult to relate network function with network connectivity. Training is usually easier if networks have many neurons (with many parameters), hence there is a bias towards choosing networks with more neurons than the minimum required, making the number of weight configurations even larger. However, some of these shortcomings can be handle by leveraging the particular goals of modelling in neuroscience. First, we expect our network to solve tasks alike the ones solved by animals in behavioural experiments. These tasks are usually easy to solve, in the sense that we already know how to solve them. We can even enumerate more than one algorithm that solves the same task. Second, we usually have some hypotheses about how the modelled system represents and processes information. For instance, we may have evidence that neural populations in the prefrontal cortex handle working memory demands by instantiating discrete attractors that encode relevant stimuli[8, 9], or that networks in the entorhinal cortex encode position in a 2D continuous attractor[10, 11, 12]. Therefore, it would be of interest to have a network that solves a task through a given *a priori* dynamic,

and then study how the underlying connectivity relates to function. However, specifying the dynamic through minimization of a loss function by means of an optimization algorithm could be very hard to accomplish. In a previous work we proposed an algorithm that constructed networks by solving a system of linear equations [13]. In this algorithm there was one equation for each transition in the transition graph, the coefficients were the firings of each neuron in the source state, and the variables were the membrane potentials in the target state. The critical aspect of the algorithm was a way to define the membrane potentials such that the system of equations had an exact solution. We successfully employed the algorithm in a particular task -a sequence memory task-, but how to generalize the algorithm to arbitrary transition graphs with any number of stimuli remained an open question. In the present work we described a much more general algorithm: the generalized Firing-to-Parameter (gFTP) algorithm. This new version takes a transition graph as input, decides if there is a neural network capable of following the transition graph exactly, and if not, expands the original graph to construct a new graph that fulfill two conditions: it is realizable in terms of a neural network, and it retains all of the information present in the original graph. Then, we define the network firing states associated with each vertex in the graph, in such a way that linear combinations induced by stimuli are satisfied. Finally, weight matrices are found with the accelerated perceptron algorithm [14]. We give a detailed description of the rationale behind the algorithm, and evaluate its performance on random transition graphs, graphs that instantiates a 2-dimensional continuous attractor that encodes position, and discrete attractor dynamics.

2 The Generalized Firing to Parameter Algorithm

Network model and consistency conditions

We will consider recurrent neural networks composed of binary neurons (Fig. 1a). Neurons receive inputs from sensory neurons and from all other neurons in the network. Sensory neurons codify stimuli in a one-hot fashion. The network's activity at iteration k is dictated by:

$$
\begin{aligned}
u_k &= \mathbf{y}_k^{\mathrm{T}}\mathbf{w}_y + \mathbf{z}_{k-1}^{\mathrm{T}}\mathbf{w}_r \\
z_k &= \mathrm{H}(u_k)
\end{aligned}
$$

where u is the preactivation of the neuron, $\mathbf{y}$ is the vector that collects the activations of sensory neurons (one neuron per stimulus), $\mathbf{z}$ is the vector that collects the activations of neurons in the recurrent network and H the Heaviside function. Vectors $\mathbf{w}_y$ and $\mathbf{w}_r$ are the synaptic weights of the inputs coming from the sensory neurons and neuron in the recurrent network respectively, for a given neuron in the network. Conversely, matrices $\mathbf{W}_y$ and $\mathbf{W}_r$ collect in their columns the synaptic weight vectors $\mathbf{w}_y$ and $\mathbf{w}_r$ of all neurons in the recurrent network (Fig. 1f).

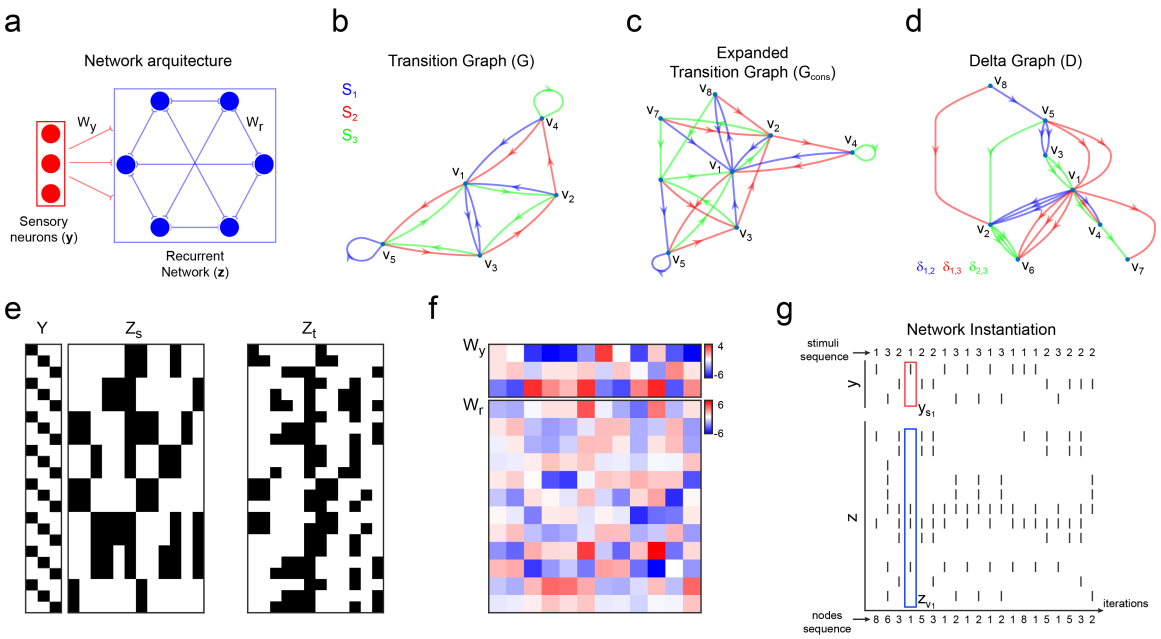

Figure 1: Recurrent neural networks that follow a pre-specified transition graph. **(a)** Recurrent networks of binary neurons receive connections from neurons that encode stimuli in a one-hot fashion. **(b)** An example of transition graph that defines the desired network dynamic. Each vertex represents a different neural population state in the recurrent network. Directed arcs depict transitions from source to target states that are triggered by different stimuli, encoded by the sensory neurons and shown with arrows of different colors. **(c)** Graph en (b) was expanded until no inconsistencies remained. **(d)** Graph D computed from the expanded graph in (c). Arcs color represents the arc label (delta value, one for each stimuli pair). The graph shows no cycles, confirming that the expanded graph is now realizable in terms of a neural network. **(e)** Matrices $\mathbf{Y}$, $\mathbf{Z}_s$, and $\mathbf{Z}_d$, constructed by backtracking through the neurons activations that differentiate all vertices and comply with delta constraints. **(f)** Synaptic weight matrix found through the accelerated perceptron algorithm, for which matrix $[\mathbf{Y}, \mathbf{Z}_s]$ are the samples, and matrix $\mathbf{Z}_t$ are the binary classes. **(g)** Raster plot showing the activity of a neural network constructed to follow expanded graph in (c) during presentation of a sequence of random stimuli.

We want to find the synaptic weight matrices $\mathbf{W}_y$,$\mathbf{W}_r$ that instantiates a network that unfolds some target dynamics. Network dynamics can be specified in the form of a labeled multi digraph $G = (V, K, S, f_s, f_t, l_G)$, where V is the set of vertices $(v_{1,}v_{2,}...v_{N_v})$, $K \subset \mathbb{N}$ is the set of transitions specified (the arcs in the graph), $S \subset \mathbb{N}$ the set of stimuli (which act as labels) the network can have as inputs, f_s the function that takes a transition and gives the source vertex, f_t the function that takes a transition and gives a target vertex, and l_G**,** the function that takes a transition and gives the label (the stimulus that triggers that transition) (Fig. 1b). Conversely, a matrix $\mathbf{G}$ can be constructed, such that row k is $(l_G(k), f_s(k), f_t(k))$. When unambiguous, we will use subindexes to indicate that a given state vector is associated with another vector in a certain way (for example, that $\mathbf{z}_v$ is the $\mathbf{z}$ that codifies vertex v).

We define matrices $\mathbf{Y}$, $\mathbf{Z}_s$ and $\mathbf{Z}_t$ such that each row collects the activations of the sensory neurons ($\mathbf{Y}$), and the activation of the recurrent network in a source state ($\mathbf{Z}_s$) and in a target state ($\mathbf{Z}_t$), for each transition/row k (i.e. $\mathbf{Y}(k,:) = \mathbf{y}_{\mathbf{G}(k,1)}$, $\mathbf{Z}_s(k,:) = \mathbf{z}_{\mathbf{G}(k,2)}$) and $\mathbf{Z}_t(k,:) = \mathbf{z}_{\mathbf{G}(k,3)}$,(Fig. 1e)). A network that instantiates graph G must satisfy:

$$\mathbf{C}\,\mathbf{W} = \mathbf{U}, \;\; s.t. \; \mathrm{H}(\mathbf{U}) = \mathbf{Z}_t, \tag{1}$$

with

$$\mathbf{C} = \begin{pmatrix} \mathbf{Y} & \mathbf{Z}_s \end{pmatrix} \tag{2}$$

$$\mathbf{W} = \begin{pmatrix} \mathbf{W}_y \\ \mathbf{W}_r \end{pmatrix} \tag{3}$$

This is a problem of linear separability. Matrix $\mathbf{W}$ exists iff. $rank(\mathbf{C}) = rank((\mathbf{C}, \mathbf{U}))$, (i.e. the augmented matrix $(\mathbf{C}, \mathbf{U})$ and matrix $\mathbf{C}$ present the same linear combinations), and the signs of values in $\mathbf{U}$ are consistent with $\mathbf{Z}_t$. If $\mathbf{Y}$, $\mathbf{Z}_s$ and $\mathbf{Z}_t$ are chosen properly, then a solution for equation 1 exists and can be found with recently proposed sped-up versions of the simple perceptron algorithm. Finding suitable matrices is a hard combinatorial problem. However, we can leverage some regularities to reduce its complexity. For a start, we must first find if there exist such $\mathbf{Z}_s$, $\mathbf{Z}_t$ in the first place, since it could be the case that, for some transition graphs, the interdependencies between source and target vertices preclude its instantiation in the shape of any neural network. In this sense, we must find out if the graph is realizable, and if not, we must find a way to turn the graph into a realizable graph, without loosing the critical aspects of the target dynamic.

There is a set of linear combinations that directly stems from the transition graph and can be detected and corrected if needed. Lets consider two transitions that start from the same source vertex v, but are triggered by different stimuli:

$$\begin{pmatrix} \mathbf{y}_{s_i} & \mathbf{z}_v \end{pmatrix} \begin{pmatrix} \mathbf{W}_y \\ \mathbf{W}_r \end{pmatrix} = \mathbf{u}_{s_i,v} \tag{4}$$

$$\begin{pmatrix} \mathbf{y}_{s_j} & \mathbf{z}_v \end{pmatrix} \begin{pmatrix} \mathbf{W}_y \\ \mathbf{W}_r \end{pmatrix} = \mathbf{u}_{s_j,v} \tag{5}$$

where $\mathbf{u}_{s_i,v}$, $\mathbf{u}_{s_j,v}$ are the preactivations after the network receives stimulus s_i or s_j, starting from the same vertex v. Subtracting both sides of the equations we get:

$$\begin{pmatrix} \mathbf{y}_{s_i} - \mathbf{y}_{s_j}, & \mathbf{z}_v - \mathbf{z}_v \end{pmatrix} \begin{pmatrix} \mathbf{W}_y \\ \mathbf{W}_r \end{pmatrix} = \left(\mathbf{u}_{s_i,v} - \mathbf{u}_{s_j,v} \right) \tag{6}$$

$$\left(\mathbf{y}_{s_i} - \mathbf{y}_{s_j} \right) \mathbf{W}_y = \left(\mathbf{u}_{s_i,v} - \mathbf{u}_{s_j,v} \right) \tag{7}$$

Since both transitions start from the same vertex, the inputs that a neuron receives that come from the other neurons in the recurrent network cancel, and the difference in $\mathbf{u}$ only depends on the input from the stimuli. Vector $\delta_{i,j} := (\mathbf{y}_{s_i} - \mathbf{y}_{s_j})\,\mathbf{W}_y$ is a constant that does not depend on the source vertex. It follows that vector $\delta\mathbf{u}_{i,j,v} := \mathbf{u}_{s_i,v} - \mathbf{u}_{s_j,v}$ does not depend on v either, since it must be equal to $\delta_{i,j}$ for all v. For a given neuron n:

$$\mathbf{u}_{s_i,v}(n) - \mathbf{u}_{s_j,v}(n) \geq 0 \Rightarrow \mathbf{z}_{s_i,v}(n) \geq \mathbf{z}_{s_j,v}(n) \Rightarrow (\mathbf{z}_{s_i,v}(n), \mathbf{z}_{s_j,v}(n)) \in \{(1,0),(0,0),(1,1)\} \tag{8}$$

$$\mathbf{u}_{s_i,v}(n) - \mathbf{u}_{s_j,v}(n) < 0 \Rightarrow \mathbf{z}_{s_i,v}(n) \leq \mathbf{z}_{s_j,v}(n) \Rightarrow (\mathbf{z}_{s_i,v}(n), \mathbf{z}_{s_j,v}(n)) \in \{(0,1),(0,0),(1,1)\} \tag{9}$$

Therefore, all pairs of $\mathbf{z}$ that are reached from a given pair of stimuli from the same source are restricted on the values that can jointly adopt. Not taking this fact into consideration will lead to $\mathbf{Z}_s$ and $\mathbf{Z}_t$ that are not realizable in any neural network. Moreover, it could be the case that the graph itself does not allow *any* $\mathbf{Z}_s$, $\mathbf{Z}_t$ to be realizable. Consider for example the graph in Fig. 2a:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 3 \\ 1 & 3 & 3 \\ 2 & 3 & 1 \end{pmatrix} \tag{g$_1$}$$

If a neuron differentiates v_1 from v_2, by firing when the network is in state v_1 ($z_{v_1} = 1$) and not firing in state v_2 ($z_{v_2} = 0$), then $\delta_{1,2} > 0$. This in turn means that the neuron must not fire in state v_3 ($z_{v_3} = 0$), because if it fired then $\delta_{i,j} < 0$, leading to a contradiction to the value of delta defined by transitions 1 and 2. We say that the value of zero assigned to v_2 "propagates" to v_3. This value propagates from v_3 to v_1 too ($z_{v_3} = z_{v_1} = 0$), from where we started. But this is in contradiction with the original value for z_{v_1}. The inconsistency is circumvented only when v_1 and v_2 are not differentiated by any neuron, but in this case there would be only two network

states instead of three. Therefore, the graph is unrealizable as it is. However, the graph can be modified to make it realizable. Let's consider the modified graph in Fig. 2b:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 3 \\ 1 & 3 & 3 \\ 2 & 3 & 4 \\ 1 & 4 & 1 \\ 2 & 4 & 2 \end{pmatrix} \tag{g$_2$}$$

We have replaced v_1 in transition 6 by a new vertex (v_4), and added transitions for this new vertex as source, equal to the transitions of the vertex it replaces (transitions with v_1 as a source vertex). Since the new node does not appear anywhere else, a vector $\mathbf{z}$ can be assign according to any $\delta_{i,j}$, without possibility that it contradicts a previous assignment, as it occurred in graph (g$_2$). On the other hand, this new vertex encodes the same information than the replaced vertex (i.e. the occurrence of stimulus s_2 starting from v_3) and leads to the same transitions too (going to v_1 through s_1 and v_2 through s_2). Therefore, the graph has been altered and it is now realizable, and all the information encoded in the network states is preserved. The new node is a twin of the node it replaces, in the sense that it leads to the same targets through the same stimuli. Note that, if we had a graph of the type:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 2 & 2 \\ 2 & 2 & 3 \\ 1 & 3 & 3 \\ 2 & 3 & 1 \\ 1 & 4 & 3 \\ 2 & 4 & 1 \end{pmatrix} \tag{g$_3$}$$

we would need to replace node v_1 in two transitions (transition 6 and 8) in order to attain consistency. We will say that a transition graph is *delta consistent* if there are $\mathbf{Z}_s$ $\mathbf{Z}_t$ and $\mathbf{U}$ such that relations (8-9) are satisfied. If this is not the case (because differentiating some nodes leads to a conflict with relations (8-9)) then the graph is termed *delta inconsistent*. We will name "expansion" to the action of replacing a target node by a new node at a set of conflicting transitions, as we did with graph (g$_1$). Each expansion removes one inconsistency.

In graph (g$_1$) we see that the inconsistency appears because we differentiate v_1 from v_2, defining $\delta_{i,j}$ in the process. Assigned values "propagate" to other vertices, reaching back to one of the vertices we already defined. Therefore, an inconsistency appears because we tried

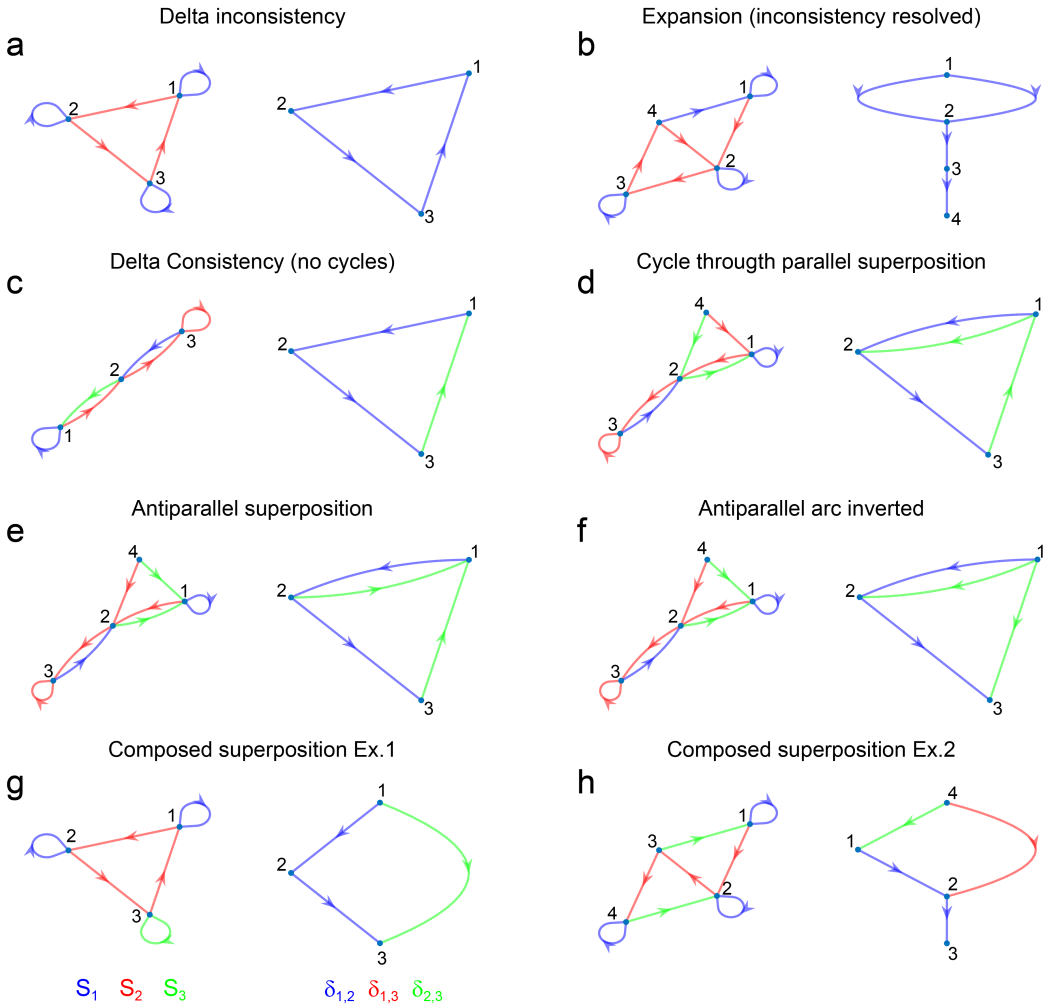

Figure 2: Example graphs G (left of each panel) and their associated graph D (right of each panel). **(a)** A cycle of 3 nodes. If a neuron is active in one vertex and not active in the other vertex, we arrive at a contradiction. This is evidenced by the presence of a cycle in graph D. **(b)** Inconsistency in (a) is resolved by adding vertex v_4 with same targets than v_1 (vertex v_1 is expanded). The associated graph D has no cycles. **(c)** A graph with 3 vertices and 3 stimuli. Graph D has no cycles because it is not possible to flow from a blue arc to a green arc in graph D (there is no superposition of deltas). **(d)** Blue and green arcs in graph D are superimposed, putting them in the same traversable set. Hence, there is a cycle. **(e)** Blue and green arcs in graph D are superimposed but they have opposite directions. These deltas are linked, and arcs labeled with one of these deltas must be inverted. **(f)** Same graph D as in (e) but green arcs have been inverted. All arcs belong to the same traversable set, but no cycles are present. Graph G is left unchanged. **(g)** Arcs blue and green are not directly superimposed but they are linked: defining $\delta_{1,2}$ or $\delta_{2,3}$ also defines $\delta_{1,3}$. **(h)** Composed superposition appears any time two vertices are reachable by more than one path (regardless of arcs direction).

to differentiate two vertices that are part of a certain kind of *cycle*. To make this intuition more precise we define a directed multi digraph $D = (V, A, C, g_s, g_t, l_D)$,where V is the set of vertices, $A \subset \mathbb{N}$ is the set of arcs, $C = \{\delta_{i,j} : (i, j) \in S \times S\}$,the set of labels for the arcs, g_s the function that takes an arc and gives its source vertex, g_t the function that takes an arc and gives its target vertex, and l_D the function that takes an arc and gives its label (Fig. 1d). An arc a from source vertex v_i to target vertex v_j is in A if: $v_i \neq v_j$, $l_D(a) = \delta_{m,n}$ and exist k_1, k_2, v_s such that $f_t(k_1) = v_i$, $f_t(k_2) = v_j$, $f_s(k_1) = f_s(k_2) = v_s$ and $l_G(k_1) = m$, $l_G(k_2) = n$. In other words, two vertices are connected by an arc in D if in graph G the vertices are target vertices of the same source vertex. The label of the arc is $\delta_{m,n}$, the delta associated with the stimuli that lead to each of the connected vertices. The direction of the arc is given by the sign of $\delta_{m,n}$. Self loops ($g_s(a) = g_t(a)$) are not included in D because they represent cases in which $\delta = 0$, which does not constrain assignments to z in any way.

Graph D connects vertices if they are linked by deltas, as we have reasoned above. If this graph has a cycle, then there is at least a pair of nodes that cannot be differentiated without getting a delta inconsistency. Therefore, one way to assure delta consistency is to construct graph D and search for cycles. If a cycle is found, then we must break it by breaking any of the arcs that composes it. Since an arc exists when two nodes are target of the same source node, the arc can be broken by expanding one of the vertices of the arc, at all transitions in G that allow the arc to exist. The expansion breaks the cycle. We have to repeat the process of search and expansion until no cycles are present. The final graph will be delta consistent, and will hold all the information encoded by the original graph (Fig. 1c, d, g).

Navigating the graph D is related to differentiating nodes and propagating z values. Two nodes can be connected by 2 or more arcs of different label. Consider the graph in Fig. 2c:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & - \\ 1 & 2 & - \\ 2 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 3 & 3 \\ 3 & 3 & - \\ & \vdots & \end{pmatrix} \tag{g$_4$}$$

This subset of the graph shows no cycles, and nodes v_1, v_2 and v_3 could be so far differentiated. However in the graph of Fig. 2d:

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & - \\ 1 & 2 & - \\ 2 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 3 & 3 \\ 3 & 3 & - \\ 1 & 4 & - \\ 2 & 4 & 1 \\ 3 & 4 & 2 \\ & \vdots & \end{pmatrix} \tag{g$_5$}$$

node v_1 and v_2 are connected by two arcs with different labels, meaning that differentiating these two nodes will define two different deltas. Therefore, this graph do present a cycle. *Arc label superposition* links the deltas that label the arcs involved, since defining one delta necessarily defines the other one. In turn, arcs with linked deltas can be traversed as if they had the same label. Linked deltas can be represented as a partition P of the set of labels C (i.e. $P = \{\mathscr{C}_i\}_i$, with $\cap_{i,j}(\mathscr{C}_i, \mathscr{C}_j) = \emptyset$, $\cup_i \mathscr{C}_i = C$). Labels $\delta_{i,j}$ and $\delta_{m,n}$ are in an element of the partition $\mathscr{C}_k$ if there are arcs a, b such that $g_s(a) = g_s(b) = v_s$, $g_t(a) = g_t(b) = v_t$ and $l_D(a) = \delta_{i,j}$, $l_D(b) = \delta_{m,n}$. Then, for each element of the partition, there is one *traversable set* of arcs $V_{trav}(\mathscr{C}_i) = \{(v_s, v_t) : l_D(a) \in \mathscr{C}_i, v_s = f_s(a), v_t = f_t(a), a \in A\}$. Before detecting any superposition, deltas are only linked with themselves ($a = b$), and hence $P = C$. As superimposed arcs are detected, deltas are linked by merging the elements of the partitions that contain the linked deltas.

In brief, to find cycles in graph D we choose a traversable set $\mathscr{C}$ from P, and then a source vertex from $\mathscr{C}$. Next, we perform depth first search (DFS), traversing the graph through arcs that are members of $\mathscr{C}$. When a cycle is encountered, a vertex from the cycle is expanded, graph G and D are updated and the process is restarted from a new source vertex in $\mathscr{C}$. When no cycles are encountered withing one traversable set, a new $\mathscr{C}$ is chosen from P. If all traversable sets were explored and no cycles were found, the process stops and returns the final consistent graph G_{cons}.

Arc superposition can be parallel, like in the example above (superimposed arcs have the same direction), but can also be antiparallel (arcs in opposite direction). In this case, differentiating v_1 from v_2 defines two deltas, but the graph cannot be traversed in the same way as when superposition goes in the same direction. Consider the graph in Fig. 2e:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & - \\ 1 & 2 & - \\ 2 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 3 & 2 \\ 2 & 3 & 3 \\ 3 & 3 & - \\ 1 & 4 & - \\ 2 & 4 & 2 \\ 3 & 4 & 1 \\ & \vdots & \end{pmatrix} \tag{g$_6$}$$

which is equal to the previous graph, except for the last two transitions, where the target nodes where switched. In this case, no inconsistency occurs. Yet, a cycle would be detected if the algorithm so far described is executed. This issue is solved by noting that antiparallel superposition implies that deltas have opposite sign, meaning that the graph should be traversed in opposite direction for arcs of superimposed labels. Hence, for each case of antiparallel superposition we flip the direction of all arcs that share the label with one of the antiparallel superimposed arcs (Fig. 2f), with the precaution of only flipping arcs that weren't already flipped.

Finally, we can see that the parallel and antiparallel superpositions shown so far are particular examples of a broader definition of superposition. Consider the graph in Fig. 2g:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & - \\ 1 & 2 & 2 \\ 2 & 2 & 3 \\ 3 & 2 & - \\ 1 & 3 & - \\ 2 & 3 & 1 \\ 3 & 3 & 3 \\ & \vdots & \end{pmatrix} \tag{g$_7$}$$

In this case, differentiating v_1 from v_2 defines $\delta_{1,2}$ but also $\delta_{1,3}$. In the graph of Fig. 2h:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 3 & 1 & - \\ 1 & 2 & 2 \\ 2 & 2 & 3 \\ 3 & 2 & - \\ 1 & 3 & - \\ 2 & 3 & 4 \\ 3 & 3 & 1 \\ 1 & 4 & 4 \\ 2 & 4 & - \\ 3 & 4 & 2 \\ & \vdots & \end{pmatrix} \tag{g$_8$}$$

differentiating v_1 from v_2 defines $\delta_{1,2}$ but also one of the other two deltas: $\delta_{4,2}$ (parallel to $\delta_{1,2}$) or $\delta_{4,1}$ (antiparallel to $\delta_{1,2}$). It is clear that superposition can occur between two arcs that do not share their source and target vertices. We say it is a *composed superposition*. With this in mind we are able to propose a more general way of detecting arc label superposition. We first define a *complete path* as a path composed of arcs that are within a traversable set, which may or may not be a cycle, but has the property that the first node in the path is never a target node in the traversable set, and the last node is never a source node in the traversable set. Then, if it is possible to go from one node of the complete path to another node of the complete path, traversing through arcs contained in any of the *remaining* traversable sets (without taking their direction into consideration), then one of the labels of the arcs that connect the two nodes is superimposed to the arcs of the path. The superposition is parallel, if the superimposed arc was traversed in the direction of the arc, or antiparallel otherwise. Once the superposition is detected, elements of the partition P and the associated traversable sets are merged to account for the new linked deltas. Note that this way of detecting composed superposition will also detect simple parallel and antiparallel superpositions as the ones in examples g_5 and g_6. In gFTP, we start by finding the partition P and traversable sets as defined above. Then, we find all the complete paths to detect the composed superpositions as described above. Partition P and traversable sets are modified according to the presence of parallel and antiparallel superposition. For antiparallel arcs, we invert the direction of all the arcs that share the label with one arc of the arcs of the antiparallel pair of arcs. Arcs that were flipped in their direction are not flipped again. Finally, we merge elements of P and traversable sets according to the superpositions encountered (see the Appendix for pseudocode of the gFTP algorithm and its subfunctions).

Construction of $\mathbf{Z}_s$ and $\mathbf{Z}_t$

Matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$ are generated in a neuron-by-neuron basis, i.e. the neuron activation states $\mathbf{z}_{neu} \in \{1,0\}^{N_{tran}}$ are defined by defining its output after each transition ($\mathbf{z}_{neu}$ is a column in matrix $\mathbf{Z}_t$, of number of elements equal to the number of transitions N_{tran}). Finding $\mathbf{z}_{neu}$ is a constrained satisfaction problem, which we solve with a backtracking algorithm, where there is one binary decision for each vertex, and the constrain to satisfy is delta consistency. Values 1 or 0 are assigned to vertices, and if delta consistency is sustained, we search for vertices that are obliged to have a certain value, either 1 or 0, such that delta consistency is not violated, i.e. assigned values are "propagated" to other values, according to deltas defined so far (for example, if $\delta_{ij} = u_1 - u_2 > 0$ and $z_1 = 0$, then z_2 must be 0 in order to sustain delta consistency). Value propagation using delta consistency helps in reducing the number of combinations to try. To reduce it further, vertices are explored in decreasing order of their in-degree in graph G_{cons}. This follows from the assumption that vertices that appear more times as targets in G have more instances to satisfy a given delta. This should make their assignment more difficult and critical to following assignments, and thus they should be assigned first.

To ensure that all vertices are differentiated, we define neurons so that their output is different for at least a pair of vertices. Thus, we choose two vertices (v_1 and v_2) that are going to be differentiated. First, the algorithm assigns value 1 to vertex v_1 and 0 to v_2. Next, values 1 or 0 are assigned to each one of the remaining vertices, one at a time, following a list L_u of undefined vertices sorted in decreasing in-degree order. The first element in this list is set as the active element, the one that is going to be defined. Any node that is set as active is registered in a list of active elements L_a. Each new assignment may define a new delta, so delta consistency must by checked. If consistency is sustained, new node assignments are made as necessary, according to the z and delta values defined so far (propagation of values according to delta consistency). Then, consistency is checked again. If sustained, the first non defined vertex in L_u is set as the active element. The process continues with the new active element until all vertices have z value. If delta consistency is violated, another z value is tried for the active element (1 if 0 was already tried, 0 if 1 was already tried). If neither of the assignments was successful, the algorithm makes a step back by searching for the last successfully assigned active element in L_a, making it active, and erasing from the list all elements after it. All vertices that were defined after the active element are set as undefined. Assigned values are removed from the list of viable values for that vertex, but all values are restored after a step back, for all the following nodes in L_u, after the active one. If all z values were tried for the first vertex in L_u, without success, then the algorithm stops with no $\mathbf{z}$ as output (there is no vertex left to step back), and transition graph is delta inconsistent. A graph that was previously expanded to obtain a consistent graph G_{cons} should always lead to a fully assigned $\mathbf{z}_{neu}$ for any pair of differentiated nodes.

Once we get vector $\mathbf{z}_{neu}$, we identify all vertices that were differentiated (which can be more than just v_1 and v_2), and remove them from the list of pairs of vertices to differentiate. If

all pairs of vertices were already differentiated, then each node has one vector $\mathbf{z}_v$ that encode that node, and constitutes the rows of matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$. These two matrices define N_{neu} linear separation problems (one for each neuron), where the N_{tran} row vectors in $\mathbf{Z}_s$ are the points to separate, and the N_{tran} ones and zeros of each column in $\mathbf{Z}_t$ are the classes. For each neuron we find its input weights by fitting a simple perceptron through the accelerated perceptron learning algorithm (algorithm No. 2 in [14]). If at least one neuron is encountered for which the separation problem cannot be perfectly solved, we assume that at least one linear combination in $\mathbf{Z}_s$ cannot be reproduced in $\mathbf{U}$. In this case, new matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$ are generated and concatenated to the previous ones, to increase the number of neurons and reduce the number of linear combinations in $\mathbf{Z}_s$. This process is repeated until each neuron can be perfectly solved.

Overview of the gFTP algorithm

1. Graph D is constructed from graph G
2. Arc superpositions are detected and traversable sets are updated
3. Graph D is traversed through DFS.
4. If a cycle is found, one of its nodes is expanded
5. Points 1 to 4 are repeated until no cycles are present
6. Matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$ are constructed
7. Matrices $\mathbf{W}_y$ and $\mathbf{W}_r$ are obtained with the accelerated perceptron algorithm
8. If point 7 fails, new matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$ are constructed and concatenated to the previous ones
9. Points 7 and 8 are repeated until point 7 is successful

3 Results

We employed gFTP to construct networks from three kinds of transition graphs: random transition graphs (Fig. 3a), transition graphs that encode position in 2-dimensional space (Fig. 3b), and transition graphs that instantiate discrete attractor dynamics (Fig. 3c).

For random transition graphs, we set a number of vertices and stimuli. For each vertex, we defined the target vertices for each stimulus. Vertices were numbered, and target vertices were taken at random, with replacement, from the list of vertices that goes from 2 vertices before the source vertex, to 2 vertices after the source vertex, avoiding self-targeting. Therefore, transitions were at random, but constrained to a vicinity around the source. For the second type of transition graph, we defined a discrete 2-dimensional space, square in shape, of a certain integer side

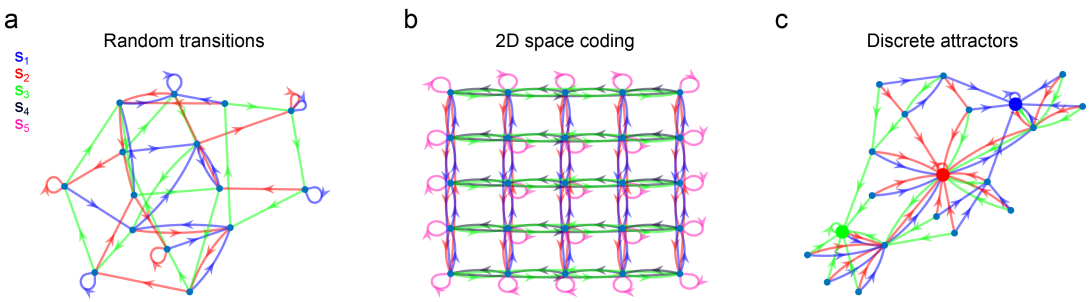

Figure 3: Examples of the transition graphs evaluated (before expansion) **(a)** Random graph with 15 vertices and 3 stimuli. A source vertex i has target vertices taken at random from $i-2$ to $i+2$. **(b)** Transition graph of a network that encodes 25 discrete positions in a 2-dimensional square space, of side length = 5. There are five stimuli, associated with displacement to the right (green), left (gray), up (blue), down (red), and a fifth stimuli (pink) associated to stillness. **(c)** Transition graph with 3 vertices acting as attractors (depicted with bigger markers). Each stimulus leads to one specific attractor through a short-distance path. Graph shown is before completing vertices without source.

length. There is one vertex for each discrete position in this space (i.e. 9 positions and vertices for a 3x3 square arena). Parallel sides were glued together, forming a torus. There are 5 stimuli, 4 for vertical or horizontal displacements, and a 5th for stillness. This graphs instantiate a discretized version of the dynamic of a continuous attractor model, thoroughly studied in the context of spacial coding [11]. Finally, for the third transition graph we defined a number of vertices, and a number of stimuli, which is the same than the number of discrete attractors (each stimulus has an associated attractor vertex, and that stimulus leads to its associated attractor). Then, we embedded all vertices in a 2-dimensional space, and defined a graph in which each vertex is connected to its closest 6 vertices, measured in euclidean distance. Finally, we found the shortest path from each vertex to each of the attractor vertices. We construct graph G by making each stimulus trigger a transition to the first vertex in the shortest path pointing to the attractor associated with that stimulus. Some vertices might end up being source without being target. For each one of these sourceless vertices, one extra vertex is introduced, that goes to the sourceless vertex through one stimulus, and to itself through the remaining stimuli.

We measured computing wall-clock execution time of the algorithm for each kind of transition graph. The complexity of the graph was adjusted by setting the number of vertices or stimuli in random graphs, the side-length of the arena in 2-dimensional space encoding graphs, or the number of vertices in the case of discrete attractor graphs. We separately measured the time required to obtain a consistent graph (consistency time, Fig. 4) and the time to construct $\mathbf{Z}_s$, $\mathbf{Z_t}$ and $\mathbf{W}$ matrices (construction time, Fig. 5). Consistency time was well fitted by a power function and not an exponential (Fig. 6), suggesting polynomial time complexity to obtain a realizable graph from a starting random graph. The number of vertices in the consistent graphs was higher than in the initial graphs (except for the discrete attractors graphs), and the relation between this two numbers was almost linear. Execution times differed considerately between

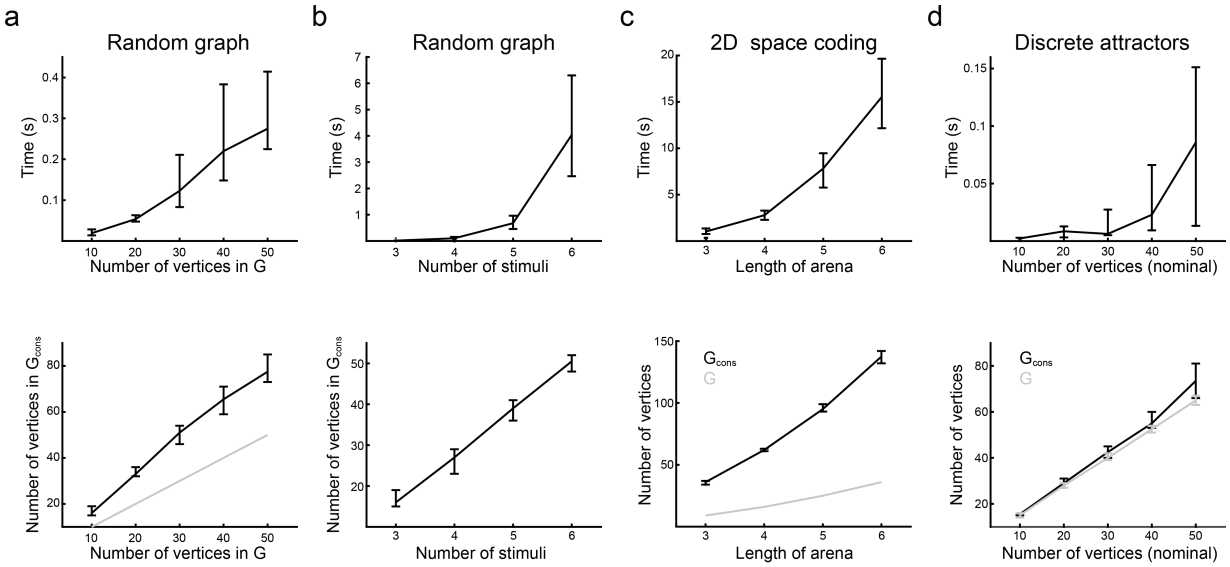

Figure 4: Time performance during consistency assessment. **(a-d)** Execution time until a consistent graph is obtained (upper row), and number of vertices (lower row), for the three types of transition graphs tested. G: initial graph, G_{cons}: consistent graph obtained after successive cycle detection and node expansion. **(a)** Random graphs with 3 stimuli and increasing number of vertices. The gray line is the identity function. **(b)** Random graphs with 10 vertices and increasing number of stimuli. **(c)** 2-dimensioanl space coding graphs, with 5 stimuli and (length of arena)^2 number of vertices (gray line). **(d)** Discrete attractor graphs, with 3 attractors, one stimulus per attractor that leads to it, and an increasing number of vertices. The x axis shows (nominal) number of vertices in the graph (before adding new source vertices to sourceless vertices). The y axis shows the final number of vertices in G after correcting for sourceless vertices, and the final number of vertices in G_{cons}. Plots depict median values. Error bars depict the 25th and 75th percentiles, computed over 30 networks for each condition.

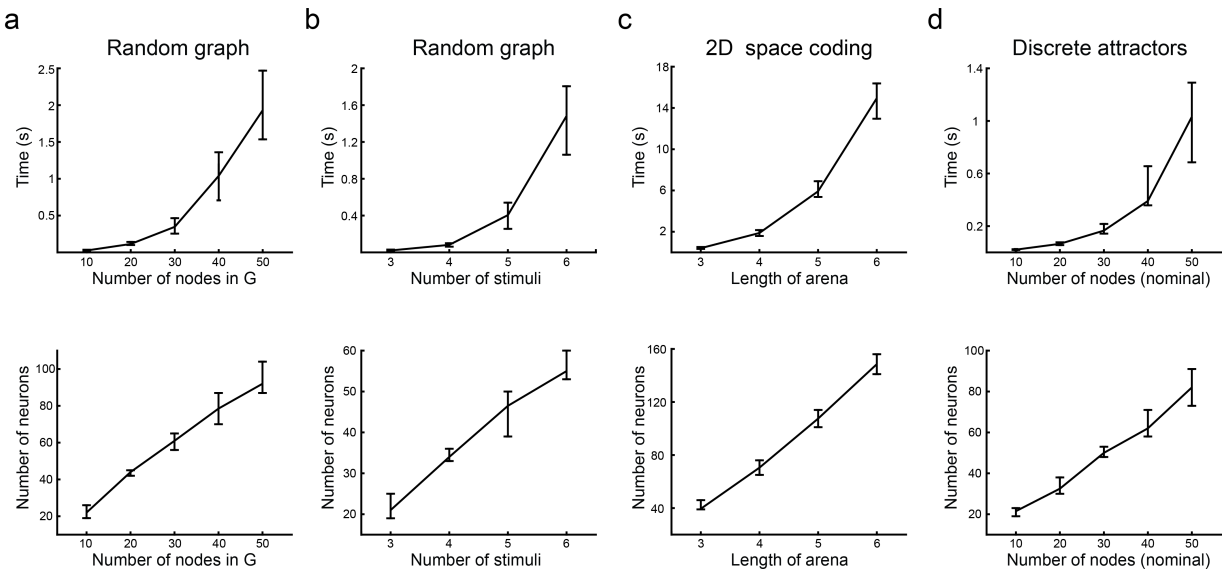

Figure 5: Time performance during network construction. **(a-d)** Execution times until successful construction of matrices $\mathbf{Z}_s$, $\mathbf{Z}_t$, $\mathbf{W}_y$ and $\mathbf{W}_r$ (upper row), and number of neurons in the retrieved network (lower row) for the graphs analyzed in Fig. 4. Plot depict median values. Error bars depict the 25th and 75th percentiles.

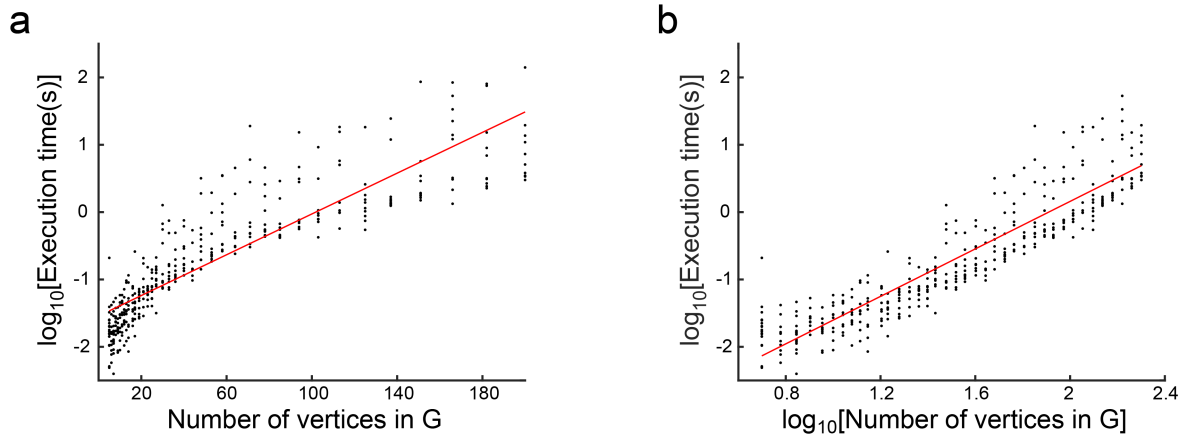

Figure 6: Polynomial execution time to attain delta consistency. **(a)** Execution time until consistency (T_c) as a function of number of vertices (N_v) in G (before expansion), is not well fitted by an exponential: $\log_{10}T_c(N_v) = aN_v + \log_{10}b$, $a = 0.015, \mathrm{CI} = (0.014, 0016)$, $\log_{10}b = -1.54$, $\mathrm{CI} = (-1.60, -1.48)$, d.f. $= 398$, $\mathrm{R-squared} = 0.77$. **(b)** Execution time is well fitted by a power function: $\log_{10}T_c(N_v) = a\log_{10}T_c(N_v) + \log_{10}b$, $a = 1.76$, $\mathrm{CI} = (1.68, 1.84)$, $\log_{10}b = -3.37$, $CI = (-3.49, -3.24)$, $\mathrm{R-squared} = 0.82$, d.f $= 398$. Execution times were computed for random graphs, with 3 stimuli and between 5 and 200 vertices, in logarithmic scale. A total of 10 graphs for each N_v value were constructed.

graph types. Lower consistency times were found for discrete attractor graphs, followed by random graphs with 3 stimuli, random graphs with 6 stimuli, and 2-dimensional space coding graphs. Neither consistency nor construction times were fully explained by the number of nodes in the final graphs, suggesting that the shape of the graphs was an important factor in determining the number of steps necessary for making the graph consistent, and for constructing the network. The number of neurons in the constructed network grew linearly with graph size, and was close to the number of nodes in the final graph. This implies that, in most cases, 1 neuron per vertex was enough, since gFTP adds neurons to the network in multiples of the number of vertices.

4 Discussion

We have introduced an algorithm that takes a target dynamic as input and returns the connectivity matrix of a neural network that follows the target dynamic. This approach departs from other model building approaches in several ways. In bottom up approaches, a model is a specific formalization of a series of proposed mechanisms. Its dynamic is a complex emergent of these mechanisms, and in most cases it cannot be specified in advance. Although it is possible, for some models, to prove analytically some of their properties (like in [15, 16]), this is mostly done at the cost of many simplifications. Complex dynamics that can instantiate specific algorithms are difficult for analytic treatment. On the other hand, normative approaches constrain dynamics to the subsets of dynamics that minimize the loss function, but neural network models

have enough degrees of freedom to find very disparate solutions. While a multiplicity of solutions is appealing in itself, as a way of exploring novel dynamics, it also makes it problematic to go from specific solutions to general conclusion regarding mechanisms. In this sense, our algorithm brings a new angle to the way we build neural network as models of brain function. The gFTP algorithm starts by defining the dynamic that will be unfolded by the network, specified as a transition graph in which vertices are network activation states, and arcs are transitions between states, triggered by stimuli. Then, the algorithm constructs an auxiliary graph (the D graph). If this graph has a cycle, then the transition graph cannot be realizable by a neural network. The algorithm proceeds by expanding the original transition graph, adding vertices that break cycles in graph D, without loosing any of the information encoded in each network state. Eventually, all cycles are removed, and the final transition graph is realizable. Thus, gFTP provides a high level of control over the network dynamics. In turn, this higher control can be exploited in several ways:

1. We can, for instance, construct networks which exhibit dynamics that have been proposed theoretically, or found experimentally, like the discrete attractor dynamics proposed to subserve working memory, or the toroid dynamic proposed to subserve spacial coding. Also, we can define dynamics by means of the algorithms that it instantiates, either at the level of cognition or behaviour. This aspect is particularly specific to our approach, since normative methods are not well suited for specifying exactly how the network solves a task. This makes gFTP particularly useful for studying the interdependencies between dynamic and the algorithms it implements, the information encoded in the activation states, and network connectivity. We could define different algorithms that solve the same task and then compare the expanded graphs, or the networks obtained, and draw conclusion about how the differences in the algorithms translate into differences in dynamics or network connectivity. If it happens that gFTP finds a graph to be inconsistent, then it means that the instantiated dynamic cannot, as such, be realizable in a neural network. This fact is in itself useful, because we can study what features of the dynamic are essential to a given algorithm, and what features are there because of the constrains imposed by the fact that dynamic has to be carried out by the neuronal machinery.

2. For each transition graph that is inconsistent there are many different ways to expand it into a consistent one. We can then run the algorithm multiple times, sampling from the distribution of expanded graphs that come from the same initial graph, and study how different the expanded graphs can be.

3. Once we have a consistent graph, we can generate matrices $\mathbf{Z}$ in a way that they all match some target activation statistics (e.g. same mean firing rate and correlation). Then, we can assess how the dynamics/algorithms constrain the variability of those statistics.

4. We can assess how dynamics, or firing statistics impact on network connectivity. For each dynamic, and matrix $\mathbf{Z}$, we can generate a sample of synaptic matrices that instantiate the same dynamic with the same activation states, and assess how dynamics or firing statistics impacts on network connectivity.

In sum, the gFTP algorithm provides a high level of control over key aspects of analysis, like network function, dynamics and firing statistics, which in turn helps to disentangle their effects by carrying out controlled numerical experiments.

Our results with random graphs showed a time complexity well fitted by a power function, with exponent ~1.76, for the operations of consistency assessment and expansion. On the other hand, to construct $\mathbf{Z}_t$, our current implementation explores the tree of possible assignments for each neuron in each network state, accelerating the process by computing deltas and filling assignments that are completely specified by the deltas and z values already defined. We also used the heuristic of assigning z value first to vertices with higher degree in graph G, for these are expected to be part of cycles with higher probability. This heuristic helped in reducing execution time during construction of matrices $\mathbf{Z}_t$. Yet, the algorithm would benefit from other ways of assigning z values. Studying the structure of graph D and cycle formation will certainly help. Also, optimization methods like genetic algorithms could be good options to evaluate.

Once matrices $\mathbf{Z}_t$ were constructed, we found matrices $\mathbf{W}$ by employing the accelerated perceptron algorithm. We decided to add neurons to the network in multiples of the number of vertices in the transition graph. The reason for this criterion is that, for matrix $\mathbf{W}$ to exist, matrix $\mathbf{U}$ must have the same linear combinations that matrix $\mathbf{C}$, and $\mathrm{H}(\mathbf{U}) = \mathbf{Z}_t$. Not all matrices $\mathbf{Z}_s$ and $\mathbf{Z}_t$ allow the existence of a matrix $\mathbf{U}$ that fulfill this two conditions. Yet, the probability of finding one viable matrix $\mathbf{U}$ is expected to be higher if the number of linear combinations in $\mathbf{C}$ is low. This can be achieved by adding neurons to the network, with random values of z, since the probability of a linear combination between rows of $\mathbf{Z}_s$ is expected to drop with the number of its columns.. Adding neurons in multiples is also convenient in that the number of calls to the accelerated perceptron is reduced: too few neurons would certainly not lead to a solvable system, and the accelerated perceptron will have to reach the maximum of allowed iterations in order to stop and try with a higher number of neurons, thus increasing execution time. In our experiments, a number of neurons approximately matching the number of vertices were enougth to define a solvable system of linear separation problems.

One limitation of our method is that it is restricted to a simple neuron model. The binary neuron is not a proper model of membrane potential, because preactivations only depend on the neuron inputs, and not on preactivations in the previous time step. Therefore, the model cannot give spike trains with realistic time intervals. This network model is suitable, nonetheless, to model broad dependences between dynamics, firing statistics, and connectivity, that does not heavily depend on membrane potential dynamics[17, 18][17, 18]. Another limitation is that for constructing matrix $\mathbf{Z}_t$ we rely on a backtracking approach, which can be inefficient. Better methods could be developed in the future for this task, as we have discussed above. Also, matrix $\mathbf{Z}_t$ could be generated only to be close to consistent, and $\mathbf{W}$ only leading to an approximation of $\mathbf{Z}_t$. In this case, the obtained network will not follow the target dynamic exactly. Nevertheless, it is expected that, with enough neurons and redundancy in $\mathbf{Z}_t$, the approximate network will follow a dynamic that is close enough to the target. It could also be possible to use the network

found with gFTP as a good starting point for further optimization with another algorithm.

As we have already stated, the gFTP algorithm is suitable for studying the interdependence between network parameters and its function, by generating a sample of networks that share a given target property. Recent works have address similar goals. In Brennan *et al.*[19] a method, termed LOOPER is introduced, in which a Markov process is constructed from recorded neural activity, with transition probabilities modelled as an asymmetric diffusion process, such that the global dynamic respects convergences and divergences observed in the data. The method relates to ours in that it allows to construct a model of an arbitrary dynamical system. This is accomplish in our algorithm in the transition graph, in which convergences and divergences are what differentiate one graph from another. LOOPER has the advantage of finding a model directly from data, while the use of a binary neuron model in gFTP makes it more difficult to translate recorded firing activity into a transition graph. However, and in constrast to our method, LOOPER never constructs a neural network model, so there is no explicit way of linking the low dimensional dynamic recovered in the model with any underlying network connectivity. On the other hand, in the work of Brittner *et al.*[20] and Gonçalvez *et al.*[21] a deep neural network is used to approximate a probability density function over parameters of a target neural network, such that a target mean over an emergent property is satisfied. The method proposed by Brittner *et al.*, called Emergent Property Inference (EPI), is very versatile, in that the target model only has to be differentiable, but it does not give control over specific dynamics, because different dynamics can show the same emergent property. In Gonçalvez *et al.* the method called Sequential Neural Posterior Estimation (SNPE) is introduced, in which a distribution over parameters is fitted to the observed data, or data summary features computed from the data. In this case the distribution is fitted through data collected by simulation, which gives the possibility of assessing non differentiable models. Tractability of both methods is an issue: when tested to find distributions of low rank recurrent networks with up to 1000 parameters, constrained to have synaptic weight matrices of given eigenvalue, SNPE failed to converge with the biggest networks, while for EPI it took up to 5 hours of training. While impressive, this is a fitting task with constrains on the synaptic weight matrix, which is expected to be easier than with constrains on network activity. Although in gFTP we generate one network at a time, each generation is fast, hence a big population of networks can be generated. Also, it is guaranteed that the network will have the desired dynamic. Beside, gFTP can always be adapted to give networks that only approximate the target dynamic, as discussed above.

In conclusion, we have introduced a systematic way of constructing transition graphs that are realizable by a neural network of binary neurons, allowing to test hypothesis regarding function, dynamics and connectivity in ways not matched by other methods. We believe the gFTP algorithm is going to be a valuable addition to the toolbox of the theorist.

Acknowledgements

This research was supported by the Consejo Nacional de Investigaciones Científicas y Técnicas, Grant No. 11220200102316CO, Agencia Nacional de Promoción de la Investigación, el Desarrollo Tecnológico y la Innovación, Grant No. PICT-2021-I-A- 00957, and Universidad de Buenos Aires.

References

[1] B. A. Richards, T. P. Lillicrap, P. Beaudoin, Y. Bengio, R. Bogacz, A. Christensen, C. Clopath, R. P. Costa, A. de Berker, S. Ganguli, C. J. Gillon, D. Hafner, A. Kepecs, N. Kriegeskorte, P. Latham, G. W. Lindsay, K. D. Miller, R. Naud, C. C. Pack, P. Poirazi, P. Roelfsema, J. Sacramento, A. Saxe, B. Scellier, A. C. Schapiro, W. Senn, G. Wayne, D. Yamins, F. Zenke, J. Zylberberg, D. Therien, and K. P. Kording, "A deep learning framework for neuroscience," *Nature Neuroscience*, vol. 22, no. 11, pp. 1761–1770, 2019.

[2] G. R. Yang, M. R. Joglekar, H. F. Song, W. T. Newsome, and X. J. Wang, "Task representations in neural networks trained to perform many cognitive tasks," *Nature Neuroscience*, vol. 22, no. 2, pp. 297–306, 2019. [Online]. Available: http://dx.doi.org/10.1038/s41593-018-0310-2

[3] A. Dubreuil, A. Valente, M. Beiran, F. Mastrogiuseppe, and S. Ostojic, "The role of population structure in computations through neural dynamics," *Nature Neuroscience*, vol. 25, no. 6, pp. 783–794, 2022.

[4] A. E. Orhan and W. J. Ma, "A diverse range of factors affect the nature of neural representations underlying short-term memory," *Nature Neuroscience*, vol. 22, no. 2, pp. 275–283, 2019.

[5] S. Vyas, M. D. Golub, D. Sussillo, and K. V. Shenoy, "Computation Through Neural Population Dynamics," pp. 249–275, 2020.

[6] M. R. Whiteway and D. A. Butts, "The quest for interpretable models of neural population activity," *Current Opinion in Neurobiology*, vol. 58, pp. 86–93, 2019. [Online]. Available: https://doi.org/10.1016/j.conb.2019.07.004

[7] J. A. Frank, M. J. Antonini, and P. Anikeeva, "Next-generation interfaces for studying neural function," *Nature Biotechnology*, vol. 37, no. 9, pp. 1013–1023, 2019. [Online]. Available: http://dx.doi.org/10.1038/s41587-019-0198-8

[8] H. K. Inagaki, L. Fontolan, S. Romani, and K. Svoboda, "Discrete attractor dynamics underlies persistent activity in the frontal cortex," *Nature*, vol. 566, no. 7743, pp. 212–217, 2019. [Online]. Available: http://dx.doi.org/10.1038/s41586-019-0919-7

[9] N. Brunel, E. N. Sup, and X.-j. Wang, "Effects of Neuromodulation in a Cortical Network Model of Object Working," pp. 63–85, 2001.

[10] R. J. Gardner, E. Hermansen, M. Pachitariu, Y. Burak, N. A. Baas, B. A. Dunn, M. B. Moser, and E. I. Moser, "Toroidal topology of population activity in grid cells," *Nature*, vol. 602, no. 7895, pp. 123–128, 2022.

[11] Y. Burak and I. R. Fiete, "Accurate path integration in continuous attractor network models of grid cells," *PLoS Computational Biology*, vol. 5, no. 2, 2009.

[12] C. D. Brody, R. Romo, and A. Kepecs, "Basic mechanisms for graded persistent activity: Discrete attractors, continuous attractors, and dynamic representations," *Current Opinion in Neurobiology*, vol. 13, no. 2, pp. 204–211, 2003.

[13] C. J. Mininni and B. S. Zanutto, "Probing the structure â function relationship with neural networks constructed by solving a system of linear equations," *Scientific Reports*, vol. 11, no. 1, pp. 1–18, 2021. [Online]. Available: https://doi.org/10.1038/s41598-021-82964-0

[14] G. Wang, R. Hanashiro, E. Guha, and J. Abernethy, "On Accelerated Perceptrons and Beyond," no. 1962, 2022. [Online]. Available: http://arxiv.org/abs/2210.09371

[15] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, no. 8, pp. 2554–2558, 1982.

[16] H. Sompolinsky, A. Crisanti, and H. J. Sommers, "Chaos in random neural networks," *Physical Review Letters*, vol. 61, no. 3, pp. 259–262, 1988.

[17] N. Brunel, "Is cortical connectivity optimized for storing information?" *Nature Neuroscience*, vol. 19, no. 5, pp. 749–755, 2016. [Online]. Available: http://www.nature.com/doifinder/10.1038/nn.4286

[18] A. Alemi, C. Baldassi, N. Brunel, and R. Zecchina, "A Three-Threshold Learning Rule Approaches the Maximal Capacity of Recurrent Neural Networks," *PLoS Computational Biology*, vol. 11, no. 8, pp. 1–23, 2015. [Online]. Available: http://dx.doi.org/10.1371/journal.pcbi.1004439

[19] C. Brennan, A. Aggarwal, R. Pei, D. Sussillo, and A. Proekt, *One dimensional approximations of neuronal dynamics reveal computational strategy*, 2023, vol. 19, no. 1.

[20] S. R. Bittner, A. Palmigiano, A. T. Piet, C. A. Duan, C. D. Brody, K. D. Miller, and J. Cunningham, "Interrogating theoretical models of neural computation with emergent property inference," *eLife*, vol. 10, pp. 1–41, 2021.

[21] P. J. Gonçalves, J. M. Lueckmann, M. Deistler, M. Nonnenmacher, K. Öcal, G. Bassetto, C. Chintaluri, W. F. Podlaski, S. A. Haddad, T. P. Vogels, D. S. Greenberg, and J. H. Macke, "Training deep neural density estimators to identify mechanistic models of neural dynamics," *eLife*, vol. 9, pp. 1–46, 2020.

Appendix

Below we provide pseudocode for the gFTP algorithm and its key subfunctions, together with the following notes:

- The expression $x(1)$ represents the first element of a vector or list, while $x(end)$ is the last element.
- The expression $[x_1 ... x]$ stands for concatenation of $x_1 .. x_n$.
- The function DFS performs Depth First Search. When a cycle is encountered, the program stops and returns the arcs (v_s, v_t) that compose the cycle, and also a function *delta* that maps each arc with the $\delta_{i,j}$ that labels the arc.
- The graph D' is like graph D but with its arcs flipped when there were antiparallel superimposed arcs. It is equal to D if no antiparallel superpositions were found.
- The set V_{dif} is the set of all pairs of vertices that are differentiated by the neuron (their assigned z values are different).
- The difference between two sets A and B is written $A - B$.
- In Algorithm 1: gFTP, *max_iter* is the maximum number of iterations executed by the accelerated perceptron algorithm while searching for a solution. It was set to $1000N_{tran}$.
- In Algorithm 3: Find_Superposition, when two sets in P are merged, the sets are removed from P, and a new set is added, that is the union of the two merged sets.
- In Algorithm 4: Make_Z_Target, the function z_{valid} takes a vertex as argument and returns a vector, specific for that vertex, which can be (1,0) or (0,1), chosen at random from a Bernoulli distribution. These vectors are defined once at the beginning of the algorithm, and are kept constant thereafter. The variable *firing_rate* is the parameter of the Bernoulli distribution. It was set to 0.5 in all experiments.

Algorithms were implemented in MATLAB R2021a (The MathWorks, Inc., Natick, MA, USA), running on a notebook computer with Intel^{TM} i7-8550U CPU and 32 GB RAM.

Algorithm 1 gFTP

Input: graph G
Output: graph G_{cons}, $\mathbf{Y}$, $\mathbf{Z}_s$, $\mathbf{Z}_t$, $\mathbf{W}$

$G_{cons} \leftarrow$**Make_Consistent**(G)
$e \leftarrow 1$
$\mathbf{Z}_t \leftarrow [\,]$
Build $\mathbf{Y}$ by concatenating the identity matrix according to G_{cons}
while $e > 0$ **do**:
    $pair_list \leftarrow \{(v_i, v_j)\}_{j>i}$
    while $pair_list \neq \emptyset$ **do**:
        choose (v_i, v_j) from $pair_list$
        $\mathbf{z}, V_{dif} \leftarrow$ **Make_Z_Target**(G_{cons}, v_i, v_j)
        $\mathbf{Z}_t \leftarrow [\mathbf{Z}_t, \mathbf{z}]$
        remove V_{dif} from $pair_list$
    build $\mathbf{Z}_s$ from $\mathbf{Z}_t$ and G_{cons}
    $\mathbf{W}, e \leftarrow$ **Make_W**$(\mathbf{Y}, \mathbf{Z}_s, \mathbf{Z}_t, max_iter)$

Algorithm 2 Make_Consistent

Input: G
Output: G

```
D ← Make_D(G)
D′, P, V_trav ← Find_Superposition(D)
choose C ∈ P
choose (v_s, v_t) ∈ C
D_rest ← D′ restricted to V_tran(C)
while D′ ≠ ∅ do:
    cycle, delta ← DFS(D_rest) starting in v_s
    If cycle ≠ ∅:
        G ← Expand_Node(G, cycle, delta)
        D ← Make_D(G)
        D′, P, V_trav ← Find_Superposition(D)
        choose C ∈ P
        D_rest ← D′ restricted to V_trav(C)
    else:
        D′ ← D′ − D_rest
        P ← P − C
        choose C ∈ P
        choose (v_s, v_t) ∈ C
        D_rest ← D′ restricted to V_tran(C)
```

Algorithm 3 Find_Superposition

Input: D

Output: D_{output}, P, V_{trav}

```
C_used ← { }
D_output ← D
while D ≠ ∅ do:
    compute P, V_trav from D
    choose 𝒞 ∈ P
    C_comp ← C − 𝒞
    choose p, a complete path in V_trav(𝒞)
    D_und ←undirected version of D
    for v_i, v_j ∈ p, i ≠ j do:
        find path p_2 from v_i to v_j in D_und, through arcs in C_comp
        choose arc a in p_2
        find C_a ∈ P, such that the label of a is in C_a
        merge C_a with C in P
        update V_trav according to updated P
        if a has opposite direction in p_2 than in D and label(a) ∉ C_used:
            flip the direction of all arcs in D_output that have labels in C_a
            C_used ← C_used ∪ C_a
    remove p from D
```

Algorithm 4 Make_Z_Target

Input: G, v_1, v_2
Output: $\mathbf{z}$, V_{dif}

$(\mathbf{z}(t_1), \mathbf{z}(t_2)) \leftarrow (1,0)$ with $(f_t(t_1), f_t(t_2)) = (v_1, v_2)$
def: $z_{valid} : v \rightarrow [x, 1-x]$, $x \sim$ Be(*firing_rate*)
$z^0_{valid} \leftarrow z_{valid}$
$V \leftarrow$vertices in G, sorted in order of decreasing in-degree
$V_{def} \leftarrow [v_1, v_2]$
$L_u \leftarrow V - V_{def}$
$L_a \leftarrow L_u(1)$
$\mathbf{z} \leftarrow$empty vector of dimension N_{tran}
$\mathbf{z}_{backup} \leftarrow []$
while $L_a \neq \emptyset$ **and** $L_u \neq \emptyset$ **do**:
    $v \leftarrow L_a(end)$
    $z_1 \leftarrow z_{valid}(v)$
    if $z_1 \neq \emptyset$ **do**:
        $z \leftarrow z_1(1)$
        $\mathbf{z}^0 \leftarrow \mathbf{z}$
        $\mathbf{z}_{backup}(v) \leftarrow \mathbf{z}$
        $\mathbf{z}(t) \leftarrow z$ for transitions t in which v is a target vertex
        remove z from $z_{valid}(v)$
        consistency $\leftarrow$ **Check_Consistency**(D, $\mathbf{z}$)
        if *consistency* = True:
            propagate deltas over $\mathbf{z}$
            consistency $\leftarrow$ **Check_Consistency**(D, $\mathbf{z}$)
            if *consistency* = True:
                $V_{def} \leftarrow [v$ with z value assigned in $\mathbf{z}]$
                $L_u \leftarrow V - V_{def}$
                $L_a \leftarrow [L_a, L_u(1)]$
            else:
                $\mathbf{z} \leftarrow \mathbf{z}^0$
        else:
            $\mathbf{z} \leftarrow \mathbf{z}^0$
    else:
        $L_a \leftarrow L_a - v$
        $\mathbf{z} \leftarrow \mathbf{z}_{backup}(L_a(end))$
        $V_{def} \leftarrow [v$ with z value assigned in $\mathbf{z}]$
        $L_u \leftarrow V - V_{def} - L_a(end)$
        reset z_{valid} to z^0_{valid} for each $v \in L_u$
$V_{dif} \leftarrow \{(v_i, v_j)\}_{ij}$ s.t. $z(v_i) \neq z(v_j)$

Algorithm 5 Expand_Node

Input: G, *cycle, delta*
Output: G

1. choose $e_{cycle} = (v_1, v_2)$, an arc in *cycle*
2. $\delta_{s_1,s_2} \leftarrow delta(e_{cycle})$
3. $v_{replace} \leftarrow e_{cycle}(1)$
4. $v_{new} \leftarrow |V| + 1$
5. **for** all pairs of arcs $a_1 \leftarrow (v_s, v_{t_1})$, $a_2 \leftarrow (v_s, v_{t_2})$ in G **do**:
6. **if** $(v_{t_1}, v_{t_2}) = e_{cycle}$ and $(\text{label}(a_1), \text{label}(a_2)) = (s_1, s_2)$:
7.   delete arc a_2 with label s_2 in G
8.   add arc (v_s, v_{new}) to G
9. **for** each arc $a \leftarrow (v_{source}, v)$ in G with $v_{source} = v_{replace}$ **do**:
10. add to G an arc (v_{new}, v_{target}) with the label of a