UAB
Universitat Autònoma
de Barcelona

Universitat Autònoma de Barcelona

Master's Degree in Modelling for Science and Engineering

Attractors in Protein-Protein Interaction Networks as Content Addressable Memories

Christian Germer Margüenda
Supervisors: María Jinxue García-Pantaleón Tarifa, Jordi Garcia-Ojalvo

Abstract

Hopfield Networks, introduced by John Hopfield in 1982, are complete neural networks –each neuron is connected with all other neurons– that function as content addressable memories (CAM) –memories that search data by content. On the other hand, protein-protein interaction networks (PPINs) are the biological networks that define the interactions between proteins and execute the specific molecular mechanisms that define cell types. In this work, we propose a sparse Biological Hopfield Network using the topological structure of a protein-protein interaction network, with the objective of retrieving the attractors of the biological model and defining cell types as the attractors of the network. The model did not retrieve unique attractors, but rather clusters of attractors, allowing for multiple interpretations. While these findings are inconclusive, this work establishes a foundation for future development.

Contents

List of Figures

1 Introduction

In this work, we propose that different cell types could be identified as attractors in protein-protein interaction networks (PPINs) when modeled as content addressable memories using (CAM) the Hopfield Model. The work is divided into five chapters, whose contents are described in what follows.

In Section 2, we give some biological context as to what cell types are. We also introduce related work that inspired the fundamental concepts of our work, and finally, we introduce our objectives and explain the theoretical basis of PPINs and the Hopfield Model.

Continuing with Section 3, we introduce the data sources used in our model, followed by an explanation of the Hopfield Model. Finally, we introduce the two clustering algorithms used to analyze the obtained results.

In Section 4, we present the experiments done on a Standard Hopfield Network and a Biological Hopfield Network. For the standard network, we perform three tests: two using a trained Hopfield Network, in which we retrieve stored noisy patterns and randomly initialized patterns, and one with a Hopfield Network with random weights, in which we retrieve randomly initialized patterns. For the biological network, two test configurations were performed using randomly initialized patterns: one using randomly initialized weights, and another using network weights defined by the combined score values of the STRING database.

In Section 5, we recall the most relevant results from the work. We discuss the clusters found in the Biological Hopfield Network and how they can be related to cell type attractors.

Finally in Section 6, we propose the mapping of cell type patterns in protein-protein interaction networks.

2 Biological context and recent work

Cell differentiation is the mechanism whereby different sorts of cells arise. There are more than 200 different specialized cell types in a vertebrate body, ranging from epidermis to thyroid epithelium, lymphocyte, or neuron, and each cell type owes its special character to particular proteins coded by the expression of specific genes [1]. Cell differentiation is therefore the transition from one cell type to another, which involves a switch from one pattern of gene expression to another [2].

During development, cellular differentiation is driven by changes in gene expression guided by epigenetic modifications [3]. This process is primarily controlled by transcription factors (TFs), which bind to cis-regulatory elements (CREs) to regulate transcription levels. TFs interact within complex gene regulatory networks (GRNs) (See Fig.1) to establish cell identity, with specific 'Master TFs' exerting a disproportionate impact on lineage choices [3].

The biological foundation of this process begins in the genome, which contains the chromosomes. Within these chromosomes are long strands of DNA containing sections called genes, which hold the instructions to make proteins. While every cell in an organism contains the same genome, its distinct function is determined by the specific genes it expresses and the proteins they encode. This process begins with transcription: a gene is switched on, and the RNA polymerase enzyme attaches to the start of the gene and generates the corresponding messenger

RNA (mRNA). Following transcription, the mRNA exits the cell's nucleus to the cytoplasm, where protein factories, the ribosomes, bind to it. The ribosomes read the genetic code in the mRNA to generate amino acid chains. Subsequently, these chains fold into complex shapes of functional proteins. Ultimately, it is the specific 'proteome', proteome the collection of proteins present at any given time, that dictates a cell's structural and functional capabilities.

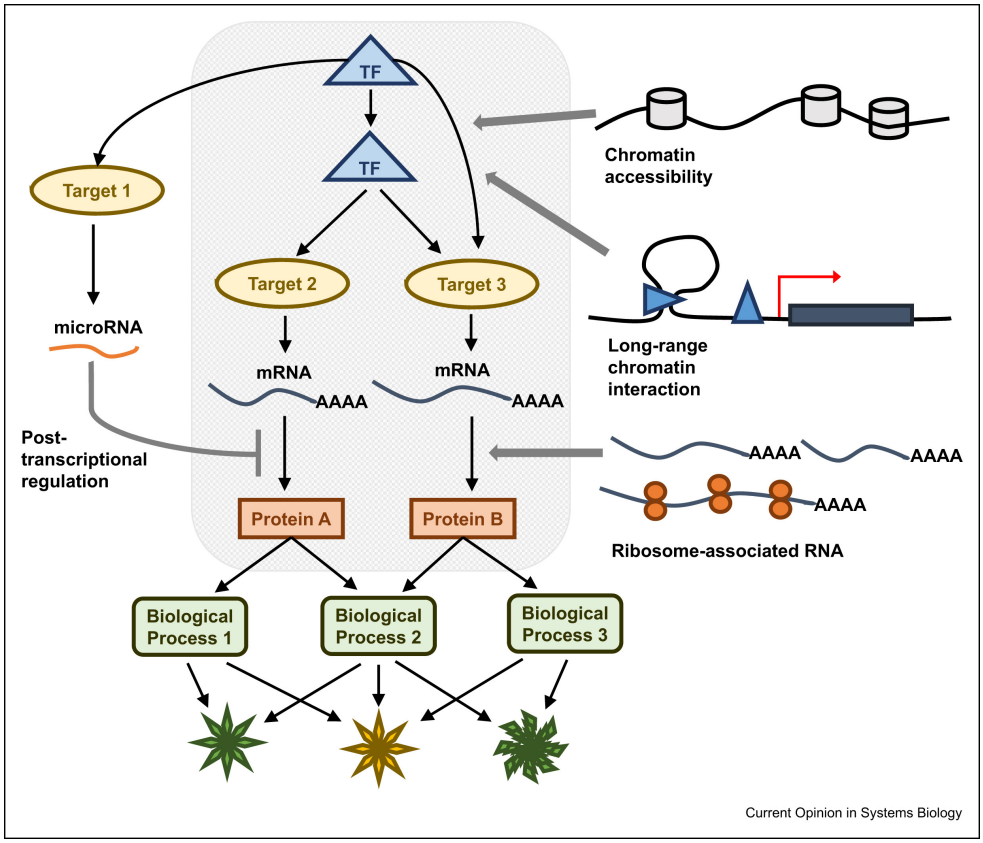


Figure 1: The path from genomic information to phenotypic traits is regulated at multiple levels. A GRN describes the collection of molecules and interactions underlying the biological processes that impact the phenotypes of the plants. The gray area represents a canonical GRN with transcription factor (TF) binding and target gene expression as the center of the network. The black arrow depicts information flow from the genome to phenotypes. Outside of the gray area are new features incorporated into the GRN. Dark gray arrow indicates the steps at which these features affect the GRN [4].

Protein-protein interaction networks are a part of the interactome of a cell and represent the physicall interactions between proteins. The interactome represents the physical connections within a given proteome, meaning that PPINs are ultimately governed by the proteins present in the cell. While GRNs drive the expression and decoding of genes to synthesize the proteome, it is the physical interaction between these synthesized proteins that establishes the interactome. Therefore, GRNs directly dictate the composition of the PPINs, which in turn execute the specific molecular mechanisms that ultimately define and maintain distinct cell types.

2.1 Cell type attractors in Gene regulatory networks

Gene regulatory networks control animal development and regulate the expression of thousands of genes in any given process. They are regulatory codes with the task of specifying the sets of genes that must be expressed in specific spatial and temporal patterns [5]. In the work of Bornholdt and Kauffman, [6] they discuss Boolean networks as the primary method to model GRNs, and how the dynamical attractors of random Boolean networks model cell types, by defining the Boolean network as a directed system with N nodes (genes). Each gene receives inputs from K genes chosen among the N genes. The value of K may vary from gene to gene. This assignment of input defines the connections of the network. By placing an initial state in this network, each gene will assess the values of its K inputs and will determine wether it should be on or off. Overtime the system traces a sequence of states and will keep traversing the state cycle forever. If multiple trajectories end up on the same state cycle, that latter is defined as an attractor of the network. It is through this process that Bornholdt and Kauffman [6] define cell types as different attractors in GRNs and open the door to new interpretations for attractors in biological networks.

2.2 Protein-protein interaction networks

Early biological experiments revealed proteins as the main agents of biological function. As such, proteins ultimately determine the phenotype of all organisms. Since the advent of molecular biology, we have learned that proteins do not function in isolation; rather, their interactions with other proteins and molecules (e.g., DNA, RNA) mediate metabolic and signaling pathways, cellular processes, and organismal systems [7]. Proteins physically associate to form macromolecular structures of varying complexity and heterogeneity. Additionally, proteins interact in pairs to form dimers, multi-protein complexes, or long chains [8].

Contrary to GRNs, which are usually represented as directed graphs, protein-protein interaction networks are commonly represented by undirected graphs [9], which means that a connection of two nodes in the networks goes both directions. For instance, if protein A is connected to protein B, both proteins affect each other's behavior. As has been proven, cellular functions are coordinated activities of many proteins that interact with each other.

By knowing this, we pose the following question: Can cell types be attractors in PPINs? If so, how can we model PPINs and find these cell type attractors?

2.3 The Hopfield Model

Hopfield networks [10], originally developed to study memory retrieval in neural networks, have become versatile tools for modeling diverse biological systems in which function emerges from collective dynamics [11]. These networks are defined as complete undirected graphs (See Fig.2) that follow a simple Hebbian learning rule [12] which proposes that if two interconnected neurons are both ON at the same time, then the weight between them should be increased.

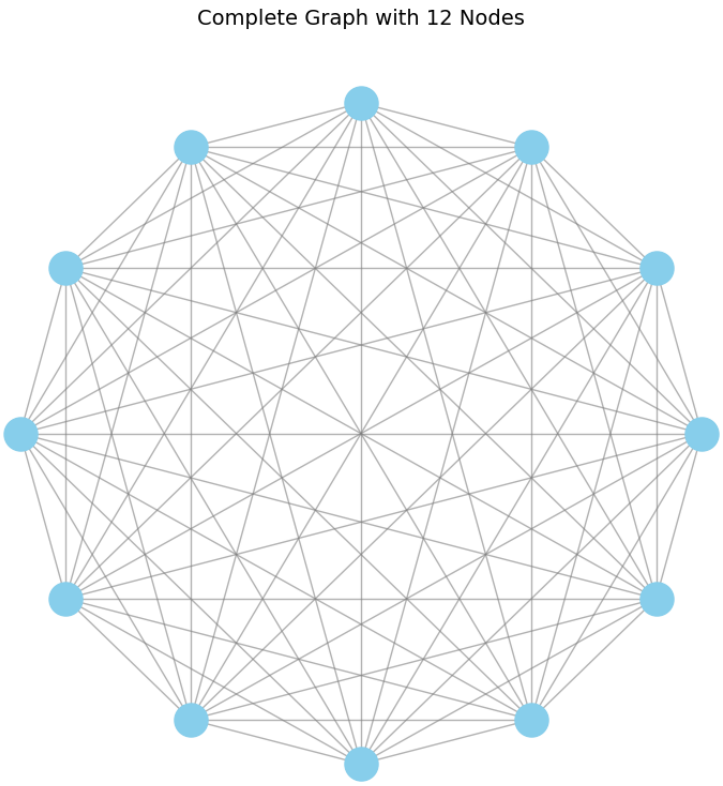


Figure 2: Complete graph with 12 nodes. Each node is connected to every other node, with no self-loops. The connections are undirected (bidirectional), resulting in (N(N-1)/2) edges, where (N) is the number of nodes in the graph.

These networks can 'remember' binary patterns used to train the network by letting the neurons of the network balance themselves out using the weights of their connections as time iterations pass.

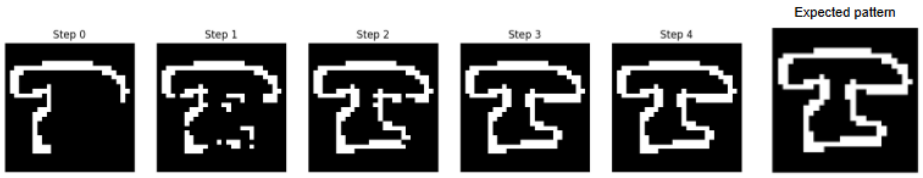


Figure 3: Progression of the states in a Hopfield network when retrieving a stored pattern [13]. In each iteration the network converges the states of the neurons towards the expected pattern.

The memories stored in a Hopfield network can be regarded as attractor states. When the network is provided with an initial state that is sufficiently close to one of these attractors, it will converge to the corresponding stored memory, as long as the initial state falls within the same energy basin associated with that memory (See Fig.3).

The undirected nature of the connections in Hopfield networks suggests that PPINs could operate as Hopfield networks and, consequently, as content-addressable memories. In this framework, a subset of attractor states would correspond to the stored memories, which in our case represent different cell types.

3 Materials and Methods

3.1 Data sources and preprocessing

As a consequence of experimental and bioinformatic approaches providing data about interacting proteins on a genome and proteome-wide scale, several research groups have also made an important effort in designing and setting up databases that include computer-controlled information about these 'interactomes'. One of the most significant public databases of PPINs is STRING [14], a database of predicted functional associations among genes/proteins [8], and our main source of data for this work. In particular, we decided to used the data related to the *Mus Musculus* [15], also known as the 'House Mouse'.

The STRING Database not only provides data on the structure of the PPIN, it also provides a connection weight between proteins, called combined score, which is a confidence value that determines the likelihood of the connection between two proteins to exist. It is calculated by using evidence scores of different evidence channels (experimental, curated databases, gene neighborhood, etc.) to calculate it [14].

After extracting the network data from STRING by setting the minimal combined score to 700, we were left with a network of roughly 10 thousand nodes and 41 thousand edges.

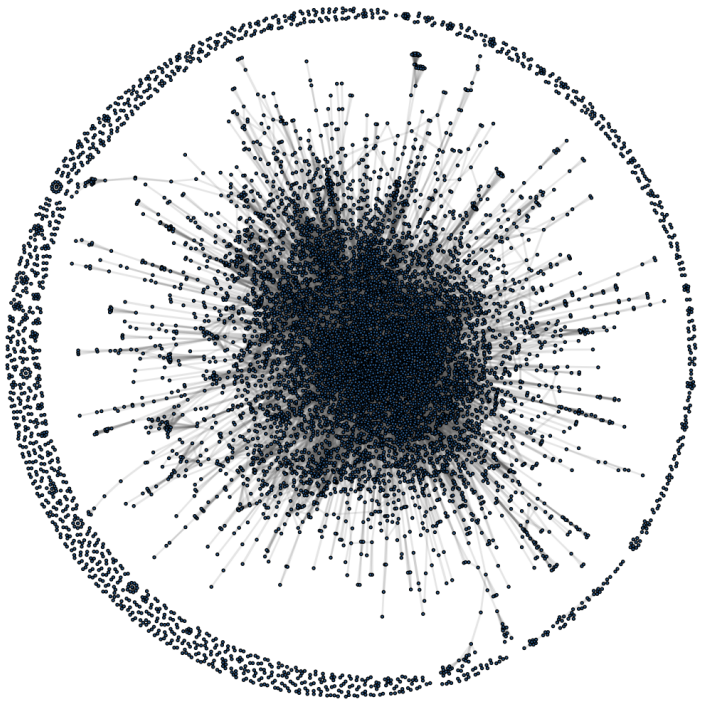


Figure 4: Full network of the Mus musculus [15] after processing all protein connections with a combined score greater or equal to 700. The entire network is conformed of 10.082 nodes and 40.783 edges.

The goal of this project is to compare the biological PPIN to a Hopfield network, however, using the entirety of the Mus musculus PPIN would produce a Hopfield network of 51 million edges. Processing data at this scale proves to be not feasible with the available computational resources.Therefore, to reduce the dimensionality of the network, we decided to search for communities inside the PPIN. For the community search, we used the Leiden algorithm of the leidenalg python library [16]. The Leiden algorithm is an improved version of the Louvain algorithm [17] specifically designed to fix a major flaw of the Louvain algorithm, namely, that Louvain can find communities that are internally disconnected or poorly connected. The Leiden algorithm consists of three phases: (1) local moving of nodes, (2) refinement of the partition and (3) aggregation of the network based on the refined partition, using the non-refined partition to create an initial partition for the aggregate network [16]. By following these three phases, the Leiden algorithm guaranties that all the communities found are well connected.

In Figure 5 we can see the top 5 biggest communities the Leiden algorithm has found in the original PPIN. By looking at the diagonal of the plot, we observe that each community has a high number of connections between nodes inside the same community. Futhermore, these communities tend to have a relatively lower number of connections between communities in comparison to the connections inside each community.

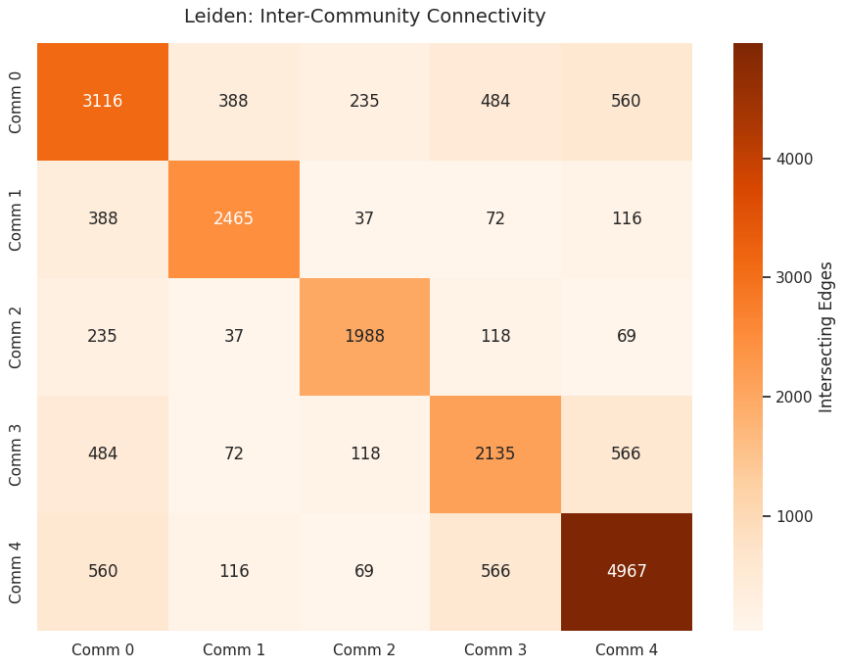


Figure 5: Communities connectivity heatmap: The communities found present high inter-connectivity between nodes inside the community while also presenting a low number of cross-community connections leading to highly independent communities.

We selected the largest community, which contains 709 nodes and 3116 edges (Figure 6), thereby considerably reducing the dimensionality of the problem.

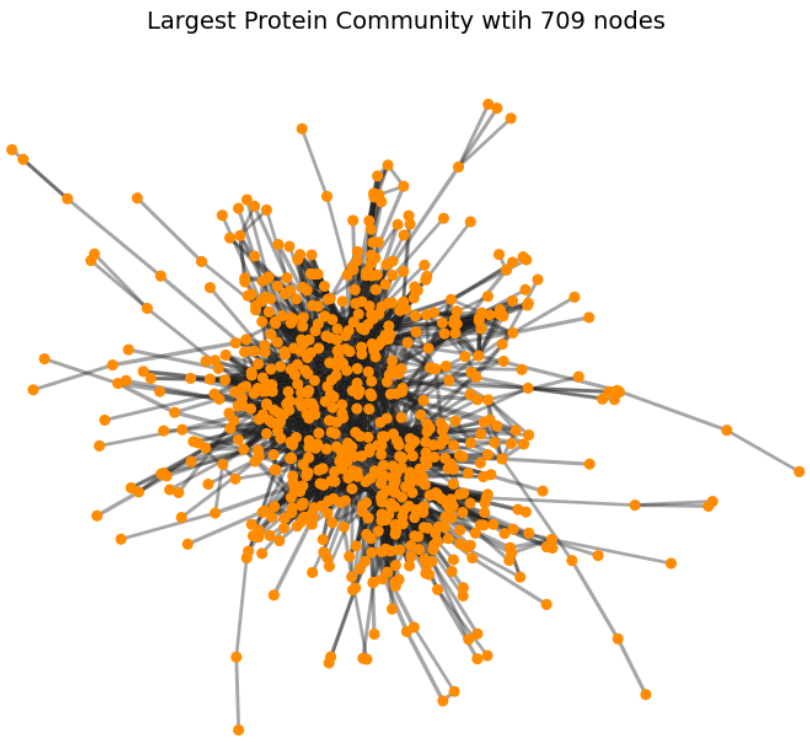


Figure 6: Largest Leiden community found in the Mus musculus PPIN.

3.2 The Hopfield Model

The Hopfield Model was introduced by J.J Hopfield in 1982 [10], where he proposed that in physical systems, made from a large number of simple elements, the interactions among simple components yields a collective phenomena such as the stable magnetic orientations and domains in a magnetic system or the vortex patterns in fluid flow. He questioned the ability of this collective phenomena for computational activities, such as, the stability of memories, the construction of general categories, or, time-sequential memory. Based on this initial theory, Hopfield proposed a model able to act as a localized content addressable memory ([10],[18]).

The proposed system is formed by a set of general coordinates with a set of stable fixed points ([10],[18]).

$$\text{Coordinates Vector}: \ S_m = \{ \ x_1, \ x_2, \ ..., \ x_n\} \in \{0, 1\} \tag{1}$$

$$\text{Stable Fixed Points}: \ V^s = \{S_1, \ S_2, \ \ldots, \ S_m\} \tag{2}$$

This set of coordinates 1 defines the neurons of the network, while the fixed points are the memories stored in the CAM. As an example, if we consider a starting position $S(t_0) = S_a + \Delta$ that is sufficiently near S_a, then, the system will converge slowly until $S \approx S_a$, where S_a is a stable fixed point. In other words, $S = S_a + \Delta$ represents a partial knowledge of the element in

memory S_a and the system takes care of retrieving the missing information of S_a by associative memory to obtain the complete element [10].

Hopfield defines these neurons 1 as having a specific state in a specific time $S_i(t)$ that represents if the neuron is 'firing' or not.

$$\begin{aligned} S_i(t) &= 0 \quad \text{The neuron is not firing} \\ S_i(t) &= 1 \quad \text{The neuron is firing} \end{aligned} \tag{3}$$

The neurons are connected to each other in the form of a complete graph, meaning, each neuron is connected to all the other neurons directly. The strength of this connection is given by a weight w_{ij} which is used to reconfigure the states of the N neurons for a specific time (t) if a threshold $U_i = 0$ is reached for each neuron.

$$S_i(t+1) = \sum_{j \neq i} w_{ij} S_j(t) > U_i \tag{4}$$

This value is generally referred to as the input potential (Eq. 5). For a given neuron (i), it is calculated as the sum of all inputs received by that neuron and determines whether its state will be ON (+1) or OFF (-1), depending on whether the input potential is greater than or less than U_i, respectively.

$$h_i(t) = \sum_{j} w_{ij} S_j(t) \tag{5}$$

For a set of states $V^s = \{S_1,\ S_2,\ \ldots,\ S_n\}$ that we want to store, the following algorithm was proposed to calculate the weights of the neurons:

$$w_{ij} = \sum_{s} (2V_i^s - 1)(2V_j^s - 1) \tag{6}$$

The patterns stored are in the network binary ($V_i \in \{0, 1\}$). Hopfield proposes Equation 6 to convert the binary states of the patterns to a bipolar state. That way, for $V_i = 1$ then $(2V_i^s - 1) = 1$ and for $V_i = 0$ then $(2V_j^s - 1) = -1$. This is done since the calculation of the weights is as follows:

$$w_{ij} = \sum_{s} V_i^s V_j^s \tag{7}$$

If the weights were calculated as shown in 7, for $V_i^s = 0$, $V_j^s \times V_j^s = 0$ regardless of the value of V_j^s being 1 or 0. This conflict would then not properly take into account the states of the neurons to calculate the weights. This is why equation 6 is used instead.

3.3 Attractor Clustering and visualization

In order to find the attractor points in the biological network, we applied two approaches: a Greedy assignment search and a Two-Pass search. The goal of both algorithms is to find what we will call attractor clusters. These clusters are a group of final states retrieved by the model which are in a certain distance radius. To calculate the distance between each of the final

retrieved states we made use of the Hamming distance, which is often used to quantify the extent to which two bitstrings of the same dimension differ [19]. The Hamming distance is defined as follows: given two bitstrings of the same dimension, the Hamming distance is the minimum number of symbol changes needed to change one bitmap into the other [19]. For example, for $S_a = [1, 0, 0]$ and $S_b = [1, 1, 1]$ the Hamming distance would be 2 since it's the minimum amount of changes to S_a or S_b needed for $S_a = S_b$.

Algorithm 1 Greedy Leader Clustering Algorithm

procedure CLUSTERING(*states_matrix*, *distance_threshold*, *num_runs*)
    states_matrix ← array with the final retrieved patterns from the model for all the runs
    num_runs ← number of times the model has run
    distance_threshold ← hamming distance threshold
    assigned ← array of size *num_runs* filled with *False*
    cluster_labels ← array of size *num_runs* filled with *null*
    cluster_counts ← empty list
    cluster_id ← 1
    for $i \leftarrow 0$ **to** $num\_runs - 1$ **do**
        if $assigned[i] = True$ **then**
            continue
        end if
        seed_state ← $states\_matrix[i]$
        mismatches ← $(states\_matrix \neq seed\_state)$
        hamming_distances ← mean of *mismatches* along columns
        in_same_cluster ← $(hamming\_distances \leq distance\_threshold) \wedge (\neg assigned)$
        runs_in_cluster ← $\sum(in\_same\_cluster)$
        APPEND(*cluster_counts*, *runs_in_cluster*)
        $cluster\_labels[in\_same\_cluster] \leftarrow cluster\_id$
        $assigned[in\_same\_cluster] \leftarrow True$
        $cluster\_id \leftarrow cluster\_id + 1$
    end for
    return *cluster_labels*, *cluster_counts*
end procedure

The Greedy clustering algorithm is a basic implementation of the Leader clustering algorithm [20] and works as follows: first, it defines an *assigned* array to distinguish which patterns have been allocated to a cluster. Then, for all retrieved patterns, the algorithm checks whether the patterns of a given run has already been assigned to a cluster. If it has not, that specific run is defined as a *seed_state* (the center of a new cluster). The algorithm then uses the *states_matrix*, which contains all the final patterns, to calculate the Hamming distance between all states and the *seed_state*. Next, these Hamming distances are compared against a predetermined Hamming threshold. For each distance calculated, if a retrieved patterns distance to the *seed_state* is less than or equal to the threshold, and that given pattern has not ben assigned yet, it becomes assigned to the new cluster.

On the other hand, the Two-Pass clustering algorithm 2 is a modification of the classic Leader Algorithm [20] implemented in algorithm 1, that aims to assign state patterns to their

closest cluster a more precisely. This is achieved by first finding the cluster centers using the same method as in algorithm 1, but only the centers are mapped and none of the remaining patterns are assigned to any cluster center. Once all of the cluster centers have been found, the

Algorithm 2 Two-Pass Cluster Discovery and Assignment Algorithm

```
procedure TWO-PASS CLUSTERING ALGORITHM(states_matrix, distance_threshold)
    num_runs, num_neurons ← dimensions of states_matrix
    Step 1: Discover Unique Seed States
    assigned_discovery ← array of size num_runs filled with False
    unique_seeds ← empty list
    for i ← 0 to num_runs − 1 do
        if assigned_discovery[i] = True then
            continue
        end if
        seed_state ← states_matrix[i]
        APPEND(unique_seeds, seed_state)
        mismatches ← (states_matrix ≠ seed_state)
        hamming_distances ← mean of mismatches along columns
        in_same_cluster ← (hamming_distances ≤ distance_threshold)
        assigned_discovery[in_same_cluster] ← True
    end for
    Step 2: Closest-Center Assignment
    cluster_labels ← array of size num_runs filled with − 1
    for i ← 0 to num_runs − 1 do
        run_state ← states_matrix[i]
        mismatches ← (unique_seeds ≠ run_state)
        distances_to_seeds ← mean of mismatches along columns
        closest_cluster_id ← index of minimum value in distances_to_seeds
        if distances_to_seeds[closest_cluster_id] ≤ distance_threshold then
            cluster_labels[i] ← closest_cluster_id
        end if
    end for
    return cluster_labels, unique_seeds
end procedure
```

algorithm proceeds to determine, for each unassigned pattern in the *states_matrix*, which center it is closest to, by once again calculating the Hamming distance between the pattern and all the cluster centers. Through this process, instead of assigning the patterns in the *states_matrix* to the first cluster center, without taking into account possible centers not yet found, all of the cluster centers that may be found are first laid out, ensuring that each pattern is assigned to the cluster to whose center it is closest.

In order to visualize the clusters found by clustering algorithms we will resort to the scikit-learn python library [21] to use Principal Component Analysis (PCA). PCA is a linear dimensionality reduction technique with applications in high dimensionality data, and can be expressed as a projection-based approach, finding the low-dimensional space that best represents a cloud

of high-dimensional points [22]. It is commonly use for dimensionality reduction of binary data [23], which is the format of the patterns we will be working with.

4 Results

4.1 Standard Hopfield Network

In order to validate our model, we first implemented a standard Hopfield network. For this implementation, we used a network of $N = 3969$ nodes trained with $0.138N$ randomly generated patterns, which is the maximum theoretical capacity of patterns the network can store [10] and equates to 547 patterns stored. While the network architecture does not impose any restriction on the number of neurons, $N = 3969$ was explicitly selected to facilitate two-dimensional visualization. Because 3969 is a perfect square (63×63), it allows the network states to be displayed as a 2D pixel grid.

The network training is performed using the method described in equation 6 and the update rule is applied following the equation 5.

After training, to confirm the network functions as intended, for each of the 547 stored patterns we apply noise to the pattern by randomly flipping 15% of the states of the pattern. After generating the noisy patterns, we run the model on each of them to evaluate its ability to retrieve the corresponding original stored pattern. Doing this, out of the $547 \approx 0.138 \cdot 3969$ patterns that we attempted to retrieve with a 5% error acceptance (Hamming distance) with respect to the expected pattern, we obtained that 470 patterns converged to the expected memory pattern. The other 77 did not reach the 5% error threshold and did not converge to the expected pattern. The mean error of all the retrieval attempts was 0.0470 with an average of 12.67 retrieval steps. In summary, we were able to retrieve 85.92% of the stored patterns with a 5% error acceptance, while the other 14.08% did not converge. While these results seem to contradict the traditionally accepted capacity of a Hopfield network, they align with the work of Amit, Gutfreund, and Sompolinsky (1985) [24]. They demonstrated that as the ratio of stored patterns (P) to neurons (N) approaches the critical capacity ($\alpha = P/N \approx 0.138$), interference from other stored memories increases. Equations (8-11), presented by Amit, Gutfreund, and Sompolinsky [24], evaluate the network's macroscopic steady states by tracking the interactions between the retrieval overlap (m), the background noise (r), and the network susceptibility (C) at the low-temperature limit ($T \to 0$). The system is bounded by the zero-temperature limit of the Edwards-Anderson order parameter, q:

$$q = 1 - CT \tag{8}$$

This parameter q measures memory persistence over time by checking if a neuron's state remains correlated with itself infinitely into the future. As temperature approaches absolute zero ($T \to 0$), fluctuations vanish and $CT \to 0$. This leaves $q = 1$, which signifies that the network's states are completely frozen into fixed, static configurations. The stability of these configurations depends on the network's susceptibility to background noise, C, which is defined as:

$$C = \sqrt{\frac{2}{\pi \alpha r}} \exp\left(-\frac{m^2}{2\alpha r}\right) \tag{9}$$

Equation 4.1 measures how sensitive the neurons' local alignment is to the macroscopic interference generated by the other stored memories. This susceptibility is strictly modulated by the current retrieval overlap (m) within the exponential term $\exp(-m^2/2\alpha r)$. C directly determines the scaling of the background crosstalk parameter, r:

$$r = \frac{1}{(1-C)^2} \tag{10}$$

Here, r represents the structural variance of the interference caused by stored patterns, acting as a Gaussian noise vector with a variance proportional to $\alpha \cdot r$. As Equation 10 indicates, any increase in noise susceptibility C shrinks the denominator $(1-C)^2$, causing the background crosstalk r to amplify. Finally, this accumulated crosstalk acts as the primary regularizer for the final retrieval overlap, m, in the self-consistent field equation:

$$m = \operatorname{erf}\left(\frac{m}{\sqrt{2\alpha r}}\right) \tag{11}$$

Equation 11 is a self-consistent field equation, meaning the variable m appears on both sides because the macroscopic state of the network determines the behavior of individual neurons, which in turn recalculates the macroscopic state. It uses the error function (erf) to balance the retrieval signal (the numerator, m) against the total magnitude of the background noise (the denominator, $\sqrt{2\alpha r}$). This establishes a continuous mechanism. If α is too high, the background noise $\sqrt{2\alpha r}$ suppresses the error function, driving $m \to 0$. This collapse of m removes the negative exponent in Equation 8, maximizing C, which in turn drives $r \to \infty$ in Equation 10. This locks the network into a Spin-Glass state which is a disordered, non-retrieval state where the neurons freeze into random, conflicting configurations eliminating any capacity of associative memory.

Sompolinsky et al. 1985 [24] define two solutions for the set of equations (8-11). The first one, where $m = 0$, and $q, r \neq 0$. They define this solution as a representation of a Spin-Glass state, which has no macroscopic overlap with the stored patterns and does not contribute to associative memory. The second solution is a Ferromagnetic state defined by $m \neq 0$. This solution enables associative memory and exists for sufficiently small values of α. At the zero-temperature limit ($T \to 0$), the temporal order parameter freezes completely to $q = 1$ Our retrieved patterns yield an average overlap of $m = 0.9060$, which directly satisfies this $m \neq 0$ condition and confirms that our network retains associative capabilities at $\alpha \approx 0.138$. However, because this overlap falls bellow ($m < 1$) the presence of background crosstalk (r), manifests as a 4.70% mean retrieval error and prevents the full recovery of all stored patterns.

In Figure 7 and 8, we observe the evolution of the network states on each iteration for a noisy pattern that has correctly converged to its expected pattern and another noisy pattern that did not converge, respectively. In Figure 7, we see that once the network has almost completely converged into the expected pattern it stays in that state equilibrium until we stop the retrieval process.

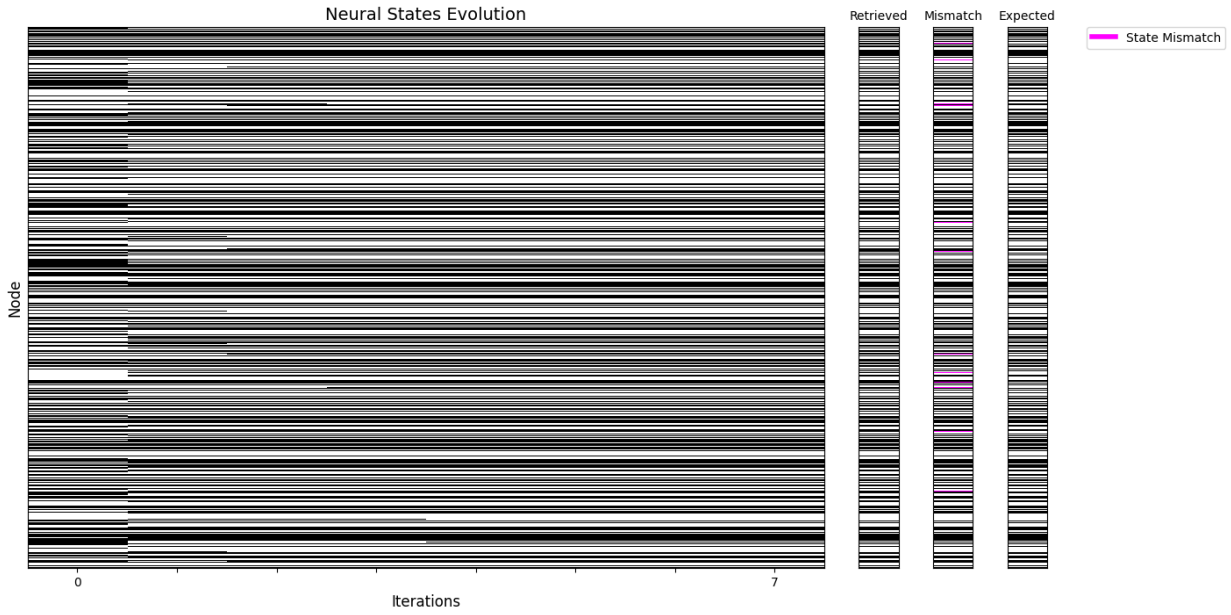


Figure 7: Display of the evolution states in the 3969 nodes standard Hopfield network trained with $0.138 \cdot N$ patterns when retrieving a pattern that converges to its expected retrieval with a 5% error acceptance. In the first column we can see the last step of the network (the retrieved pattern). In the second column we can see the retrieved pattern with the bits that don't match with the expected pattern marked. The last column shows the expected pattern the network should have converged to. As the network converges we have forcefully stopped the execution at a total of 8 step iterations. The stoppage condition is based on the Hamming distance between iterations. In every iteration, we compute the Hamming distance between the current state and the state on the previous iteration, and if it is less than or equal to 0.01, and it also doesn't increase during 5 consecutive iterations, we stop the model.

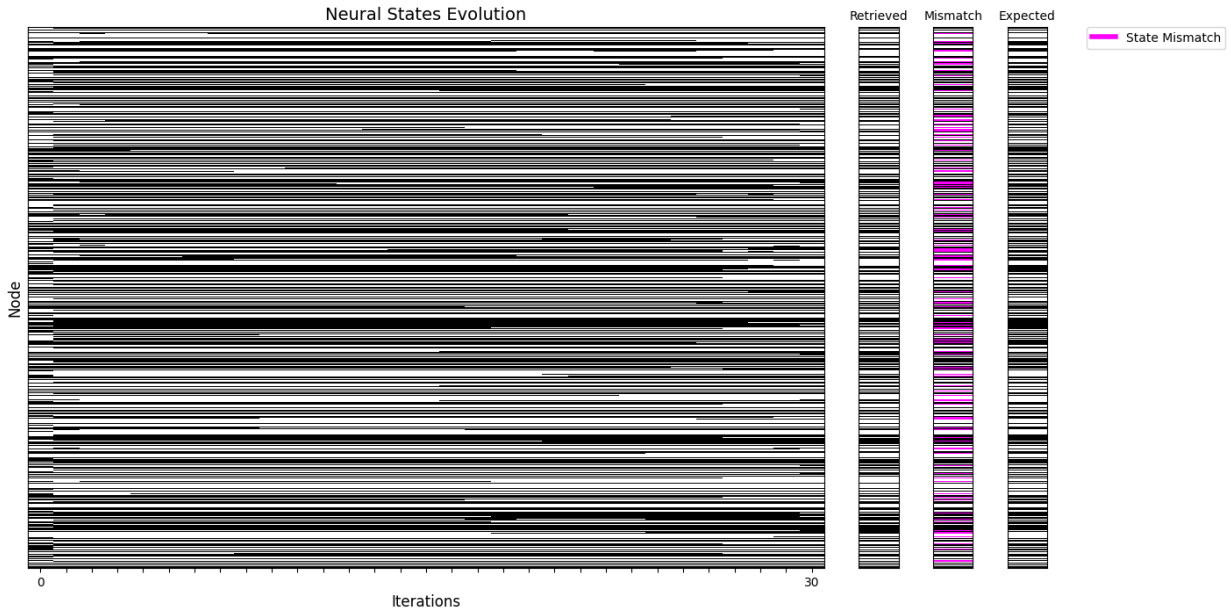


Figure 8: Same concept as in Figure 7, but this time the display is a pattern that didn't converge to its expected pattern. This phenomenon happens even if the number of steps allowed for the network to iterate is increased.

By reducing the number of stored patterns to $\alpha_c \cdot N \cdot 0.90$, in other words, reducing them to 90% of the networks theoretical maximum capacity, we observe a 100% retrieval rate with a 5% error acceptance and average retrieval error of 0.0056.

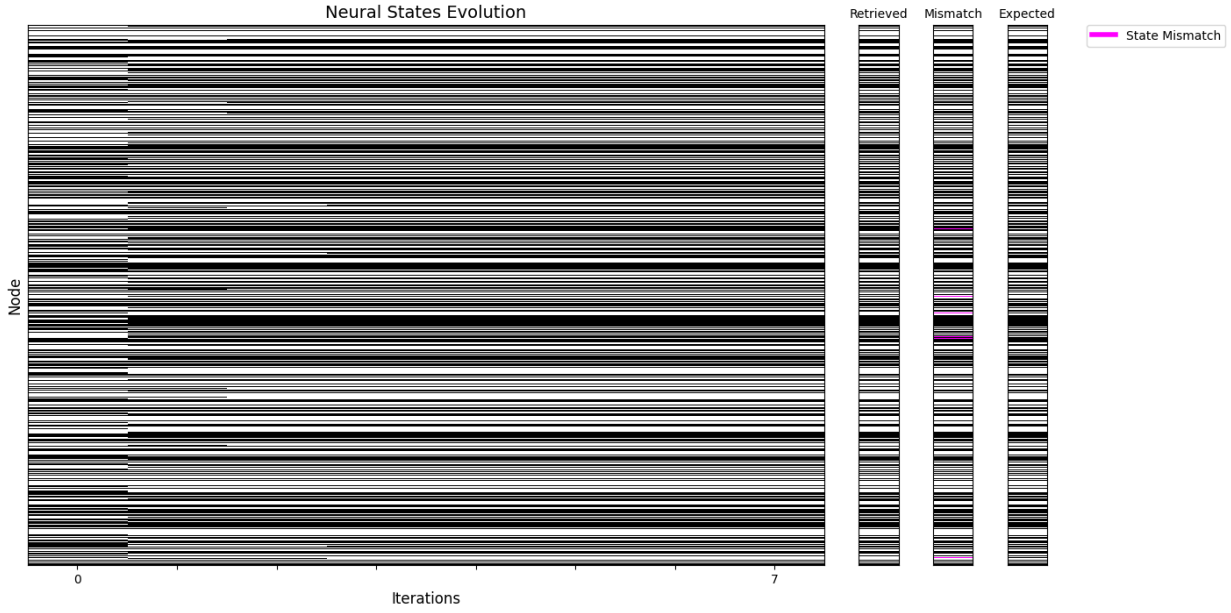


Figure 9: Display of the evolution states in the 3969 nodes standard Hopfield network trained with $0.138 \cdot N \cdot 0.90$ patterns when retrieving a pattern that converges to its expected pattern with a 5% error acceptance threshold.

4.2 PPIN and comparison with standard Hopfield networks

As previously established in Section 3.1, the PPIN was constructed using the *Mus musculus* dataset [15] from STRING [14] by isolating the largest community within the full network using a combined connection score threshold of 700. This procedure yielded a sub-network of the original PPIN containing 709 nodes and 3116 edges.

We will compare a standard 709-node Hopfield network to what we will refer to as the biological Hopfield network. To find the attractors of this biological network, we propose the following experiment: generate states in the N-dimensional space of the network, feed them as inputs, and observe where the states converge. Attractors can then be extracted by identifying recurring, final retrieval patterns across all simulation runs. Following the methodology of Kauffman [6], we will extract attractors by identifying the final states, recurring retrieval patterns that multiple initial states converge toward during the simulation runs. Since the patterns are binary $(0, 1)$, there are a total of 2^{709} possible initial states to test. Testing every single possibility is an NP-hard problem and computationally impossible for us. Instead, we propose to sample a more manageable subset of 5000 randomly generated initial states.

We will evaluate the network configurations using the following four comparisons:

1. Baseline Configurations with Random Weights:
    - Standard Hopfield Network — random patterns retrieval.
    - Sparse Biological Hopfield Network — random patterns retrieval.
2. Standard Hopfield Network trained at maximum capacity ($\alpha = 0.138$) — stored patterns retrieval and random patterns retrieval.
3. Sparse biological Hopfield network with combined score weights from the STRING database — random patterns retrieval.

For each of these experiments, we generate a set of 5000 random initial states to feed into the networks as inputs. We then monitor the networks behavior to observe whether they converge into unique attractor points or attractor clusters. Across all four cases, we apply the same convergence criteria: a network is considered to have settled into a stable state if the Hamming distance between its current and previous states remains ≤ 0.01 for five consecutive iterations without increasing.

Additionally, we enforce a strict limit of 100 iterations for the network to converge and reach stability. Once a given experiment has finished processing all 5000 initial states, the clustering algorithms introduced in Section 3.3 will be applied to the entire set of retrieved states to identify clusters among final patterns that share similarities.

Baseline Configurations with Random Weights

To establish a baseline for comparison, we first evaluated the standard fully connected network and the sparse biological network using completely random weights. Testing these systems with 5000 random initial states shows how both networks behave before any structured weight data is introduced.

As shown in Figures 10 and 11, all 5000 simulation runs for both network types successfully converged below the designated 0.01 Hamming distance threshold.

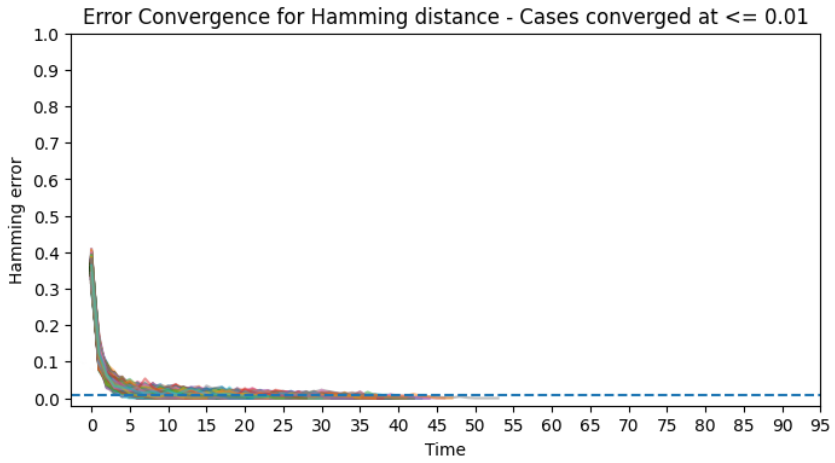


Figure 10: Convergence error (in the standard Hopfield network with random weights) for cases stopped by the 0.01 Hamming distance threshold. All 5000 runs were stopped because of stable convergence.

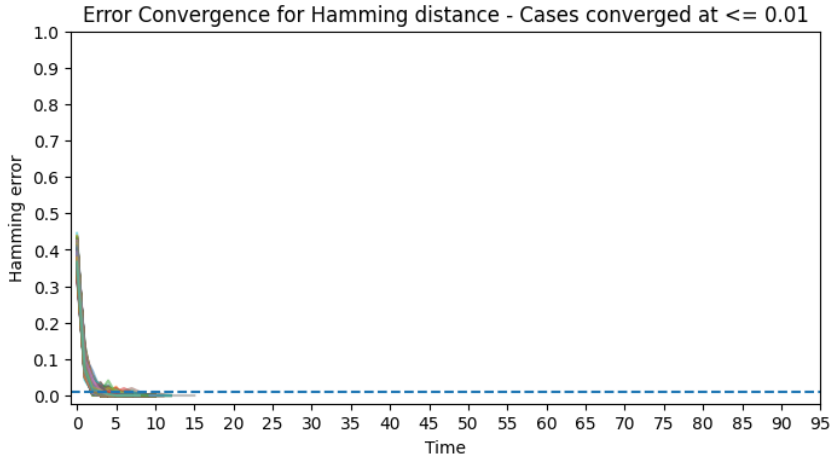
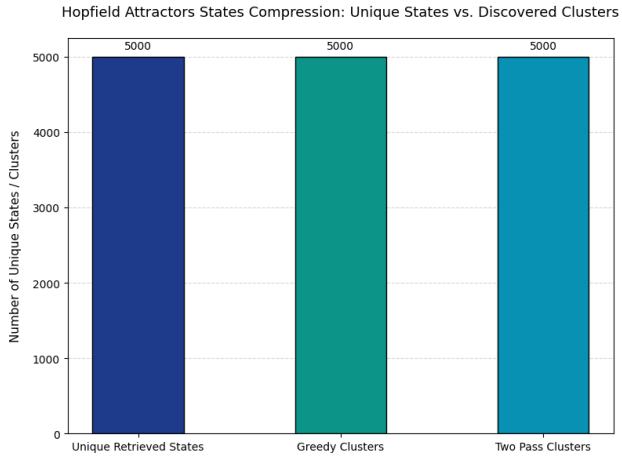
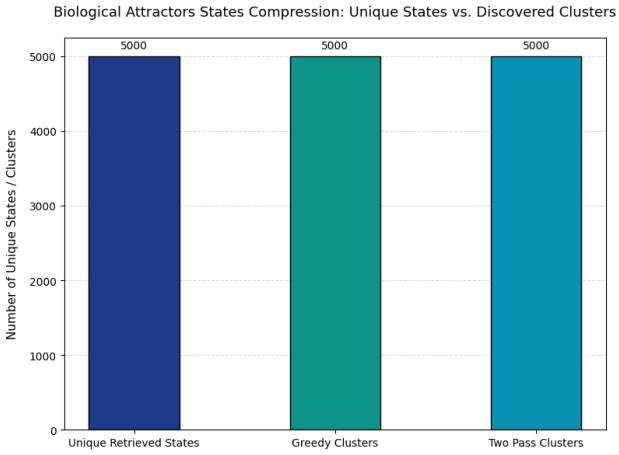


Figure 11: Convergence error (in the biological Hopfield network with random weights) for cases stopped by the 0.01 Hamming distance threshold. All 5000 runs were stopped because of stable convergence.

When looking for unique attractors among the final states, both networks behaved identically; no two initial states converged to the same attractor. Applying the Greedy and Two-Pass clustering algorithms with a 15% threshold resulted in exactly 5000 separate, single-element clusters for both models, as shown in Figure 12.



(a) Standard Hopfield network with random weights.



(b) Biological Hopfield network with random weights.

Figure 12: Attractor clusters found by the Greedy and Two-Pass algorithms with a 15% threshold between the states recovered by random weights. In both cases, no clusters were formed for either the (a) Standard Hopfield network or the (b) Biological Hopfield network.

These identical results confirm that without structured weights, neither the standard Hopfield network nor the sparse Biological network can group random inputs into shared attractor regions. With these random baselines established, the following sections examine how structured weight distributions change these network behaviors.

Standard Hopfield Network - trained at maximum capacity

For this test, a standard Hopfield network was trained at its maximum capacity ($\alpha = 0.138$). Since $0.138 \cdot 709 \approx 97$ patterns were used to train the network, running 5000 trials means we have to repeat some of these 97 patterns. To make sure every single run evaluates a different input, we added 15% random noise (flipping binary states) to each selection.

After running the model for these 5000 noisy training patterns, Figure 13 shows that all simulations converged below the 0.01 Hamming distance threshold, reaching a stable state.

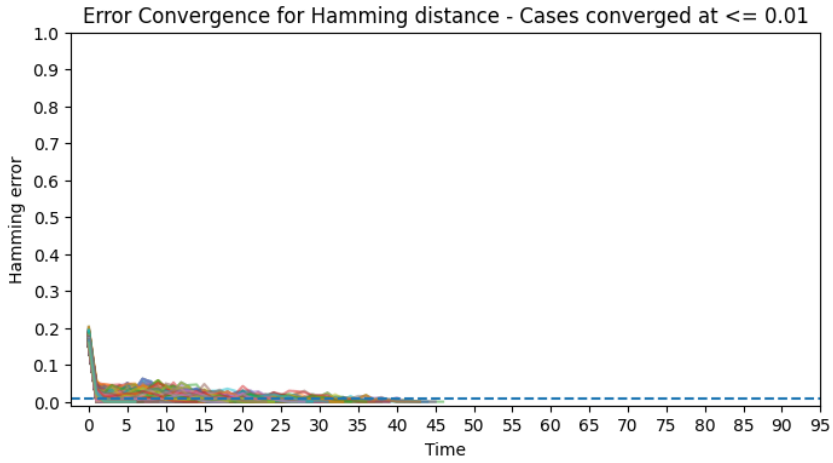


Figure 13: Convergence error for cases stopped by the 0.01 Hamming distance threshold in a trained network when retrieving noisy stored patterns. All 5000 runs were stopped because of stable convergence.

Out of the 5000 runs, 85.3% successfully converged back to an original stored pattern within an allowed error of 5%, as shown in Figure 14. The other 14.7% of the runs failed to recover a target memory and fell into false states.

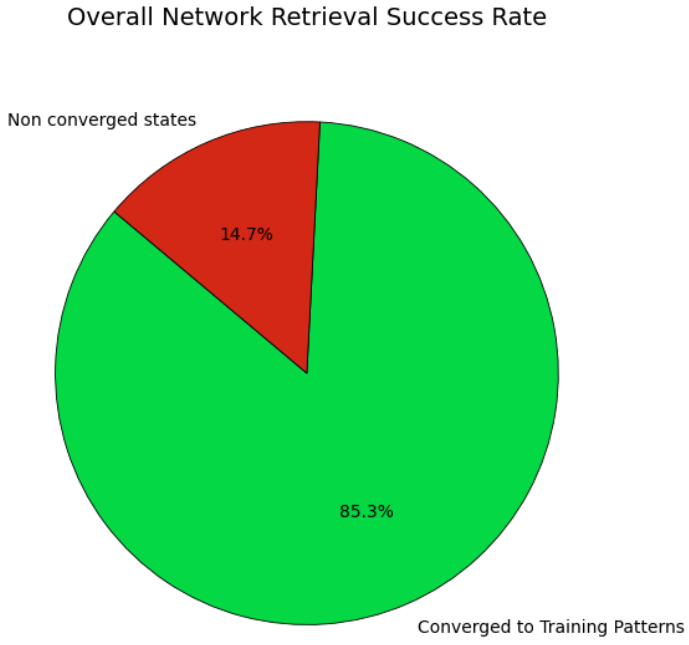


Figure 14: Pie chart showing the percentage of runs that converged into a stored pattern in a standard Hopfield network trained with $0.138N$ patterns.

When we ran the clustering algorithms on these final states, both the Greedy and Two-Pass methods found exactly 139 distinct groups, as shown in Figures 15 and 16.

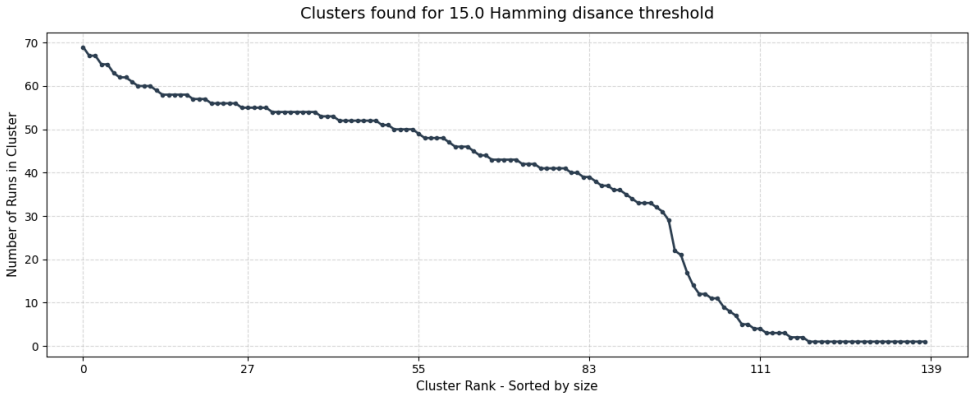


Figure 15: Clusters found by the Greedy algorithm with a 15% threshold on the states recovered by a Hopfield network when retrieving noisy stored patterns.

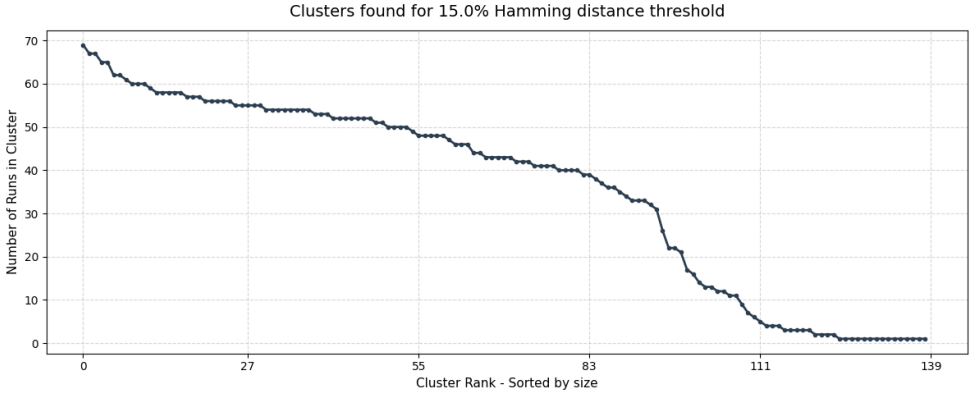


Figure 16: Clusters found by the Two-Pass algorithm with a 15% threshold on the states recovered by a Hopfield network when retrieving noisy stored patterns.

Finding 139 clusters (which exceeds the 97 originally stored training patterns) reflects the known limitations of a fully connected Hebbian network operating at its theoretical maximum storage capacity ($\alpha = 0.138$). At this limit, significant crosstalk and interference exist between the memory basins as we previously saw in Section 4.1. When the system is initialized with 15% random noise, the network is frequently unable to perfectly resolve the target attractors, settling instead into nearby stable local energy minima or spurious states formed by combinations of the stored patterns. The clustering algorithms differentiate these stable, non-target configurations as distinct, independent attractor clusters.

To see what happens when we remove the memory hints entirely, we ran the same trained network using 5000 completely random initial states. This time, the network converged into 4993 unique final states, and the clustering algorithms revealed a heavily fragmented attractor structure.

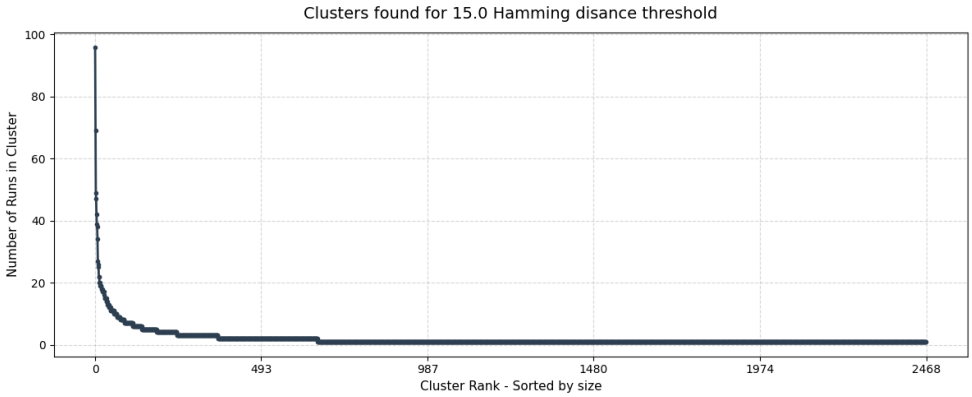


Figure 17: Attractor clusters found by the Greedy algorithm with a 15% threshold between the states recovered by a trained Hopfield network. The algorithm identifies ≈ 2500 clusters.

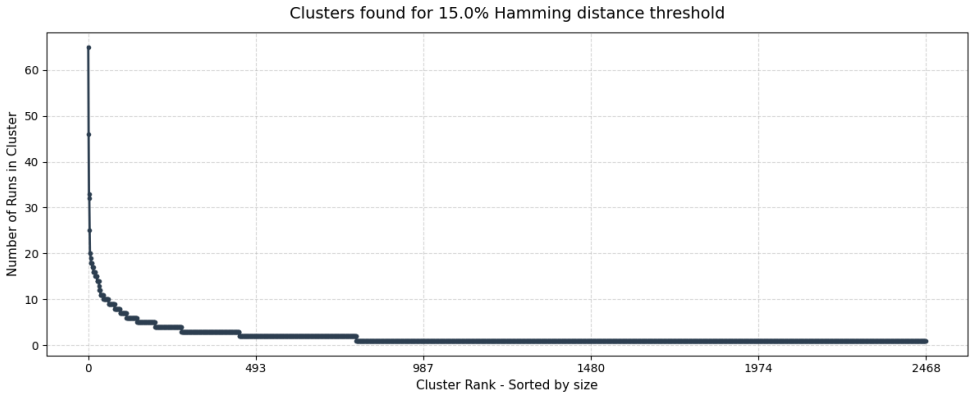


Figure 18: Attractor clusters found by the Two-Pass algorithm with a 15% threshold between the states recovered by a trained Hopfield network. The algorithm identifies ≈ 2500 clusters.

As shown in Figures 17 and 18, both algorithms found exactly 2468 separate clusters. These are classic spurious attractors (false memories). Compared to the neat 139 clusters we got using noisy memory inputs, this high number happens because none of the random initial states start near a real memory basin. This is explained by looking at the overlap graph in Figure 19.

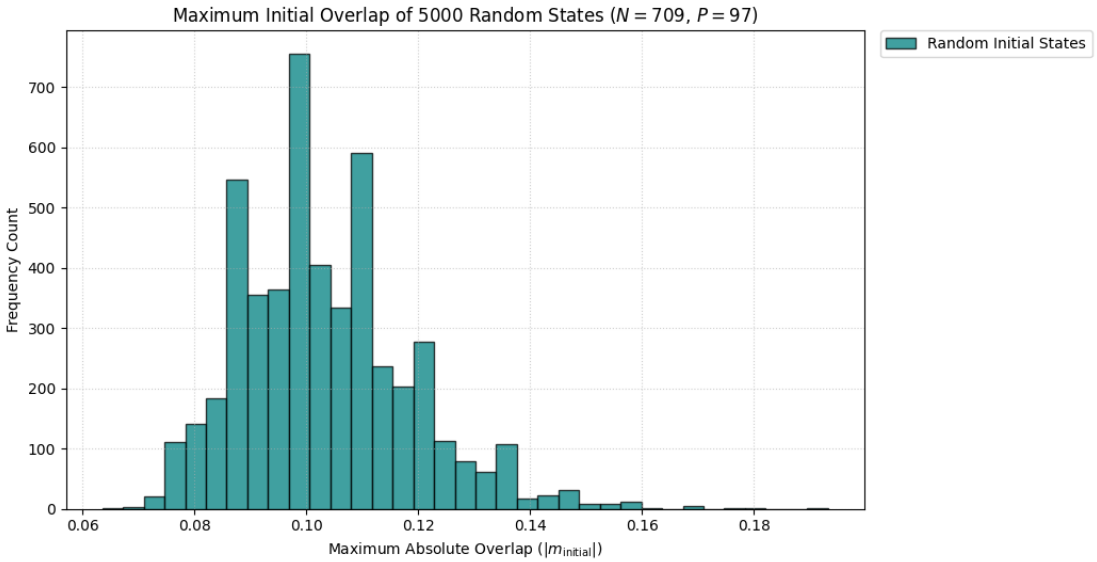


Figure 19: Maximum overlap calculated for each initial random state with any given stored pattern. The highest recorded overlap is ≈ 0.19 (19%) and the average overlap measured sits at 10%.

On average, these random initial states only share a 10% overlap with any stored memory (at most 70 matching nodes out of 709). Because they start too far away from any real memory valleys, the network cannot fix them, causing them to drift into false, spurious states.

We have established that 85.3% of the noisy stored patterns successfully converge to a memory state. To visualize the landscape of these successful retrievals, we focus on 85% of the total initial random states (4200 states). Figure 20 shows the distribution of these attractor clusters containing a total of 4200 states combined, displaying the localized basins associated with successful memory recall.

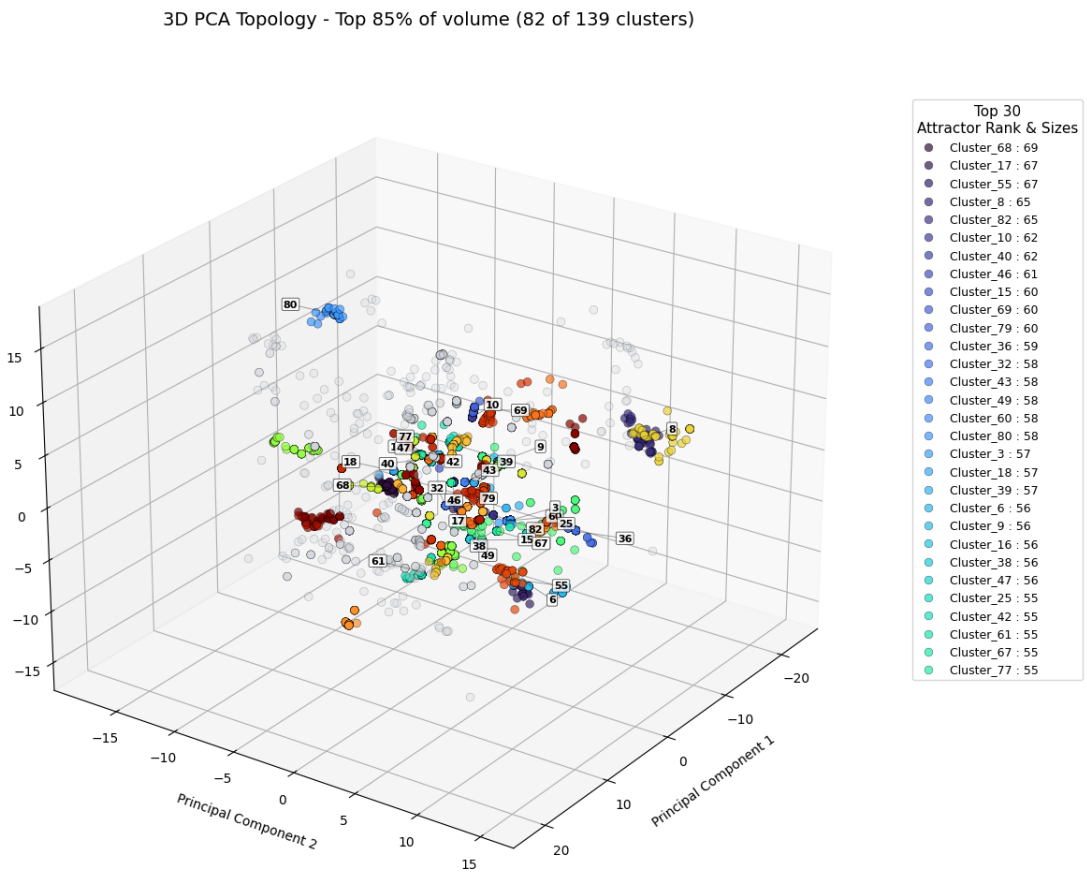


Figure 20: 3D PCA plot showing the top 85% volume of attractor clusters (found with the Two-Pass algorithm) during noisy stored pattern retrieval. These top 85% clusters have a cumulative size count of 4200, which is approximately the total correctly retrieved patterns for this simulation. This results in a total of 82 clusters being color-coded and displayed. The threshold is selected to align with the 85.3% successful retrieval rate observed in the network. The legend shows the top 30 largest clusters.

In contrast, Figure 21 shows the distribution when the same trained network attempts to retrieve 5000 completely random initial states. The landscape is vastly more fragmented, with the volume spread across 2468 spurious attractors. For this visualization, we have color-coded the top 30 clusters by population size while rendering the remaining minor clusters in light gray. This approach captures the high-density fragmentation of the energy landscape, demonstrating how the surface is dominated by a background of disparate, unstable states when the input lacks overlap with stored memory.

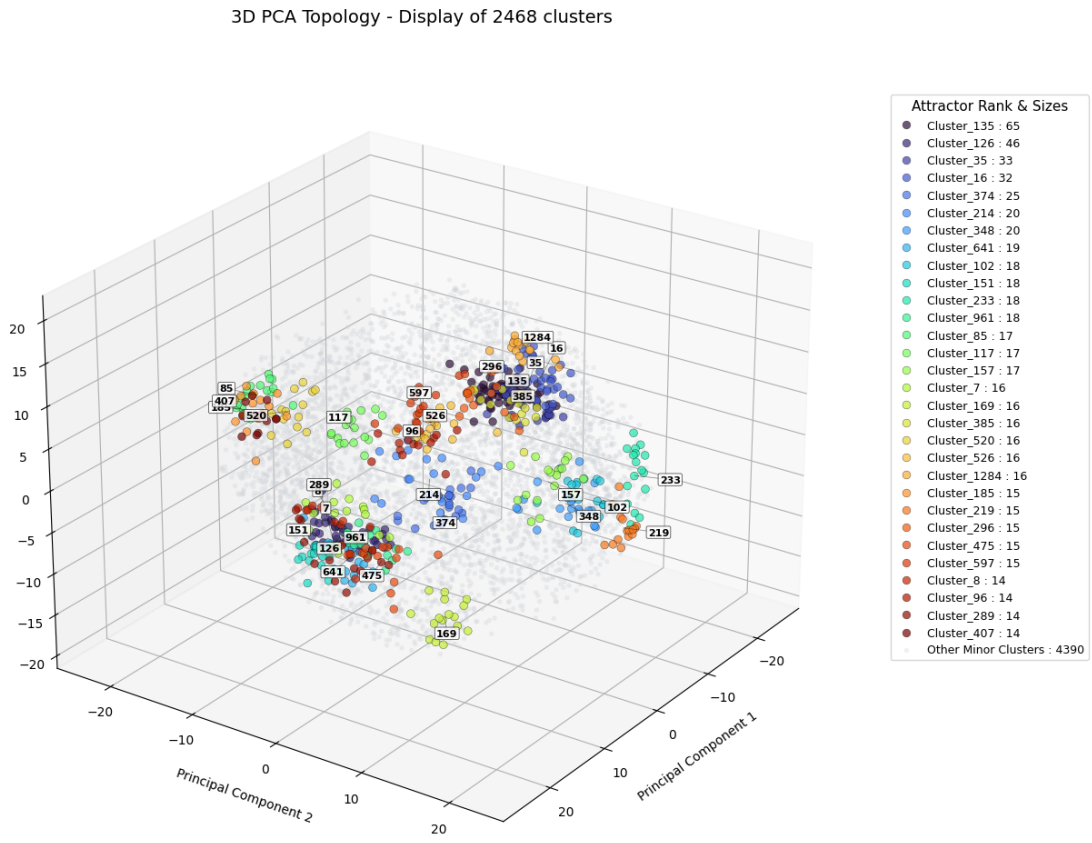


Figure 21: 3D PCA plot of the 2468 attractor clusters (found with the Two-Pass algorithm) during random input retrieval. The top 30 clusters are color-coded to display the high degree of fragmentation and the scattered nature of the spurious states.

This comparison shows that a fully connected network is not static. Whether it appears as a functional set of memory basins or a chaotic field of spurious attractors is dependent on whether the initial input aligns with stored information.

Biological Hopfield Network with Combined Score Weights

The biological Hopfield network, configured with interaction weights derived from the STRING database [14], demonstrates a different dynamic compared to the standard network. The convergence metrics for 5000 independent trials, using the same 0.01 Hamming distance threshold, are presented in Figure 22.

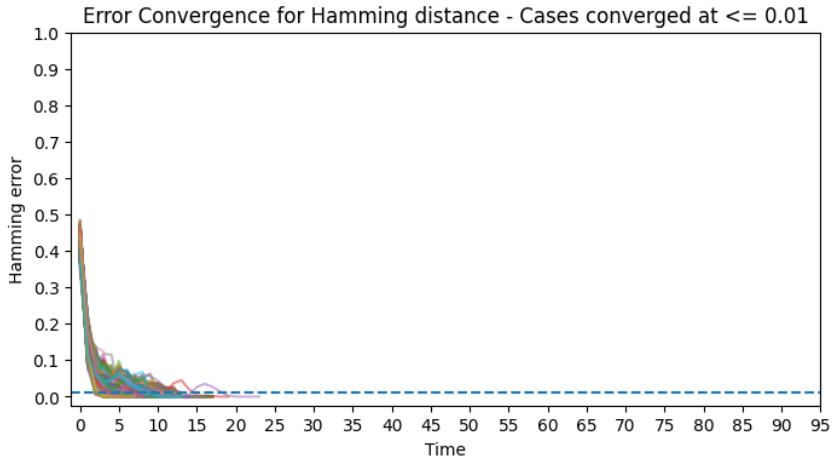


Figure 22: Convergence error (in the biological Hopfield network with combined score weights) for cases stopped by the 0.01 Hamming distance threshold. All 5000 runs were stopped because of stable convergence.

When initialized with random input patterns, the biological network identified 459 distinct attractor clusters, as shown in Figures 23 and 24. This result represents a decrease in the number of identified clusters compared to the 2468 clusters observed in the standard network under identical random input states.

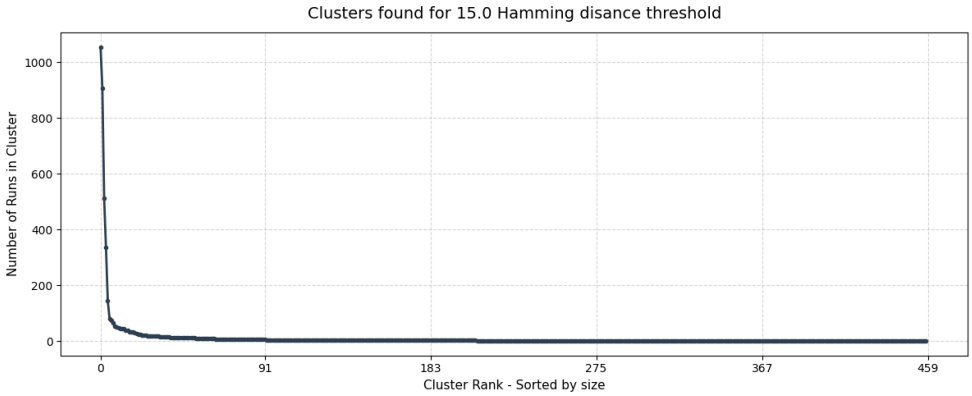


Figure 23: Attractor clusters found by the Greedy algorithm with a 15% threshold between the states recovered by a biological Hopfield network with combined score weights. The algorithm classified 459 attractor clusters.

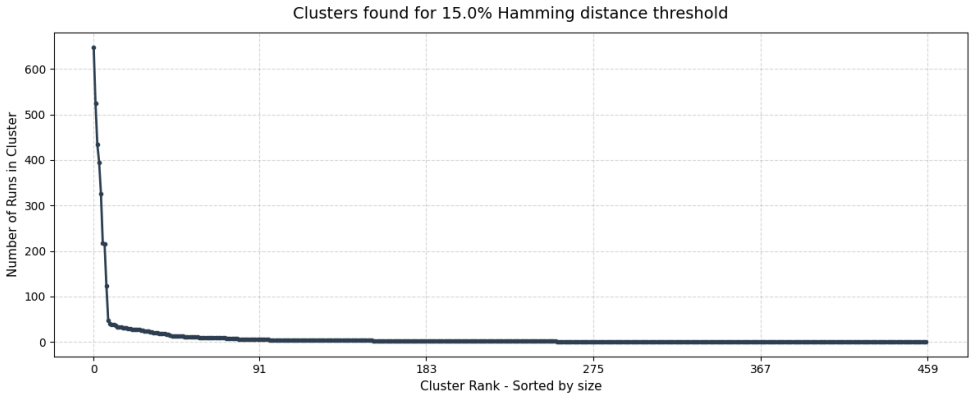
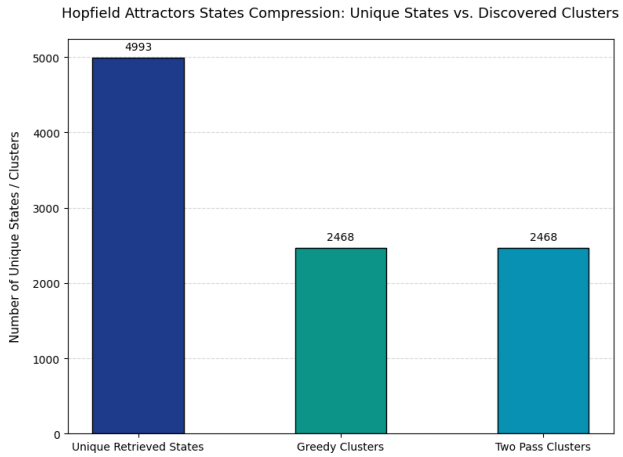
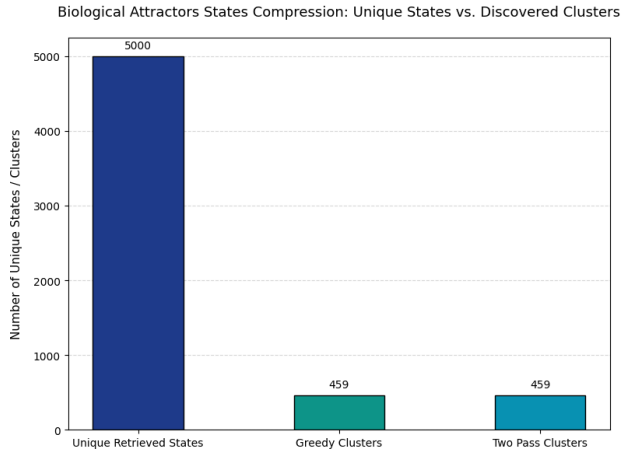


Figure 24: Attractor clusters found by the Two-Pass algorithm with a 15% threshold between the states recovered by a biological Hopfield network with combined score weights. The algorithm classified 459 attractor clusters.

Comparing these results with the standard Hopfield network, we can observer a high difference of clusters detected when both networks where tasked to retrieve random initial inputs (See Fig. 25).



(a) Standard Hopfield network with trained weights.



(b) Biological Hopfield network with combined score weights.

Figure 25: Attractor clusters found by the Greedy and Two-Pass algorithms with a 15% threshold after the networks retrieved 5000 random patterns. The standard Hopfield network with trained weights and the biological Hopfield network with combined score weights. The standard Hopfield network generated a total of 2468 clusters while the biological Hopfield network generated a much lower 459 total of clusters.

While both algorithms found the exact same number of clusters (459), they distribute the 5000 runs differently. In the Greedy plot (Figure 26), the two largest clusters take up about 40% of all the runs (Cluster 1 with 1055 states and Cluster 2 with 906 states). In the Two-Pass plot (Figure 27), the runs are spread out much more smoothly over the top seven largest clusters, with the single largest cluster dropping down to 648 states.

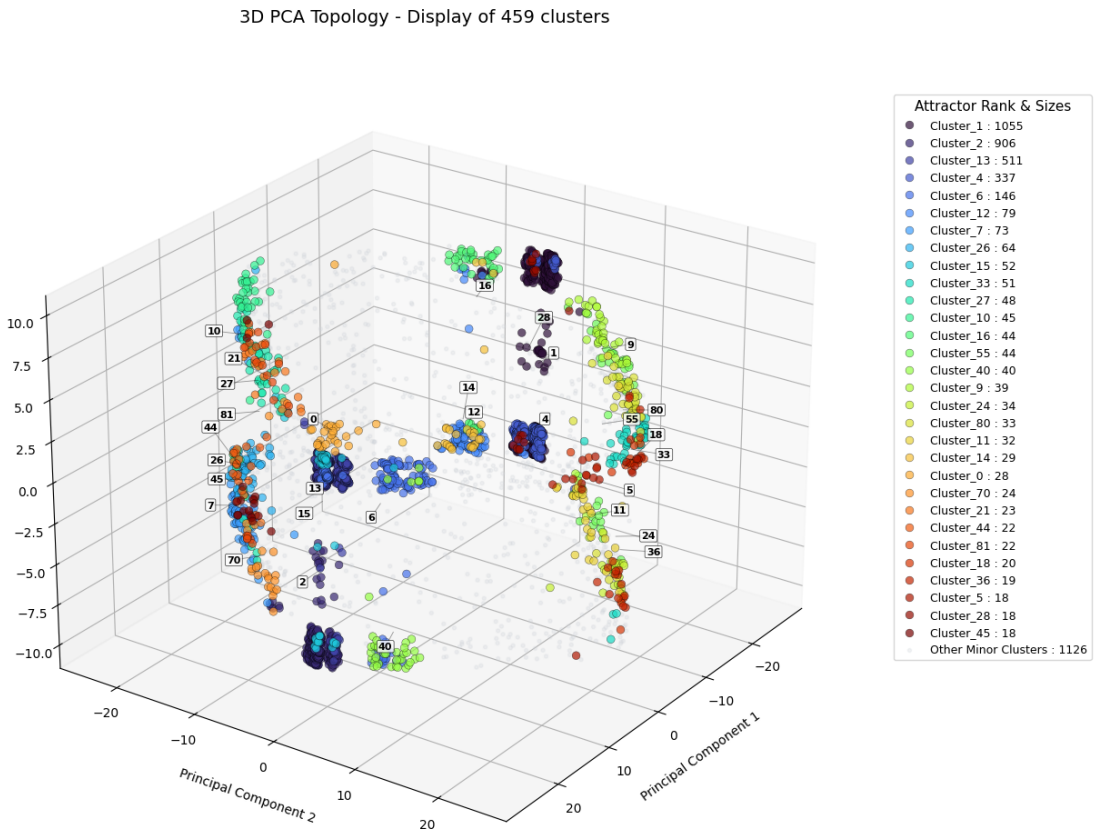


Figure 26: 3D PCA plot of the 459 found attractor clusters by the Greedy algorithm. The colored clusters are the 30 clusters with the most retrieved states assigned to them. The Other Minor Clusters element in the legend indicates the rest of the clusters that have not been color-coded.

This happens because of how the algorithms sort data. In the Greedy algorithm, as soon as a cluster center is chosen, every unassigned state within the 15% threshold is immediately absorbed by that cluster. This seems to create highly compacted clusters with a high population size and continuous, elongated stripes along the outer boundaries of the three-dimensional projection as seen in Figure 26. Because the first few selected centers capture all surrounding states within their radius, they artificially inflate their own population sizes. Resulting into a few very populated clusters and many small and distributed clusters.

The Two-Pass algorithm resolves this allocation bias. By locating all 459 valid cluster centers across the dataset, it eliminates the first-come-first-served advantage. This shift changes the distribution pattern shown in Figure 27. Instead of continuous bands, the states separate into localized clusters separated by somewhat empty regions of space. Each run is assigned strictly to its closest cluster center, which lowers the sizes of the previously dominant clusters, distributes the states more evenly, and provides a more accurate density map of the underlying attractor clusters.

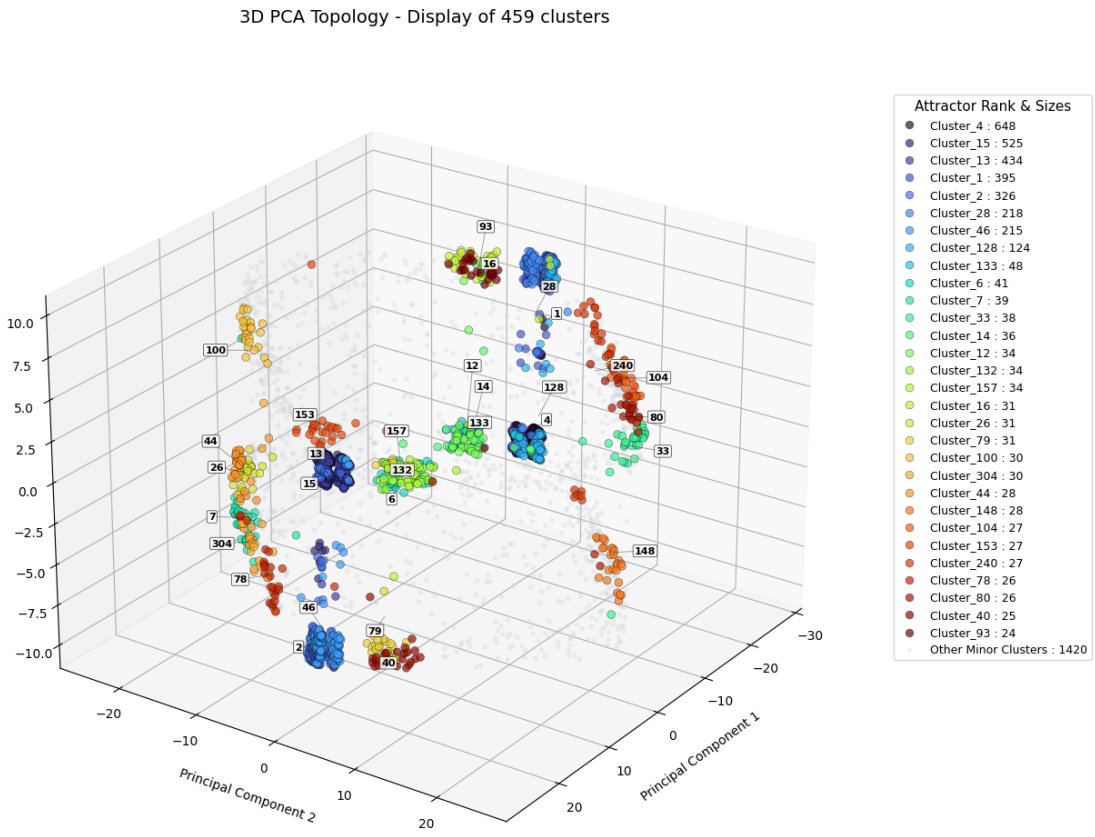


Figure 27: 3D PCA plot of the 459 found attractor clusters by the Two-Pass algorithm. The colored clusters are the 30 clusters with the most retrieved states assigned to them. The Other Minor Clusters element in the legend indicates the rest of the clusters that have not been color-coded.

5 Conclusion

Using the previous work of Kauffman et al. [6] and the definition of cell types as attractor states in Boolean networks, we transform this concept and propose the idea of cell types as attractors in protein-protein interaction networks.

Using the Hopfield Model proposed by J.J Hopfield [10], we developed a functional Hopfield network capable of producing associative memory which also shares it's non directional characteristic with PPINs.

From here we defined a Biological network with sparse connectivity by using the STRING database [14] and its large repertoire of data on PPINs. The weights of the Biological network are set by the combined scores of the protein links according to the STRING database, which are scores based on the evidence of the link between proteins. A high confidence value is defined at 700 or greater, therefore when building the biological network we considered only the links that met this criteria.

We compared the intended behavior of a Hopfield network with the Biological. When retrieving the patterns stored in the standard network we saw and 85% of retrieval success for a network trained at maximum capacity. Reaching a total of 139 attractor cluster for a Hamming distance of 15%. Since for the biological network we are trying to find these attractors, we ran the model using random initial input patterns. Using this method, we reached a total of 459 attractor clusters, which we displayed using PCA plots to visualize the attractors in a 3D space. Ultimately, the discovery of these 459 distinct attractor clusters within a biologically constrained network supports the premise that stable cell types can be mapped onto the complex topology of protein-protein interaction networks by using real-world scientific data. This framework bridges the gap between theoretical physics and functional molecular biology. These identified attractors represent strong candidates for specific cellular phenotypes or functional states. Consequently, this model provides a novel computational canvas for simulating how protein-protein interaction networks maintain cellular identity.

6 Future Work

From here, we suggest the possibility of these clusters to be near to the true attractor points of the biological network and believe that an interesting next step would be to map cell types as patterns, by use of databases like Cell Marker 2.0 [25]-[26], search for similarities between the attractor clusters found and the cell type patterns to find attractors related to specific cell types. We also believe the clustering methodology can be improved in order to find clusters of attractors more precisely. Since both algorithms are order dependent, the cluster centers are defined by the order the retrieved states are processed. True attractors centers might be ignored due to these limitations of the algorithm, resulting in more sparse clusters being formed. Finally the Biological Networks combined score weights. These weights represent confidence scores regarding the connection between two given proteins. By no means do they actually represent the 'strength' of the connection. To improve this work we believe that designing a better weight system for the Biological Network is also a requirement to improve the current results.

References

[1] Jonathan MW Slack and Leslie Dale. *Essential developmental biology*. John Wiley & Sons, 2021.

[2] Smadar Ben-Tabou de-Leon and Eric H. Davidson. "Gene Regulation: Gene Control Network in Development". In: *Annual Review of Biophysics and Biomolecular Structure* 36.1 (2007), pp. 191–212. DOI: 10.1146/annurev.biophys.35.040405.102002. URL: https://doi.org/10.1146/annurev.biophys.35.040405.102002.

[3] Maria Mircea and Stefan Semrau. "How a cell decides its own fate: a single-cell view of molecular mechanisms and dynamics of cell-type specification". In: *Biochemical Society Transactions* 49.6 (Dec. 2021), pp. 2509–2525. ISSN: 0300-5127. DOI: 10.1042/BST20210135. eprint: https://portlandpress.com/biochemsoctrans/article-pdf/49/6/2509/926725/bst-2021-0135c.pdf. URL: https://doi.org/10.1042/BST20210135.

[4] Yichun Qian and Shao-shan Carol Huang. "Improving plant gene regulatory network inference by integrative analysis of multi-omics and high resolution data sets". In: *Current Opinion in Systems Biology* 22 (2020), pp. 8–15. ISSN: 2452-3100. DOI: https://doi.org/10.1016/j.coisb.2020.07.010. URL: https://www.sciencedirect.com/science/article/pii/S2452310020300159.

[5] Eric H. Davidson and Michael Levin. "Gene regulatory networks". In: *Proceedings of the National Academy of Sciences* 102.14 (2005), p. 4935. DOI: 10.1073/pnas.0502024102.

[6] Stefan Bornholdt and Stuart Kauffman. "Ensembles, dynamics, and cell types: Revisiting the statistical mechanics perspective on cellular regulation". In: *Journal of Theoretical Biology* 467 (2019), pp. 15–22. DOI: 10.1016/j.jtbi.2019.01.036.

[7] Mileidy W. Gonzalez and Maricel G. Kann. "Chapter 4: Protein Interactions and Disease". In: *PLOS Computational Biology* 8.12 (Dec. 2012). PMID: 23300410; PMCID: PMC3531279, e1002819. DOI: 10.1371/journal.pcbi.1002819. URL: https://doi.org/10.1371/journal.pcbi.1002819.

[8] Javier De Las Rivas and Alberto De Luis. "Interactome data and databases: different types of protein interaction". In: *Comparative and Functional Genomics* 5.2 (2004). PMID: 18629107; PMCID: PMC2447402, pp. 173–178. DOI: 10.1002/cfg.377. URL: https://doi.org/10.1002/cfg.377.

[9] Attila Gursoy, Ozlem Keskin, and Ruth Nussinov. "Topological properties of protein interaction networks from a structural perspective". In: *Biochemical Society Transactions* 36.6 (Nov. 2008), pp. 1398–1403. ISSN: 0300-5127. DOI: 10.1042/BST0361398. eprint: https://portlandpress.com/biochemsoctrans/article-pdf/36/6/1398/852839/bst0361398.pdf. URL: https://doi.org/10.1042/BST0361398.

[10] J J Hopfield. "Neural networks and physical systems with emergent collective computational abilities." In: *Proceedings of the National Academy of Sciences* 79.8 (1982), pp. 2554–2558. DOI: 10.1073/pnas.79.8.2554. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.79.8.2554. URL: https://www.pnas.org/doi/abs/10.1073/pnas.79.8.2554.

[11] Maria Yampolskaya and Pankaj Mehta. "Hopfield Networks as Models of Emergent Function in Biology". In: *Annual Review of Biophysics* 55.Volume 55, 2026 (2026), pp. 323–342. ISSN: 1936-1238. DOI: https://doi.org/10.1146/annurev-biophys-101425-023356. URL: https://www.annualreviews.org/content/journals/10.1146/annurev-biophys-101425-023356.

[12] Snehashish Chakraverty, Deepti Moyi Sahoo, and Nisha Rani Mahato. "Hebbian Learning Rule". In: *Concepts of Soft Computing: Fuzzy and ANN with Programming*. Singapore: Springer Singapore, 2019, pp. 175–182. ISBN: 978-981-13-7430-2. DOI: 10.1007/978-981-13-7430-2_12. URL: https://doi.org/10.1007/978-981-13-7430-2_12.

[13] Google Creative Lab. *The Quick, Draw! Dataset*. https://github.com/googlecreativelab/quickdraw-dataset. Numpy bitmap dataset accessed via Google Cloud Storage: https://console.cloud.google.com/storage/browser/quickdraw_dataset. 2017. (Visited on 05/13/2026).

[14] Damian Szklarczyk et al. "The STRING database in 2025: protein networks with directionality of regulation". In: *Nucleic Acids Research* 53.D1 (Jan. 2025), pp. D730–D737. DOI: 10.1093/nar/gkae1113. URL: https://doi.org/10.1093/nar/gkae1113.

[15] STRING Consortium. *STRING Database: Mus musculus (house mouse) organism data, version 12.0*. https://version-12-0.string-db.org/organism/10090. Accessed: 2026-06-10. 2023.

[16] Vincent A Traag, Ludo Waltman, and Nees Jan van Eck. "From Louvain to Leiden: guaranteeing well-connected communities". In: *Scientific Reports* 9.1 (2019), p. 5233. DOI: 10.1038/s41598-019-41695-z. URL: https://doi.org/10.1038/s41598-019-41695-z.

[17] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. DOI: 10.1088/1742-5468/2008/10/P10008. URL: https://doi.org/10.1088/1742-5468/2008/10/P10008.

[18] Wulfram Gerstner et al. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[19] A. Bookstein, V. A. Kulyukin, and T. Raita. "Generalized Hamming distance". In: *Information Retrieval* 5 (2002), pp. 353–375. DOI: 10.1023/A:1020499411651.

[20] P.A Vijaya, M Narasimha Murty, and D.K Subramanian. "Leaders–Subleaders: An efficient hierarchical clustering algorithm for large data sets". In: *Pattern Recognition Letters* 25.4 (2004), pp. 505–513. ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2003.12.013. URL: https://www.sciencedirect.com/science/article/pii/S0167865503002824.

[21] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[22] Yipeng Song et al. "Principal component analysis of binary genomics data". In: *Briefings in Bioinformatics* 20.1 (Jan. 2019), pp. 317–329. ISSN: 1477-4054. DOI: 10.1093/bib/bbx119. eprint: https://academic.oup.com/bib/article-pdf/20/1/317/27689821/bbx119.pdf. URL: https://doi.org/10.1093/bib/bbx119.

[23] Andrew J. Landgraf and Yoonkyung Lee. "Dimensionality reduction for binary data through the projection of natural parameters". In: *Journal of Multivariate Analysis* 180 (2020), p. 104668. ISSN: 0047-259X. DOI: https://doi.org/10.1016/j.jmva.2020.104668. URL: https://www.sciencedirect.com/science/article/pii/S0047259X20302499.

[24] Daniel J. Amit, Hanoch Gutfreund, and H. Sompolinsky. "Storing Infinite Numbers of Patterns in a Spin-Glass Model of Neural Networks". In: *Phys. Rev. Lett.* 55 (14 Sept. 1985), pp. 1530–1533. DOI: 10.1103/PhysRevLett.55.1530. URL: https://link.aps.org/doi/10.1103/PhysRevLett.55.1530.

[25] Xinxin Zhang et al. "CellMarker: a manually curated resource of cell markers in human and mouse". In: *Nucleic Acids Research* 47.D1 (Jan. 2019), pp. D721–D728. DOI: 10.1093/nar/gky900. URL: https://doi.org/10.1093/nar/gky900.

[26] Congxue Hu et al. "CellMarker 2.0: an updated database of manually curated cell markers in human/mouse and web tools based on scRNA-seq data". In: *Nucleic Acids Research* 51.D1 (Oct. 2022), pp. D870–D876. ISSN: 0305-1048. DOI: 10.1093/nar/gkac947. eprint: https://academic.oup.com/nar/article-pdf/51/D1/D870/48440910/gkac947.pdf. URL: https://doi.org/10.1093/nar/gkac947.

A code: model and clustering functions

```
## Clustering algorithm functions

import numpy as np

def greedy_clustering_retrieved_states(final_states_retrieved, distance_threshold=0.15):

    states_matrix = np.array(final_states_retrieved, dtype=np.int8)
    num_runs, doubled_neurons = states_matrix.shape


    assigned = np.zeros(num_runs, dtype=bool)
    valley_counts = []

    cluster_labels = np.full(num_runs, -1, dtype=int)
    valley_id = 0

    for i in range(num_runs):
        if assigned[i]:
            continue

        seed_state = states_matrix[i]

        mismatches = (states_matrix != seed_state)
        hamming_distances = np.mean(mismatches, axis=1)

        in_same_valley = (hamming_distances <= distance_threshold) & (~assigned)

        runs_in_valley = np.sum(in_same_valley)
        valley_counts.append(runs_in_valley)

        cluster_labels[in_same_valley] = valley_id
        valley_id += 1
        assigned[in_same_valley] = True

    return cluster_labels



def two_pass_clustering_retrieved_states(final_states_retrieved, distance_threshold=0.15):

    states_matrix = np.array(final_states_retrieved, dtype=np.int8)
    num_runs, num_neurons = states_matrix.shape
```

```

    assigned_discovery = np.zeros(num_runs, dtype=bool)
    unique_seeds = []

    for i in range(num_runs):
        if assigned_discovery[i]:
            continue

        seed_state = states_matrix[i]
        unique_seeds.append(seed_state)

        mismatches = (states_matrix != seed_state)
        hamming_distances = np.mean(mismatches, axis=1)

        in_same_valley = (hamming_distances <= distance_threshold)
        assigned_discovery[in_same_valley] = True

    unique_seeds = np.array(unique_seeds, dtype=np.int8)
    num_valleys = len(unique_seeds)


    cluster_labels = np.full(num_runs, -1, dtype=int)

    for i in range(num_runs):
        run_state = states_matrix[i]

        mismatches = (unique_seeds != run_state)
        distances_to_seeds = np.mean(mismatches, axis=1)

        closest_valley_id = np.argmin(distances_to_seeds)

        if distances_to_seeds[closest_valley_id] <= distance_threshold
           :
            cluster_labels[i] = closest_valley_id

    return cluster_labels
```

```
## Pattern retrieval functions

from hopfield.train import error
import numpy as np
def pattern_retrieval_async(initial_neuron_states, weights,
   pattern_states, hamm_dist, steps=20, async_seed=42):
    states = np.copy(initial_neuron_states)
    hamm_history = []

    prev_hamm_dist = hamm_dist
    hamm_history.append(prev_hamm_dist)
```

```
    pattern_states.append(initial_neuron_states)

    if steps == 0:
        input_potentials = np.zeros(len(states))
        return states, input_potentials, hamm_history, 0, 0, []

    steps_done = 0
    error_count = 0
    prev_step_pattern = np.copy(initial_neuron_states)
    diff_count_array = []

    for curr_step in range(steps):
        steps_done = curr_step

        step_seed = async_seed + curr_step

        states, input_potentials = retrieval_step_asynchronous_seeded(
            states, weights, seed=step_seed)
        pattern_states.append(np.copy(states))

        diff_count, curr_hamm_dist = error.calc_hamming_error(
            prev_step_pattern=prev_step_pattern, step_pattern=states)
        diff_count_array.append(diff_count)
        hamm_history.append(curr_hamm_dist)

        if curr_hamm_dist <= 0.01 and curr_hamm_dist <= prev_hamm_dist
            :
            if error_count == 4:
                error_count += 1
                break
            else:
                error_count += 1
        else:
            error_count = 0

        prev_hamm_dist = curr_hamm_dist
        prev_step_pattern = np.copy(states)

    return states, input_potentials, hamm_history, (steps_done+1),
        error_count, diff_count_array

def retrieval_step_asynchronous_seeded(states, weights, seed):
    N = len(states)

    local_states = np.copy(states)

    rng = np.random.default_rng(seed)

```

```
    neuron_order = rng.permutation(N)

    for i in neuron_order:
        input_potential_i = np.dot(weights[i], local_states)

        if input_potential_i > 0:
            local_states[i] = 1
        elif input_potential_i < 0:
            local_states[i] = -1

    input_potentials = np.dot(weights, local_states)

    return local_states, input_potentials
```

```
    ## Retrieval error function

    import numpy as np

    def calc_hamming_error(prev_step_pattern, step_pattern):
    curr_hamm_dist = np.mean(prev_step_pattern != step_pattern)

    diff_count = np.sum(prev_step_pattern != step_pattern)
    return diff_count ,curr_hamm_dist
```

B code: data processing functions

```
    ## Read STRING db datafile function

    import pandas as pd
    import numpy as np

    zip = "{path}/10090.protein.links.full.v12.0.txt.gz"

    def read_protein_links(zip, combined_score=700):

    chunks = []

    reader = pd.read_csv(zip, sep="␣", usecols=[0, 1, 13], chunksize
       =250000)

    for chunk in reader:
        chunk.columns = ['p_i', 'p_j', 'combined_score']

        filtered = chunk[chunk['combined_score'] >= combined_score]
        chunks.append(filtered)

```

```
    protein_links = pd.concat(chunks)

    return protein_links
```

C code : plotting functions

```
    ## Cluster line plot
    def linear_plot_greedy_clusters(cluster_labels, distance_threshold
        ):

    assigned_runs = cluster_labels[cluster_labels >= 0]
    valley_counts = np.bincount(assigned_runs)

    valley_counts = valley_counts[valley_counts > 0]

    sorted_valley_sizes = sorted(valley_counts, reverse=True)
    total_clusters = len(sorted_valley_sizes)

    plt.figure(figsize=(12, 5))
    plt.plot(sorted_valley_sizes, color='#2c3e50', linewidth=2, marker
        ='o', markersize=3)

    padding = max(5, int(total_clusters * 0.05))
    plt.xlim(-padding, total_clusters + padding)

    standard_ticks = np.linspace(0, total_clusters, 6, dtype=int)
    plt.xticks(standard_ticks, labels=[str(t) for t in standard_ticks
        ])

    plt.title(f"Clusters found for {distance_threshold*100}% Hamming
        distance threshold", fontsize=14, pad=12)
    plt.xlabel("Cluster Rank - Sorted by size", fontsize=11)
    plt.ylabel("Number of Runs in Cluster", fontsize=11)
    plt.grid(True, linestyle='--', alpha=0.5)

    plt.tight_layout()
    plt.show()

def linear_plot_two_steps_cluster(cluster_labels, distance_threshold):
    assigned_runs = cluster_labels[cluster_labels >= 0]
    valley_counts = np.bincount(assigned_runs)

    valley_counts = valley_counts[valley_counts > 0]

    sorted_valley_sizes = sorted(valley_counts, reverse=True)
    total_clusters = len(sorted_valley_sizes)
```

```
37
38      plt.figure(figsize=(12, 5))
39      plt.plot(sorted_valley_sizes, color='#2c3e50', linewidth=2, marker
            ='o', markersize=3)
40
41      padding = max(5, int(total_clusters * 0.05))
42      plt.xlim(-padding, total_clusters + padding)
43
44      standard_ticks = np.linspace(0, total_clusters, 6, dtype=int)
45      plt.xticks(standard_ticks, labels=[str(t) for t in standard_ticks
            ])
46
47      plt.title(f"Clusters found for {distance_threshold*100}% Hamming
            distance threshold", fontsize=14, pad=12)
48      plt.xlabel("Cluster Rank - Sorted by size", fontsize=11)
49      plt.ylabel("Number of Runs in Cluster", fontsize=11)
50      plt.grid(True, linestyle='--', alpha=0.5)
51
52      plt.tight_layout()
53      plt.show()
```

```
1       ## Hamming error convergence plot
2
3       def hamming_error_evolution_plot(all_patterns_hamm_history_trained
            , steps):
4
5       plt.figure(figsize=(8, 4))
6       for i in range(len(all_patterns_hamm_history_trained)):
7           if(all_patterns_hamm_history_trained[i][-1] <= 0.01):
8               plt.plot(np.arange(len(all_patterns_hamm_history_trained[i
                    ][1:])), all_patterns_hamm_history_trained[i][1:], alpha
                    =0.5)
9       plt.axhline(0.01, linestyle = "--")
10      plt.xticks(np.arange(0, steps, step=5))
11      plt.yticks(np.linspace(0, 1, 11))
12      plt.ylabel("Hamming error")
13      plt.xlabel("Time")
14      plt.title("Error Convergence for Hamming distance - Cases
            converged at <= 0.01")
15      plt.show()
16
17      plt.figure(figsize=(8, 4))
18
19      for i in range(len(all_patterns_hamm_history_trained)):
20          plt.plot(np.arange(len(all_patterns_hamm_history_trained[i
                ][1:])), all_patterns_hamm_history_trained[i][1:], alpha
                =0.5)
21      plt.axhline(0.01, linestyle = "--")
```

```
    plt.xticks(np.arange(0, steps, step=5))
    plt.yticks(np.linspace(0, 1, 11))
    plt.ylabel("Hamming distance")
    plt.xlabel("Time")
    plt.title("Convergence for Hamming distance")
    plt.show()
```

```
    ## PCA 3D clusters plot

    import matplotlib.pyplot as plt
    import numpy as np
    from sklearn.decomposition import PCA
    def PCA_3D_plot_two_pass_clusters(cluster_labels, states_matrix,
       num_clusters):

    assigned_runs = cluster_labels[cluster_labels >= 0]
    cluster_counts = np.bincount(assigned_runs, minlength=num_clusters
       )

    unique_labels = [l for l in np.unique(cluster_labels) if l != -1]
    cluster_counts_dict = {l: cluster_counts[l] for l in unique_labels
       }

    sorted_labels = sorted(unique_labels, key=lambda x:
       cluster_counts_dict[x], reverse=True)
    sorted_counts = [cluster_counts_dict[l] for l in sorted_labels]

    num_individual_legend_clusters = min(30, len(sorted_labels))
    top_labels_set = set(sorted_labels[:num_individual_legend_clusters
       ])

    minor_mask = np.array([l not in top_labels_set for l in
       cluster_labels])
    total_minor_runs = np.sum(minor_mask)

    print("Running 3D PCA dimensionality reduction...")
    pca = PCA(n_components=3, random_state=42)
    embedding = pca.fit_transform(states_matrix)

    fig = plt.figure(figsize=(13, 10))
    ax = fig.add_subplot(111, projection='3d', computed_zorder=False)

    bg_handle = ax.scatter(
        embedding[minor_mask, 0], embedding[minor_mask, 1], embedding[
           minor_mask, 2],
        c='#d1d5db', s=8, alpha=0.25, label='Other Minor Clusters / 
           Noise',
        zorder=1
```

```
    )
    cmap_base = plt.colormaps['turbo'].resampled(
        num_individual_legend_clusters)
    distinct_colors = [cmap_base(i) for i in range(
        num_individual_legend_clusters)]

    legend_handles = []
    legend_texts = []

    placed_positions = []

    for idx, label in enumerate(sorted_labels[:
        num_individual_legend_clusters]):
        mask = (cluster_labels == label)

        if np.sum(mask) == 0:
            continue

        label_text = f"Cluster {label} : {sorted_counts[idx]}"
        color = distinct_colors[idx]

        scatter_handle = ax.scatter(
            embedding[mask, 0], embedding[mask, 1], embedding[mask,
                2],
            c=[color], s=40, alpha=0.7,
            edgecolors='black', linewidths=0.3,
            zorder=2
        )
        legend_handles.append(scatter_handle)
        legend_texts.append(label_text)

        cluster_points = embedding[mask]
        centroid_x = np.mean(cluster_points[:, 0])
        centroid_y = np.mean(cluster_points[:, 1])
        centroid_z = np.mean(cluster_points[:, 2])

        offset_scale = 3.5  # Controls how far outward pointer labels
            stretch
        norm = np.sqrt(centroid_x**2 + centroid_y**2)
        if norm > 0:
            final_x = centroid_x + (centroid_x / norm) * offset_scale
            final_y = centroid_y + (centroid_y / norm) * offset_scale
        else:
            final_x, final_y = centroid_x, centroid_y
        final_z = centroid_z

        min_z_distance = 1.3  # Minimum vertical space required
            between labels
```

```
        min_xy_distance = 3.5 # Minimum Euclidean 2D space required between labels

        collision = True
        attempts = 0
        while collision and attempts < 20:
            collision = False
            for px, py, pz in placed_positions:
                xy_dist = np.sqrt((final_x - px)**2 + (final_y - py)**2)
                z_dist = abs(final_z - pz)


                if xy_dist < min_xy_distance and z_dist < min_z_distance:
                    final_z += 1.2  # Shift step size
                    collision = True
                    break
            attempts += 1

        placed_positions.append((final_x, final_y, final_z))

        ax.plot([centroid_x, final_x],
                [centroid_y, final_y],
                [centroid_z, final_z],
                color='black', linewidth=0.6, alpha=0.4,
                zorder=4)

        ax.text(
            final_x, final_y, final_z,
            f"{label}",
            fontsize=8,
            fontweight='bold',
            color='black',
            ha='center', va='center',
            zorder=5,
            bbox=dict(boxstyle="round,pad=0.15", fc="white", ec="black", alpha=0.8, lw=0.5)
        )

    legend_handles.append(bg_handle)
    legend_texts.append(f"Other Minor Clusters : {total_minor_runs}")

    ax.set_title(f"3D PCA Topology - Display of {len(unique_labels)} clusters", fontsize=14, pad=20)
    ax.set_xlabel("Principal Component 1", fontsize=10, labelpad=10)
    ax.set_ylabel("Principal Component 2", fontsize=10, labelpad=10)
    ax.set_zlabel("Principal Component 3", fontsize=10, labelpad=10)
```

```
119
120     ax.view_init(elev=25, azim=35)
121
122     plt.legend(
123         legend_handles, legend_texts,
124         loc="upper left",
125         bbox_to_anchor=(1.05, 0.95),
126         title="Attractor Rank & Sizes",
127         frameon=True,
128         ncol=1,
129         fontsize=9,
130         title_fontsize=11
131     )
132
133     plt.tight_layout()
134     plt.savefig("pca_3d_two_step_offset_labels.png", dpi=300,
        bbox_inches='tight')
135     plt.show()
```

D code: model workflow

```
1     N_initial_states = 5000
2     unique_states = set()
3
4     rng = np.random.default_rng(seed=42)
5
6     while len(unique_states) < N_initial_states:
7         state = tuple(rng.choice([-1, 1], size=num_neurons))
8         unique_states.add(state)
9
10    initial_states_matrix = np.array(list(unique_states))
11
12    all_patterns_hamm_history_trained = []
13    final_states_trained = []
14    steps=100
15
16    for index, initial_state in enumerate(initial_states_matrix):
17
18        pattern_states = []
19
20        retrieved_pattern, _, hamm_history, _, _, _ = retrieval.
          pattern_retrieval_async(
21                initial_state,
22                raw_weight_matrix,
23                pattern_states,
24                hamm_dist=0,
```

```
                steps=steps,
                async_seed=42
            )
        all_patterns_hamm_history_trained.append(hamm_history.copy())
        final_states_trained.append(np.copy(pattern_states[-1]))
```