Fundamentos Teóricos y Marcos Algorítmicos para Computación Neuromórfica y Optimización de Grafos

Theoretical Foundations and Algorithmic Frameworks for Neuromorphic Computing and Graph Optimization

Máster en Métodos Formales en Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid & Universidad Politécnica de Madrid

Autor: Álvaro Sánchez-Paniagua Ríos

Tutor: Ismael Rodríguez Laguna

Colaboradores externos: Samuel López Asunción & Pablo Ituero

Septiembre 2025

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mi tutor, el Dr. Ismael Rodríguez Laguna, por su entusiasmo al adentrarse conmigo en este nuevo terreno y por su vocación investigadora.

A mis colaboradores externos, Samuel López Asunción y Pablo Ituero, por acompañarme en este proyecto y trabajar codo a codo conmigo.

A Gonzalo, Miguel e Iñigo, así como a todos mis compañeros del máster, por su ayuda, compañía y por hacer de esta etapa una experiencia enriquecedora y divertida.

A mis amigos y a Adrián Cámara Ballesteros por su apoyo incondicional.

Y, por supuesto, a mi familia, por estar siempre presente.

Abstract

As Moore's Law nears its limits, neuromorphic computing, exemplified by chips like Intel's Loihi and IBM's TrueNorth, emerges as a bioinspired alternative to von Neumann architectures. By mimicking the brain's event-driven, massively parallel processing, neuromorphic systems offer promising gains in energy efficiency and latency, particularly for graph-based workloads.

We begin with a comprehensive literature review to establish a theoretical foundation for neuromorphic computing and then propose a simple iteration-based spiking neural network (SNN) algorithm to solve the shortest path problem (SPP) in $\mathcal{O}(h|E|)$ time, where h is the length of the shortest path and E represents the graph edges. Building on the work of Schuman et al., we extend their method and develop two key issues: (1) evaluating the efficiency and complexity of neuromorphic approaches relative to classical baselines (e.g., Dijkstra's algorithm), and (2) addressing the fragmentation and lack of methodological transparency in current neuromorphic graph processing research.

Our simulations show that our proposed algorithm outperforms both Dijkstra and prior SNN-based models on large, dense graphs, achieving up to 228× energy savings. However, its performance is limited by path length, sparsity, and edge weight distribution. To address this, we evaluate two synaptic delay compression techniques, value mapping and logarithmic scaling, that reduce execution cost at the expense of precision. Results suggest that mapping offers the best trade-off for scalable, energy-efficient neuromorphic graph computation.

Keywords: neuromorphic computing; SNNs, SPP; Dijkstra's algorithm; graph problems; energy efficiency; synaptic delay compression; algorithmic complexity.

Resumen

A medida que la Ley de Moore se acerca a sus límites, la computación neuromórfica, ejemplificada por chips como Intel Loihi e IBM TrueNorth, se perfila como una alternativa bioinspirada a las arquitecturas de von Neumann. Al imitar el procesamiento por eventos y la dinámica masivamente paralela del cerebro, estos sistemas prometen mejoras notables en eficiencia energética y latencia, especialmente en tareas basadas en grafos.

Comenzamos con una revisión bibliográfica exhaustiva para sentar las bases teóricas de la computación neuromórfica, y luego proponemos un algoritmo sencillo basado en iteraciones para una red neuronal de picos (SNN) que resuelve el problema del camino más corto (SPP) en tiempo $\mathcal{O}(h|E|)$, donde h es la longitud del camino más corto y E representa las aristas del grafo. Partiendo del trabajo de Schuman et al., extendemos su enfoque y señalamos dos retos clave: (1) evaluar la eficiencia de los métodos neuromórficos frente a los clásicos (por ejemplo, Dijkstra), y (2) abordar la fragmentación y la falta de transparencia metodológica en la investigación actual de optimización de grafos con computación neuromórfica.

Nuestras simulaciones muestran que nuestro algoritmo propuesto supera tanto a Dijkstra como a modelos previos basados en SNN en grafos grandes y densos, alcanzando ahorros energéticos de hasta 228×. Sin embargo, su rendimiento se ve limitado por la longitud de los caminos, la esparsidad y la distribución de los pesos de las aristas. Para abordar estas limitaciones, evaluamos dos técnicas de compresión de retardos sinápticos; mapping y escalado logarítmico, que reducen el coste de ejecución a costa de la precisión. Donde mostramos que el mapeo ofrece el mejor compromiso entre eficiencia ganada y precisión perdida.

Palabras clave: computación neuromórfica; SNNs; SPP; algoritmo de Dijkstra; eficiencia energética; compresión de retardos sinápticos; mapeo de valores; escalado logarítmico; algoritmos de grafos; modelado de hardware.

Contents

1. Introduction **6**
1.1. Objectives . . . 7
1.2. Summarized Methodology . . . 8

2. The Essence of Neuromorphic Systems **9**
2.1. Biophysical Principles as a Computational Foundation . . . 9
2.2. Neuromorphic Architectures: Design and Formal Foundations . . . 10
2.3. Information Encoding and Representation . . . 13
2.3.1. Hyperdimensional Computing . . . 14
2.3.2. Sub-symbolism and fuzzy logic . . . 16
2.3.3. Non-linear oscillators . . . 17
2.4. Computational Properties . . . 18
2.4.1. Turing-Completeness and Expressive Power . . . 18
2.4.2. Algorithmic Complexity . . . 24
2.5. Use Cases . . . 26
2.5.1. Spiking Neuronal Networks: Deep learning . . . 27
2.5.2. Graph Problems . . . 27
2.5.3. Constraint Satisfaction Problems . . . 31
2.6. Neuromorphic Computing: Strengths and Limitations . . . 34

3. Our Spiking Neural Network for Graph Problem Solving **35**
3.1. Functional level . . . 37
3.1.1. Time complexity . . . 40
3.1.2. Spatial Complexity . . . 47
3.2. Neuromorphic level . . . 47
3.2.1. Software Simulations . . . 48
3.2.2. Weight compression methods . . . 53
3.2.3. Hypotheses . . . 56

4. Experimental methodology **57**
4.1. Graph Generation . . . 57
4.2. Estimations . . . 58
4.2.1. Time estimates . . . 58
4.2.2. Energy consumption estimates . . . 61
4.3. Performance experiments . . . 63
4.4. Energy experiments . . . 64
4.5. Weight compression tests . . . 64

5. Results and discussion **65**
5.1. Performance results 65
5.2. Energy results 77
5.3. Weight compression results 78

6. Conclusions and Future Work **81**

References **84**

A. Graphical Notation for Spiking Neural Networks **90**
A.1. Representation and neuron model 90

B. First Experiment: Logic Gates **90**
B.1. AND Gate (∧) 91
B.2. OR Gate (∨) 91
B.3. NOT Gate (¬) 92
B.4. Implication Gate (→) 93

C. Modeling **94**
C.1. Hamiltonian Cycle Algorithm 96
C.2. Algorithmic Complexity 98
C.3. TSP Algortihm 99

1. Introduction

As Moore's Law approaches its physical and energetic limits, the research community has been driven to explore alternative computational paradigms that overcome the energy and architectural constraints inherent to classical systems. In this context, leading technological actors such as Intel and IBM have pioneered the development of neuromorphic chips (e.g., Loihi and TrueNorth) [16, 54], bioinspired architectures that emulate the efficiency and flexibility of the human brain through the massive use of artificial spiking neurons and synapses. These systems are fundamentally distinguished by their unconventional operational model, in which the traditional separation between memory and processing, a hallmark of von Neumann architecture, is replaced by large-scale asynchronous parallelism. It is through the dynamic interaction of multiple elementary computational units that processing capability emerges.

Recent studies [29, 37, 54] have demonstrated not only the superior energy efficiency of these neuromorphic technologies but also their ability to achieve Turing completeness [14], positioning them as viable candidates for future universal computing. However, a critical gap persists: the lack of a formal framework. This deficiency severely limits both the scalability and adoption of neuromorphic technologies in larger-scale applications.

Programming such systems requires manipulating spiking neuron and synapse models at a highly elemental level, akin to coding in machine language. The absence of conceptual abstractions forces constant reinvention in each project, fragmenting knowledge and hindering the integration of advances into a unified theoretical body. Indeed, the evaluation of neuromorphic systems has largely focused on specific tasks, such as traditional deep learning, constraint satisfaction, or graph-based problems [33, 36, 43, 49, 52, 63] without an underlying theory to explain their operational mechanisms or enable the generalization of their successes.

This approach starkly contrasts with classical computing, where solid theoretical foundations enable the systematic design of efficient algorithms. In neuromorphic computing, the lack of consensus on what is neuromorphic, or even appropriate performance metrics, reveals an identity crisis in the field.

The practical implications of this theoretical gap are significant. The absence of formalisms prevents guaranteeing the robustness of neuromorphic systems outside ad-hoc approaches, while the lack of programmable abstractions restricts developers' ability to build complex applications without being overwhelmed by low-level details. Consequently, every hardware or software advancement risks becoming a dead end rather than a step toward an integrated, scalable system.

This work is grounded in the hypothesis that identifying, or at least proposing, a coherent set of principles interconnecting biology, theory, and engineering could provide the foundation needed to overcome the current fragmented state of neuromorphic computing. The aim is not to immediately resolve all challenges in the field but rather to organize the current landscape.

1.1. Objectives

This master thesis is divided into two distinct parts. The first part is a study of the state of the art in the emerging paradigm of neuromorphic computing. We begin by introducing the neuroscientific foundations that inspire this computational model, focusing on the behavior of neurons and synapses. We then explore how information is encoded in spiking neural systems and review complementary technologies that seek to harness the power of this paradigm, like hyperdimensional computing and frameworks for modeling neural dynamics such as nonlinear oscillators.

Once the operational principles of neuromorphic systems are understood, we move on to their computational properties. In particular, we examine neuromorphic architectures as Turing-complete systems, and analyze proposals for characterizing algorithmic, spatial, and temporal complexity within this new framework.

Finally, we review the most prominent use cases of neuromorphic computing to develop an intuition for how programming is approached in such systems. We conclude with a discussion of the main advantages and limitations of neuromorphic computation.

The second part of this work comprises its practical section, where we introduce our own algorithm for solving graph problems. We model and characterize this algorithm both at a high-level behavioral description and through Python simulations. This allows us to critically examine the nature of neuromorphic computing and assess its practical viability. We illustrate how a neuromorphic framework can be adapted to a given heuristic and compare our approach with other heuristics proposed in the literature addressing the same problem.

We analyze the advantages that neuromorphic systems may offer in terms of energy efficiency and execution time. This analysis is supported by a detailed study grounded in the temporal and spatial complexity considerations discussed in following sections.

More specifically, the objectives of this work are as follows:

1. To review the existing literature on neuromorphic computing, developing a solid intuition for its operational principles and algorithmic paradigms.

2. To design and implement a novel neuromorphic algorithm to solve the Shortest Path Problem (SPP) using the identified baselines as a point of reference.
3. To compare and characterize the proposed model against both classical and state-of-the-art neuromorphic algorithms, with particular emphasis on energy efficiency and execution time.

Thus, the overarching goal of this work is to critically examine the current landscape of neuromorphic computing, highlighting its most advanced developments, while identifying key limitations that must be addressed to unlock the full potential of this computational paradigm.

To this end, we propose and implement a new neuromorphic algorithm that not only aims to clarify the ambiguities surrounding this still loosely defined field, but also provides a rigorous and practical methodology for its deployment. This includes a thorough theoretical justification, an explanation of key design choices, and insightful comparisons with other approaches found in the literature.

By bridging the gap between abstract theoretical models and their practical implementations, our contribution seeks to add tangible value to the field. In particular, we extend the intuition found in recent academic works into a fully realized algorithmic proposal, subjecting it to detailed analysis and experimental validation, while also deepening the theoretical foundations of neuromorphic principles.

1.2. Summarized Methodology

After reviewing the relevant literature, we will proceed to implement our proposed algorithm in a complete end-to-end fashion as we previously mencioned. The implementation will be evaluated through software simulations

To objectively assess different approaches to solving graph-related problems, more specifically, the SPP, we will also implement two additional solutions: one based on a neuromorphic computing method drawn from recent literature, and another implementing the classical Dijkstra algorithm. These will be developed under the same experimental conditions as our proposal to ensure a fair comparison.

We will conduct a comprehensive analysis from three perspectives: algorithmic functionality, neuromorphic behavior, and temporal/energy performance. This will allow us to characterize all three approaches and highlight any relative advantages. In support of the theoretical premises developed, we will design and execute a series of experiments aimed at validating them.

Software Simulations

A large set of synthetic graphs with diverse topological properties will be used to cover, as broadly as possible, the graph space. This will allow us to draw meaningful conclusions regarding the performance of each approach.

Each of the three algorithms will be executed across the entire dataset, and their execution times will be estimated based on those simulations. We will analyze the graph types for which our approach is (or is not) competitive compared to the classical and neuromorphic baselines.

Advanced statistical analysis techniques will be employed, such as Principal Component Analysis and Partial Correlations, to identify patterns in the results. In addition, we will compute key statistical indicators such as the percentage of graphs for which our method shows performance improvement, inter-quartile ranges, and median gains.

Finally, we will revisit and confront our initial hypotheses in light of the experimental results, ensuring a detailed understanding of the behavior of our algorithm and its counterparts from the literature.

2. The Essence of Neuromorphic Systems

This section serves as both an introduction and a conceptual development of the neuromorphic computing paradigm. We begin by outlining the fundamental mechanisms that underpin the systems we will later utilize. Then, more specifically and with a focus on our field of interest, we draw upon the existing literature to define the key computational properties of neuromorphic systems, along with their most prominent use cases. These foundational insights will later serve as both inspiration and theoretical grounding for the practical section of this work.

2.1. Biophysical Principles as a Computational Foundation

The brain is one of the most efficient biological computing machines, achieving remarkable computational power with minimal energy consumption, just a few watts. This natural system challenges our traditional understanding of computation and inspires us to study its principles in order to replicate them artificially.

It consists of approximately 10^{11} neurons interconnected by around 10^{15} synapses. These neurons are the brain's fundamental computational units, and understanding how they function is essential for the development of biologically inspired computational models.

Each neuron has a characteristic structure, consisting of a cell body called the soma,

from which two main types of branches emerge: dendrites and axons. Dendrites act as receptors, collecting electrical impulses from other neurons, while the axon is the extension responsible for transmitting the signal to other neurons through synapses.

Neuronal communication is based on electrical impulses. When the incoming signal received by a neuron surpasses a certain threshold, the neuron fires and generates an action potential that travels along the axon to the synapses, where the signal is transmitted to downstream neurons. Synapses serve not only as connection points but also play a crucial role in modulating the signal: they can amplify or inhibit the activation of the postsynaptic neuron, and over time, they can strengthen or weaken depending on neuronal activity. These adaptive and learning properties enable the brain to process information flexibly and efficiently, dynamically adjusting to environmental stimuli.

However, this represents a profound paradigm shift, from the sequential programming model of the Turing machine to a radically different computational framework. In a Turing machine, information is encoded on a tape using simple bits, 0s and 1s. Classical computation, grounded in this model, is inherently axiom-driven, with well-defined rules yielding discrete systems in which all interactions can be interpreted as rules applied sequentially, orderly, and synchronously.

In contrast, modeling the brain as a computational system requires a fundamentally different approach. Unlike classical computation, the brain operates in a distributed and nonlinear manner. Furthermore, unlike the Von Neumann architecture, where memory and processing are separate, neuromorphic systems integrate both processes within the neurons themselves, opening new possibilities for the design of bio-inspired hardware and software.

The interest in brain-inspired computing stems from multiple motivations. From a theoretical standpoint, exploring this paradigm allows us to challenge the boundaries of traditional computation and investigate whether alternative models offer superior capabilities in specific contexts. Practically, emulating the brain's energy efficiency and processing power could revolutionize artificial intelligence, robotics, and medicine, with applications ranging from brain-machine interfaces to personalized treatments in neuroscience.

2.2. Neuromorphic Architectures: Design and Formal Foundations

Neuromorphic computing is grounded in the biological principle of non-determinism, operating through massively distributed and parallelized processes. To enable this, specialized chips are designed that decentralize both computation and information storage into the

synaptic connections between artificial neurons. These synapses are not only responsible for transmitting signals but also possess learning capabilities through a mechanism we will explore later in this section, known as synaptic plasticity, or more specifically, spike-timing-dependent plasticity (STDP).

Before delving into synaptic learning, we must first describe how neurons, the fundamental computational units in this paradigm, are modeled. The complexity of a neuron model can vary significantly depending on the level of biological fidelity desired. We begin with the simplest and most widely used spiking neuron models: the *Integrate-and-Fire (IF)* and the *Leaky Integrate-and-Fire (LIF)* neurons. These models are conceptually similar and based on nearly the same principles.

Neurons maintain a certain membrane potential. When this internal electrical potential surpasses a defined threshold, the neuron emits a discrete electrical signal known as a spike. The dynamics of IF/LIF neurons are governed by the following differential equation:

$$\tau \frac{du(t)}{dt} = -[u(t) - u_{\text{rest}}] + RI(t) \tag{1}$$

Here, $u(t)$ represents the membrane potential of the neuron, u_{rest} is the resting potential (or leak), R is the membrane resistance, $I(t)$ is the input current, and τ is the membrane time constant. When the neuron fires, its membrane potential drops sharply, entering a recovery phase. During this refractory period, the neuron becomes temporarily unresponsive to further stimulation.

Once a neuron's membrane potential reaches its threshold value, that is, when $u(t) = u_{\text{th}}$, a spike is emitted (i.e. it fires). At that point, the membrane potential is immediately reset either to a resting value u_{rest}, or by subtracting a large value, typically the threshold itself.

Synapses, on the other hand, are associated with weights w_{ij}, which scale the incoming spikes, and may introduce a synaptic delay δ_{ij}, affecting the transmission time of spikes from the presynaptic to the postsynaptic neuron.

If we discretize and solve the differential equation (1), the membrane time constant τ is mapped to a decay factor λ, yielding the following iterative update rule:

$$u_i^t = \lambda u_i^{t-1} + \sum_j w_{ij} o_j^t - u_{\text{th}}\, o_i^{t-1} \tag{2}$$

where o_i^{t-1} denotes the output spike of neuron i at time $t-1$:

$$o_i^{t-1} = \begin{cases} 1, & \text{if } u_i^{t-1} > u_{\text{th}} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Note that, in equation (2), all inputs from connected neurons are integrated according to their synaptic weights. When the neuron spikes, the final term subtracts u_{th} to the current potential. This reset-by-subtraction mechanism, as used in [50], is argued to retain more information and improve the performance of neuromorphic networks.

The LIF model extends the IF model by introducing a *leak term* that causes the membrane potential to decay over time in the absence of input. Thus, the IF model can be seen as a special case of the LIF model where the leak constant is set to zero.

The decay factor λ, with $\lambda < 1$, models the so-called leak, gradually reducing the potential over time in the absence of input. This results in a simple yet effective neuron model. Although it does not fully capture biological realism, it is more than sufficient for building useful computational primitives in neuromorphic systems.

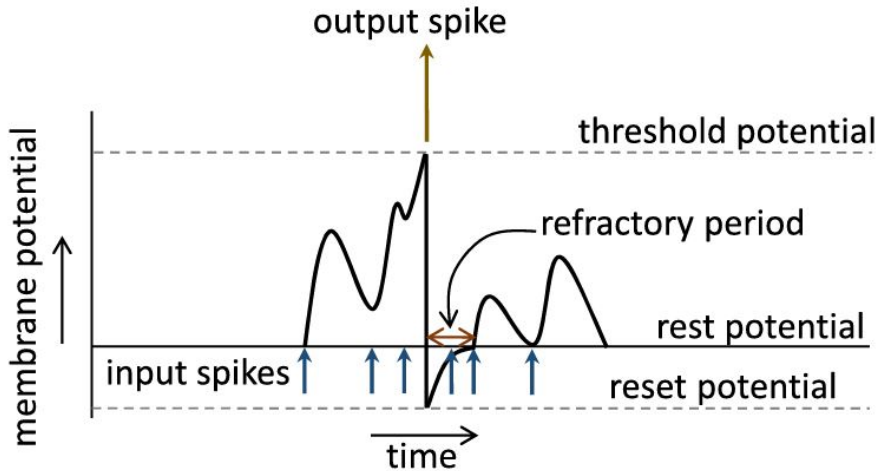


Figure 1: Dynamics of the LIF neuron. Figure extracted from [50].

There exist other types of neuron models, such as stochastic neurons or those inspired by the Hodgkin-Huxley formalism, which offer higher biological fidelity. However, we restrict ourselves to this simpler model, which retains an electrical potential up to a given threshold and leaks over time, providing a practical balance between simplicity and expressive power.

Another highly relevant mechanism in biological synapses is their inherent learning capability, known as Spike-Timing-Dependent Plasticity (STDP). This mechanism allows

synapses to modify their weights in real-time, making it particularly useful for tasks in deep learning and unsupervised learning, as we will see in later sections. The weight modification is typically described by the following rule:

$$\Delta w = \begin{cases} A_+ e^{-\frac{\Delta t}{\tau_+}}, & \text{if } \Delta t = t_{\text{post}} - t_{\text{pre}} > 0 \\ -A_- e^{\frac{\Delta t}{\tau_-}}, & \text{if } \Delta t = t_{\text{post}} - t_{\text{pre}} < 0 \end{cases} \tag{4}$$

Here, t_{pre} and t_{post} represent the spike times of the pre- and post-synaptic neurons, respectively. A_+ and A_- are learning rates, while τ_+ and τ_- are time constants that control how weights change over time.

If the postsynaptic neuron fires shortly after the presynaptic neuron, we have $\Delta t = t_{\text{post}} - t_{\text{pre}} > 0$. This temporal ordering suggests that the presynaptic spike contributed to the activation of the postsynaptic neuron. The closer in time the two spikes occur, the stronger the synaptic potentiation. This effect is captured by the exponential term: when $\Delta t \approx 0$, we get $e^{-\frac{\Delta t}{\tau_+}} \approx 1$, and thus $\Delta w \approx A_+$, resulting in a significant increase in synaptic strength. Conversely, if Δt is large, the potentiation effect diminishes exponentially.

On the other hand, if the postsynaptic neuron fires *before* the presynaptic neuron, then $\Delta t < 0$, indicating that the presynaptic spike did not causally contribute to the postsynaptic activation. In this case, the synapse is weakened through a similar mechanism: $\Delta w = -A_- e^{\frac{\Delta t}{\tau_-}}$, resulting in synaptic depression whose magnitude also decays exponentially with the temporal difference.

There are also multiple ways to model or refine this biologically inspired mechanism, such as Short-Term Plasticity (STP) or Long-Term Potentiation (LTP).

While STDP has proven to be a powerful and biologically validated learning mechanism [22, 55, 57], it may also introduce stability issues in networks and does not fully resolve the phenomenon known as 'catastrophic forgetting', where neural systems entirely forget previously learned tasks upon learning new ones [48].

2.3. Information Encoding and Representation

In this computational paradigm, memory and processing are integrated into the elementary neural units described earlier. However, information encoding can adopt multiple approaches. The most intuitive classical method involves discretizing the system by representing information through binary pulses (+1 or −1). This is fundamentally an event-driven model that leverages the asynchronous temporal dimension of neuromorphic computing to translate analog and continuous inputs (e.g., images with pixels or real-valued

numbers) into discrete spike-based events.

Two primary encoding strategies are widely studied in neuromorphic systems [2, 50]:

- **Rate Encoding:** This method calculates the average spike frequency of a neuron (or neuronal population) over a defined time window. While simple to implement, rate encoding requires multiple iterations to refine errors and improve precision. Its inefficiency stems from the fact that individual spikes carry minimal information per event. Mathematically, for a neuron i, the rate r_i is given by:

  $$r_i = \frac{N_{\text{spikes}}}{T_{\text{window}}}$$

  where N_{spikes} is the spike count and T_{window} is the observation period. This approach is often extended to populations of neurons or even measured over many simulations to enhance representational capacity.

- **Temporal Encoding:** Contrary to rate encoding, temporal encoding prioritizes the precise *timing* of spikes as the primary information carrier. This method enables high-density data representation and aligns closely with biological plausibility. Common variants include:

  - *Time-To-First-Spike:* Encodes stimulus intensity into the latency of the first spike , stronger stimuli cause the neuron to fire earlier.
  - *Rank Order Encoding:* Represents information based on the relative firing order of neurons in a population in response to a given input.
  - *Inter-Spike Interval (ISI):* Encodes information through the timing differences between successive spikes, either within a single neuron or across neuronal populations.
  - *Correlation and Synchrony:* Information is represented by the temporal alignment of spikes across groups of neurons. When neurons fire simultaneously, this pattern can signal the presence of specific input features. This can occur in a binary fashion or through selective activation of sparse neuronal subsets, where only a few neurons spike per input pattern.

2.3.1. Hyperdimensional Computing

To leverage parallelism and the massive distribution of information inherent in neuromorphic computing, several paradigmatic symbioses have emerged. In this context, hyperdimensional computing and spiking neural networks work in tandem to represent in-

formation in a distributed fashion with built-in error tolerance, thanks to the inherent parallelism of neuromorphic architectures.

Recent studies, though not yet widely recognized, indicate promising efforts to exploit the computational capabilities of these neural systems. Neural paradigms have shown significant advances in deep learning and cognitive tasks [64].

Hyperdimensional computing is motivated by the desire to understand how the brain processes information across an immense number of dimensions. In this paradigm, data are modeled as hypervectors in a high-dimensional space.

As previously noted, the brain contains a massive number of neurons and synapses. This creates a system where information processing is highly distributed, tolerant to errors, and operates at low precision, an approach that lies at the heart of hyperdimensional computing.

The notion of distributed information in high-dimensional spaces has gained prominence under the term *Vector Symbolic Architectures* (VSAs), which essentially provide a systematic framework for manipulating such types of information. This computational paradigm first maps data into a high-dimensional representation via a random embedding function $\phi : \mathcal{X} \to \mathcal{H}$.

The space $\mathcal{H}$ is a d-dimensional inner product space defined over the real numbers. These randomly generated vectors take advantage of the *concentration of measure* phenomenon, which states that, with high probability, vectors in high-dimensional spaces become nearly orthogonal to one another. Consequently, processing in this framework relies on similarity measures between hypervectors, defined via the inner product.

The neuromorphic networks we have discussed encode information in various biologically inspired ways, drawing from the physical properties of the brain. In contrast, hyperdimensional computing models information encoding at a more conceptual, abstract, and functional level. Nevertheless, a strong connection, and potential synergy, between the two paradigms can be observed. This has motivated studies such as [41, 65] that propose frameworks blending the underlying principles of both computational models.

In this approach, low-level information is extracted from neuromorphic data, that is, characteristic and consistent spike-based patterns are identified. The goal is to reduce noise and leverage the implicit learning capabilities of Spiking Neural Networks (SNNs) to converge toward a meaningful solution. Subsequently, hyperdimensional computing maps these extracted spike patterns into a higher-dimensional space through a nonlinear transformation. Within this space, the system learns and classifies the data effectively.

This represents a particularly compelling example of how the complexity of neural network architectures continues to evolve through bioinspired methodologies. For instance, the authors of this approach highlight that their memory usage and management simulate processes akin to those of the cerebellum. Such models aim to achieve both greater fault tolerance and improved energy efficiency, reflecting key attributes of biological neural systems.

2.3.2. Sub-symbolism and fuzzy logic

Contemporary artificial intelligence (AI) systems, particularly deep neural networks, are plagued by well-documented explainability challenges. Despite impressive performance across tasks, their internal decision processes often resemble "black boxes," making it difficult to diagnose errors, guarantee safety, or satisfy regulatory requirements [17, 27].

To address this, researchers have proposed various interpretability paradigms. One prominent approach is *subsymbolic computation*, which seeks to extract higher-level patterns from distributed neural activations without reverting to rigid symbolic rules. Techniques such as feature visualization or rule extraction aim to reveal latent structure in learned representations, offering insights into model behavior while preserving the continuous, high-dimensional nature of neural networks [6, 45, 62].

A complementary strategy leverages *fuzzy logic* to capture the inherently parallel, distributed, and noise-prone dynamics of neural computation. By modeling partial membership in concepts through fuzzy sets, with membership degrees in $[0, 1]$, fuzzy neural networks (FNNs) combine the interpretability of linguistic "if–then" rules with the adaptive learning of artificial neural networks [5, 42]. FNNs have demonstrated success in control, classification, and decision-making tasks, but their fixed architectures and computational overhead limit scalability to more complex or real-time applications.

Spiking neural networks (SNNs), often termed the third generation of neural models, address many of these limitations by encoding information in discrete spike events and exploiting temporal coding for low-power, event-driven processing [36]. However, SNNs alone lack native mechanisms for handling linguistic uncertainty or deriving interpretable rules.

This gap has given rise to *fuzzy spiking neural networks* (FSNNs), which integrate fuzzy inference directly with the temporal dynamics of SNNs [23, 35]. In FSNNs, fuzzy membership functions can modulate synaptic weights or spike thresholds, enabling networks to represent and reason with vague or uncertain information while still benefiting from biologically inspired, low-power spike-based computation.

2.3.3. Non-linear oscillators

Finally, this work also proposes considering a neuroscience-based approach that has been gaining importance in recent years. Computational neuroscience is rethinking its perspective on cognition. Contemporary models suggest that brain functions, such as attention, memory, and decision making, emerge from the synchronization of coupled neuronal oscillators [1, 7, 8, 20]. Under this framework, each neuron acts as a nonlinear oscillator whose frequency and phase are modulated by sensory inputs, and cognition emerges from collective patterns of oscillatory coherence (e.g., gamma waves in the prefrontal cortex [20]).

This perspective is rooted in a long-standing tradition within dynamical systems theory, dating back to early studies on synchronization in biological systems. Classical models, such as Kuramoto networks [51], have demonstrated that oscillator synchronization is a fundamental mechanism for information integration in dynamic systems. These studies laid the theoretical foundations to understand how temporal coordination among oscillatory units can give rise to complex cognitive processes, a concept that has been further solidified by modern approaches, including analyses using Koopman operators [38], within the context of computational neuroscience.

This theoretical framework offers a powerful analogy for neuromorphic computing: if neurons are modeled as oscillators and synapses as parameterizable couplings, then computational operations could be implemented through the control of synchronization and frequency modulation.

These neuroinspired models have driven recent advances in oscillator-based computing [15, 30] . The hypothesis that networks of nonlinear oscillators can serve as universal function approximators is supported by fundamental studies in dynamical systems, computational neuroscience [58], and, more recently, by innovative proposals in artificial intelligence [61].

The seminal work of [11] establishes a theoretical framework for quantifying the information processing capacity of dynamical systems driven by inputs, demonstrating that, under general conditions and with sufficient degrees of freedom, these systems can approximate any functional mapping, whether linear or nonlinear, with arbitrary precision.

Moreover, recent advances in dynamical systems have explored the computational potential of networks of nonlinear oscillators from complementary perspectives. In the classical domain, [59] demonstrated that networks of phase oscillators can perform logical operations and signal processing via controlled synchronization. In their approach, inputs are translated into frequency modulations in the oscillators, and outputs are derived from collective phase patterns, effectively serving as a physical reservoir for processing temporal information. This scheme, known as reservoir computing, leverages the transient

dynamics of oscillators to solve tasks such as speech recognition or time-series prediction, thereby validating its utility in phase-based information transmission and neuromorphic computing.

Independently, in the quantum domain, [26] proposed a model in which networks of nonlinear oscillators serve as a physical substrate for the encoding of quantum states, specifically, qubits in coherent vibrational modes. Although originally designed for quantum computing, this work highlights the capacity of oscillators to support complex computational logics.

These findings converge on a central principle: interactions among coupled oscillators constitute a versatile computational substrate, capable of implementing logics that are both biologically plausible and formally universal. Within the context of neuromorphic computing, this raises an inevitable question: if oscillator networks can emulate cognitive tasks and approximate arbitrary functions, could they serve as a unifying abstraction for neuromorphic programming, effectively hiding the low-level complexity of spiking neurons?

The answer is already suggested at the hardware level [56, 60], and it seems to lie in the intrinsic plasticity of these systems. Just as the brain adjusts synaptic strengths to modulate oscillatory activity, a framework based on oscillators could allow global parameters (such as frequencies and coupling strengths) to be tuned in order to "map" desired functions, without requiring intervention at the level of individual neural circuits. Such an approach would not only mirror the natural organization of cognition, but also take advantage of mathematical properties [26, 34] that guarantee the expressiveness of these systems.

2.4. Computational Properties

In this section, we investigate whether this paradigm is Turing-complete and how it can be formally analyzed through precise definitions of complexity and neuromorphic algorithms. As we have seen, the shift from classical computing is substantial, and naturally, the methods for studying these systems also change. This subsection is particularly important for understanding how we will analyze the proposed algorithms throughout this work, as well as for highlighting their main limitations.

2.4.1. Turing-Completeness and Expressive Power

Turing completeness refers to a machine's ability to compute any arbitrary function that a Turing machine can compute. Gödel and Herbrand proposed a computational model

based on a set of functions, called μ-recursive functions [24], which were proven to be equivalent, so it suffices for us to demonstrate that these neuromorphic models are capable of computing the class of μ-recursive functions.

In this work, we follow the formalism introduced in [14] to characterize the Turing completeness of neuromorphic systems. The authors provide a constructive framework for proving that certain spiking neural networks, under well-defined conditions, can simulate any Turing machine. Therefore, our discussion and conclusions apply specifically to neuromorphic systems that meet these criteria.

Nevertheless, given that the terminology of neuromorphic systems encompasses a wide range of implementations, we can only draw conclusions for systems that fulfill the requirements proposed in [14]. Even so, most such systems share a common foundation: the use of Leaky Integrate-and-Fire (LIF) neurons.

The neuromorphic model considered for this proof of Turing-Completeness incorporates neurons that are allowed to connect with each other through synapses and can also receive external synaptic inputs representing I/O operations, which we designate as I/O neurons.

For the notation and representation of neurons used throughout this work, please refer to Appendix A.1. For the moment, we provide a brief explanation: Neurons are depicted as circles and are labeled with a numerical identifier. Each neuron is characterized by two parameters: the threshold v_i, an integer that specifies the value a neuron must exceed in order to fire, and the leak λ_i, another integer that defines the amount of time a neuron needs to get its internal state back to 0 in case it does not spike. We denote each neuron by

$$\{v_i,\, \lambda_i\}.$$

On the other hand, synapses are represented as arrows connecting one neuron to another. Each synapse is denoted by a numerical tuple indicating the presynaptic and postsynaptic neurons, respectively. Similarly, these synapses are endowed with two parameters: the synaptic weight ω_i, which scales the signal transmitted upon firing, and the delay δ_i, which represents the time lag before the spike reaches the postsynaptic neuron. Accordingly, we define each synapse as

$$\langle \omega_i,\, \delta_i \rangle.$$

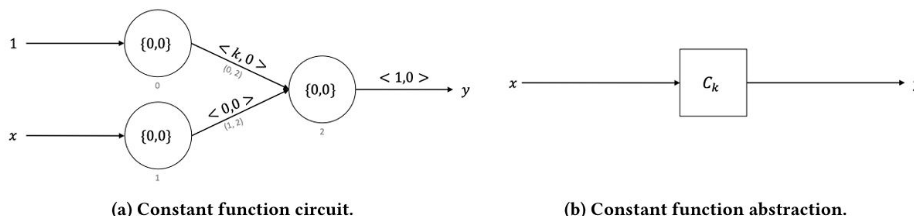
In our model, every synapse has an implicit delay of 1, which is crucial for verifying Turing-completeness using this approach. Moreover, the spikes carry numerical information, aligning with our study on the capability of representing information in spikes through spatiotemporal coding.

There are six μ-recursive functions that a neuromorphic circuit must be able to compute: the constant function, the successor function, the projection function, the recursion operator, the composition operator, and the minimization operator.

As an illustration, we will examine two functions to demonstrate the methodology used to implement such systems and highlight the key mechanisms employed to ensure Turing-completeness, in particular, the first one and the last one form the previous list.

1. **The Constant Function** is one of the simplest and earliest examples. It takes a natural number x as input and returns a constant natural number k. It is formally defined as:

$$C_k(x) := k \tag{5}$$



(a) Constant function circuit. (b) Constant function abstraction.

Figure 2: Neuromorphic circuit and abstraction of the Constant function. Figure extracted from [14].

As depicted in the corresponding figure, the implementation consists of three neurons. Neuron 0 receives an input spike with value 1, while neuron 1 receives the input value x. Both neurons emit spikes and transmit them to neuron 2. However, the synapse from neuron 1 to neuron 2, denoted as $(1, 2)$, has a synaptic weight of 0. Therefore, regardless of the value of x, its contribution is nullified. On the other hand, the synapse $(0, 2)$ has a weight of k. Since neuron 0 propagates a spike with value 1, the resulting input to neuron 2 is precisely k, causing it to spike and emit the constant value k as output.

This example illustrates how synaptic weights alone can encode logical operations. However, as we will see in the next example, more complex functions require more sophisticated mechanisms.

2. **The Minimization Operator**, discussed in the final section of the referenced paper, represents a significantly more complex case and introduces additional mechanisms of interest. In this function, we are interested in finding the smallest nat-

ural number for which a given function becomes zero. We define the function as $f : \mathbb{N}^{N+1} \to \mathbb{N}$, while the minimization operator is defined as:

$$\mu(f)(x_1, ..., x_N) := z \tag{6}$$

such that:

$$f(i, x_1, ..., x_N) > 0 \quad \forall i = 1, 2, ..., z - 1 \tag{7}$$

$$f(z, x_1, ..., x_N) = 0 \tag{8}$$

In order to understand the circuit for this latter function, we must first formalize an auxiliary neuromorphic component: the *Trigger Circuit*. As its name suggests, this circuit is designed to store a value (and erase it if needed) and propagate it upon receiving a specific signal.

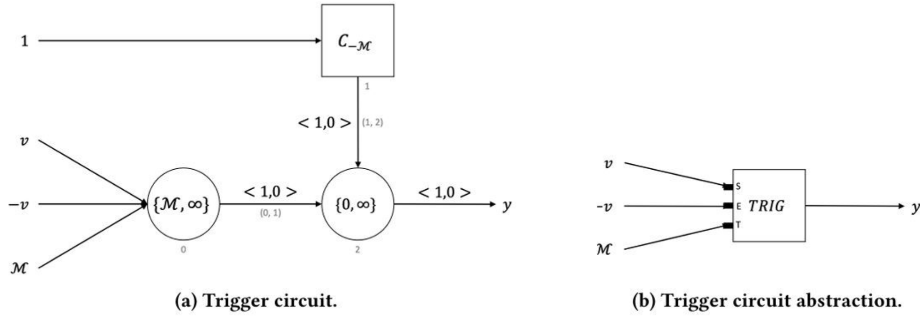


Figure 3: Neuromorphic circuit and abstraction of the Trigger function. Figure extracted from [14].

This circuit goes like this: neuron 0 receives three possible inputs: a value v, its negation $-v$, and a very large natural number $\mathcal{M}$. The value $\mathcal{M}$ is set as the threshold of neuron 0. Importantly, the leak λ of neuron 0 is set to infinity, meaning that the neuron retains its internal potential indefinitely until it spikes. This design allows neuron 0 to store the value v until $\mathcal{M}$ arrives. The presence of $-v$ allows for the deletion of the stored value if needed.

When $\mathcal{M}$ reaches neuron 0, it is added to the current internal potential regardless of its value. This sum exceeds the threshold, causing the neuron to spike. On the other side, circuit 1 implements a constant function we discussed previously, which propagates $-\mathcal{M}$ to neuron 2, allowing it to store this negative value.

The spike from neuron 0 travels to neuron 2, where all potentials are combined. Neuron 0 transmits $\mathcal{M} + v$, while neuron 2 already contains $-\mathcal{M}$. The result is the value v, which is emitted as the final output of the circuit.

These circuit designs also make clever use of temporal dynamics by incorporating the implicit delay of each synapse. This allows us to infer which signals arrive first. For instance, if neuron 0 takes a long time to spike, we know that $-\mathcal{M}$ will have already been stored in neuron 2. Since this stored value is far below its threshold, neuron 2 will not spike prematurely. In any case, these delays become far more critical in the circuit for the minimization operator, which we will now see.

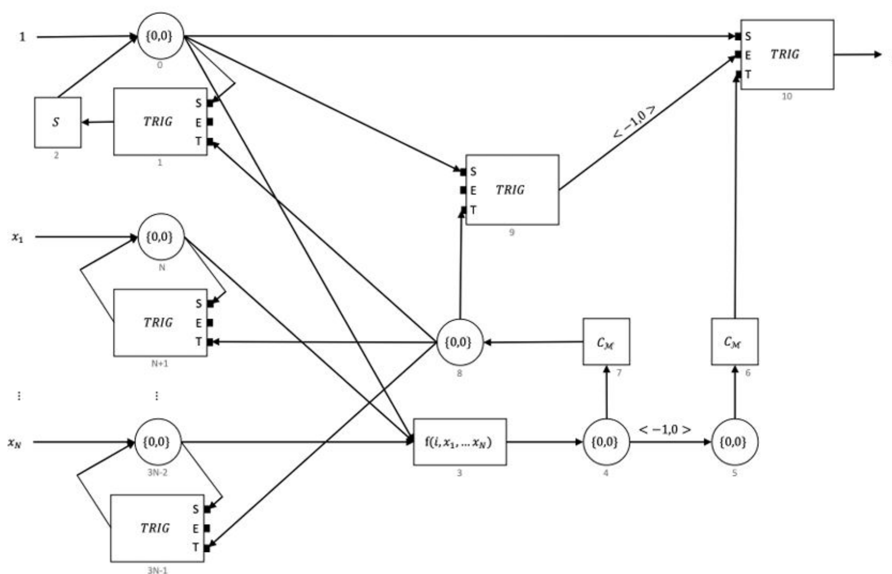


Figure 4: Neuromorphic circuit of the minimization operator. Figure extracted from [14].

In the minimization operator circuit, we receive all input values $x_1, \ldots, x_N$ through neurons N to $3N-2$, which are immediately stored in their associated Trigger Circuits, from neurons $N+1$ to $3N-1$.

Neuron 0 acts as our index i, iteratively searching for the smallest natural number such that the function f evaluates to zero. This neuron receives a constant input of 1, ensuring that it always spikes, and its value is stored in Trigger Circuit 1. Additionally, this index value i is stored in Trigger Circuit 10 in case it turns out to be the correct solution. If it is not, and we need to prepare the circuit for another iteration, the value is also saved in Trigger Circuit 9 to erase the value from Trigger Circuit 10.

We now consider Circuit 3, which receives both i and all inputs $x_1, \ldots, x_N$, and evaluates the function f at the current index. The result is then propagated to neuron 4. This marks

the key decision point of the entire circuit, which we will now analyze by distinguishing between two cases for better clarity:

1. **If** $f(i, x_1, \ldots, x_N) > 0$**:** In this case, neuron 5 will not spike because the propagated value is multiplied by -1, making it negative. Since the threshold of neuron 5 is set to 0, any negative value will prevent it from firing. Simultaneously, the constant function circuit 7 propagates the large value $\mathcal{M}$ to neuron 8, which in turn forwards this value to Trigger Circuit 9. This spike deletes the value of i previously stored in Trigger Circuit 10, and will trigger the contents of all other Trigger Circuits, specifically, Trigger Circuits 1 and $N+1$ through $3N-1$, resetting the entire system in preparation for the next iteration. Also it is crucial to note that once Trigger Circuit 1 propagates its value i, it goes to a Successor Circuit 2; this is another μ-function that we have not covered, but pretty similar to the constant function we saw and it just sums 1 to the value received, essentially dealing with the iterative functionality.

2. **If** $f(i, x_1, \ldots, x_N) = 0$**:** Starting again from neuron 4, both downstream synapses propagate the value 0. This time, neuron 5 does spike, because $0 \times (-1) = 0$, which matches its threshold. This triggers a race condition between two pathways: one aiming to delete the value of i, and the other to return it. Since the path from neuron 5 to Trigger Circuit 10 consists of only two synapses (each with an implicit delay of 1), and the path through circuit 7 involves three synapses (each also with a delay of 1), the spike from neuron 5 reaches Trigger Circuit 10 first. Therefore, the value of i is successfully returned *if and only if* $f(i, x_1, \ldots, x_N) = 0$.

This function illustrates an ingenious use of neuromorphic computing mechanisms and how they can be combined to construct circuits that support Turing-completeness. Temporal delays play a crucial role in this final step, ensuring that the index i is recovered before the system is reset for another iteration.

The main advantage of this approach lies in its simplicity and its ability to generalize across the many variations of neuromorphic systems currently in development. However, a key limitation is that it does not account for Spike-Timing-Dependent Plasticity (STDP), which appears to be an essential mechanism in neural computation and biological learning.

Several other studies explore this kind of computation from alternative perspectives. For instance, in [9] the author proposes a demonstration of Turing-completeness for recurrent neuronal networks that encode information through discrete synaptic levels rather than spikes. In other words, they focus on neuromorphic systems that assume STDP-based mechanisms to define their transition rules.

This added biological fidelity comes at the cost of more complex models that are harder to implement and replicate. Consequently, there remains an open debate between adopting simpler neuromorphic models, which are more tractable but less biologically faithful, and more detailed models that better emulate neural dynamics at the expense of increased computational and analytical complexity.

2.4.2. Algorithmic Complexity

As a relatively novel paradigm, neuromorphic computing lacks a well-established formal framework for analyzing its algorithmic properties. Most current approaches are ad hoc and primarily rely on basic intuition about how neurons function.

In [13], the authors propose a set of preliminary definitions to establish a foundation for understanding what constitutes a neuromorphic algorithm and how its complexity might be analyzed. This work extends the ideas presented before in [14], reusing the formalism of μ-recursive functions to evaluate the complexity of neuromorphic systems according to the principles they lay out.

To this end, they provide a series of definitions relevant to this context:

Definition 1: The information represented by the spike trains of a neuromorphic computer is referred to as *neuromorphic data*. A *spike train* is a temporal sequence of spikes emitted by a neuron over time, typically represented as a series of binary events (as we saw on Section 2.3) across fixed time steps.

Definition 2: A *neuromorphic algorithm* is a well-formed configuration of neurons and synapses in a neuromorphic computer, arranged in such a way as to solve a specific problem or perform a particular computation.

As the authors note, this notion significantly diverges from classical algorithmic theory, where algorithms are typically expressed in terms of well-defined instructions that are compiled into logical gates at the hardware level. In contrast, the absence of such abstraction layers in neuromorphic systems leads to a model where problems are addressed directly through the configuration of neurons and synapses, the fundamental building blocks of the circuit itself. The output of such a circuit is intrinsically tied to the spiking behavior of the neuromorphic network, which is tracked throughout the duration of the computation.

The following assumptions are made to establish a consistent computational model for neuromorphic systems:

1. Neurons require a constant and finite amount of time to perform the *integrate-and-*

fire operation, regardless of the number of incoming or outgoing synapses.

2. Synapses require a constant and finite amount of time to transmit a spike from the presynaptic neuron to the postsynaptic neuron, in the absence of any delay parameter.

3. A neuromorphic algorithm must receive neuromorphic data as input and produce neuromorphic data as output.

As described in [14], the total transmission delay between neurons is defined as $1 + \delta$, where δ is the explicit delay parameter of the synapse. This formulation assumes that, by convention, all synapses incur a default latency of 1 time unit for spike transmission.

Furthermore, by leveraging the space-time encoding previously discussed, spike trains can be associated with specific integers to simplify analysis. This contrasts with purely binary spike representations, which do not inherently carry numerical values.

The algorithmic complexity of a neuromorphic algorithm is then analyzed in terms of both space and time:

Spatial complexity is determined by the number of neurons and synapses, which represent the primitive computational units of the circuit. The number of synapses is naturally bounded by the number of neurons: in the worst-case scenario, where every neuron is connected to every other neuron, the total number of synapses is $\mathcal{O}(n^2)$, where n is the number of neurons.

Temporal complexity of neuromorphic algorithms is defined as the sum of the time required to configure and construct the circuit, and the time taken to execute it. Intuitively, the circuit configuration time is proportional to the *spatial complexity* of the system, as more neurons and synapses entail greater construction overhead.

However, the execution time is determined by the duration from the moment the neuromorphic network receives its input spikes until the final output spikes are produced. In the worst-case scenario, this corresponds to the longest path between any two neurons in the circuit, accounting for all synaptic delays along the path.

Thus, the total *temporal complexity* of a neuromorphic algorithm can be expressed as the sum of the construction time and the execution time:

$$T_{\text{total}} = T_{\text{build}} + T_{\text{exec}}$$

With these formal foundations in place, the authors proceed to analyze the algorithmic complexity of certain well-known μ-recursive functions, under the neuromorphic frame-

work. For example:

Consider the constant function employed in the Turing-completeness demonstration. Its spatial complexity is constant, as the circuit utilizes only three neurons and five synapses, resulting in $\mathcal{O}(1)$. Moreover, the execution time is simply the sum of the delays of the synapses from the input neuron to the output neuron. In our model, where each synapse has an implicit delay of 1 (with an explicit delay of 0), the longest path involves only one synaptic hop. Consequently, the temporal complexity is also $\mathcal{O}(1)$.

For more complex cases, such as the projection function [14], the spatial complexity is $\mathcal{O}(N)$, as it employs $\mathcal{O}(N)$ neurons and synapses. The configuration (or build) time of the circuit is also $\mathcal{O}(N)$, mirroring its spatial complexity. Finally, the execution time is determined by the longest path through the network. In this scenario, regardless of which branch is traversed, the path from input to output always passes through seven synapses. Since this number is constant, the execution time is $\mathcal{O}(1)$. Applying our earlier formula for the total temporal complexity, we obtain:

$$T_{\text{total}} = T_{\text{build}} + T_{\text{exec}} = \mathcal{O}(N) + \mathcal{O}(1) = \mathcal{O}(N).$$

A significant limitation of this complexity analysis is that (as we previously discussed), although it lays a foundation for evaluating the algorithmic complexity of a novel computational paradigm, it fails to capture all the intrinsic characteristics of neuromorphic networks. This model is predicated on the assumption of constant processing times to simplify the analysis, effectively sidelining critical mechanisms such as Spike-Timing-Dependent Plasticity (STDP) that are fundamental to learning and the dynamic behavior of biological neural systems. In practice, the execution time may be considerably influenced by the adaptive nature of these networks, and this complexity model only accounts for networks composed of very simple and predictable neurons.

2.5. Use Cases

In this section, we review some of the most prominent use cases of neuromorphic computing. Our focus will primarily be on the two most relevant domains for our purposes: graph-related problems and constraint satisfaction problems.

In both domains, we have implemented and analyzed several neuromorphic algorithms. Nonetheless, the main emphasis of this work lies in the development and evaluation of a neuromorphic algorithm for solving graph problems, making this section particularly relevant to our research.

2.5.1. Spiking Neuronal Networks: Deep learning

Artificial intelligence has experienced significant growth and widespread attention in recent years. The next generation of neural networks, referred to as third-generation networks, are the *Spiking Neural Networks* (SNNs), which are bio-inspired models based on the behavior of biological neurons. As previously discussed, these networks operate through electrical impulses known as *spikes*, relying on mechanisms such as membrane potentials and thresholds.

The main advantage of SNNs over second-generation *Artificial Neural Networks* (ANNs) lies in their energy efficiency, a defining trait of neuromorphic computing. SNNs support massive parallel and distributed data processing at significantly lower power consumption. Moreover, due to their inherent structure, SNNs offer a natural advantage when dealing with spatiotemporal and dynamic data, as they encode information through spatial and temporal spike patterns.

In addition, SNNs feature intrinsic learning mechanisms, such as Hebbian learning, that emerge naturally from their biological inspiration. One class of this Hebbian learning is the previously mentioned STDP, which enables native approaches to unsupervised learning and supports more biologically plausible models of adaptation and memory.

2.5.2. Graph Problems

An intuitive analogy that broadens the scope of neuromorphic computing beyond traditional machine learning is its natural correspondence with graph theory. In this perspective, neurons and synapses can be directly mapped to the vertices and edges of a graph, respectively, providing a structurally faithful and computationally powerful framework for graph-based problem solving.

In [33], the authors leverage this analogy by proposing a method to embed arbitrary graphs into a neuromorphic system. This embedding allows them to solve classical graph problems such as the Longest Shortest Path (LSP) and Minimum Spanning Tree (MST) in a straightforward and biologically inspired manner.

However, their model diverges from those seen in previous neuromorphic implementations: while synapses retain their usual parameters, the synaptic weight ω and transmission delay δ, the neurons preserve the firing threshold v, but replace the leak parameter λ with a refractory period t_R. This refractory period dictates the duration a neuron must remain inactive after firing before it becomes receptive to further input, enforcing strict control over temporal dynamics in the network.

The core of their method is a direct and systematic embedding of any directed weighted

graph $G = (V, E, w)$ into a spiking neural network. Each vertex $v \in V$ is mapped to a distinct neuron, and each edge $e \in E$ is represented by a corresponding synapse. The graphs considered are equipped with a weight function $w\colon E \to \mathbb{R}$, assigning real weights to the edges.

To break any ties they use this idea that lies in encoding these weights as synaptic delays via the following auxiliary function:

$$h\Big(e^{(i)}\Big) \;:=\; w\Big(e^{(i)}\Big) \;+\; \frac{1}{2^i} \;=\; w\Big(e^{(i)}\Big) \;+\; o_i,$$

where $e^{(i)}$ denotes the i-th edge, and $o_i = 2^{-i}$ is a fractional offset. This equation represents a transformed version of the synaptic weight. The function $h(e^{(i)})$ applies a small offset to the original weight $w(e^{(i)})$ to break ties between competing paths. It is important to note that $w(e^{(i)})$ and, by extension, $h(e^{(i)})$, ultimately correspond to the effective synaptic delay in our spiking neural network model.

Observe that the offsets o_i satisfy two crucial properties:

1. $\sum_{i=1}^{m} o_i < 1$, since the infinite geometric series $\sum_{i=1}^{\infty} 2^{-i}$ converges to 1.
2. For any two distinct index sets A and B,

$$\sum_{i \in A} o_i \;\neq\; \sum_{j \in B} o_j.$$

By adding a unique fractional offset $o_i = 2^{-i}$ to each weight $w(e^{(i)})$, we break any ties in the delays: if two edges have the same integer weight, then their different offsets make one delay strictly smaller than the other, so the corresponding neuron will always fire first. Property (2) guarantees no two sums of offsets are ever equal, and property (1) guarantees that the total shift introduced by all offsets is less than 1, so we never change the integer part of any weight, or in other words, the original order induced by the $w(e^{(i)})$ remains intact.

Finally, in the event that the neuromorphic system only accepts integer delays, simply multiply every delay by 2^m. This scales both the integer and fractional parts up to integers, preserving the strict ordering (since multiplying by a positive constant is order-preserving) while eliminating any remaining fractional components.

These offsets theoretically prevent ties, as previously discussed. However, they present two important drawbacks that are not explicitly addressed in the original paper. First, they are not strictly necessary: one can simply break ties non-deterministically, avoiding

any preprocessing overhead. For example, in the case of the LSP problem, choosing one shortest path over another is inconsequential as long as both have the same cost.

Second, their practical feasibility is highly limited. For instance, in a graph with only 100 nodes, a relatively small and manageable size, the required offsets would need a precision on the order of $1/2^{100}$, implying an extremely high number of bits for numerical representation. Moreover, this precision requirement translates directly into temporal constraints, since these delays are interpreted as synaptic delays in neuromorphic hardware. Consequently, the hardware would need to support timing resolution of that magnitude, which is practically unattainable.

For these reasons, in our subsequent studies we will not employ this weight-transformation technique. Nonetheless, it might still be worth exploring the encoding potential of such an offset system to represent shorter paths through precise spike timing. In principle, since the offsets are defined in base 2, one could infer the exact sequence of offsets traversed by a path by decoding the spike's arrival time.

The authors then set the refractory period of all neurons to a large constant value $t_R = \alpha$, where:

$$\alpha := \left(\sum_{e \in E} w(e) + 1 \right) + 1. \tag{9}$$

This value of α is deliberately chosen to be large, it exceeds the total sum of all synaptic delays plus an extra margin, ensuring that once a neuron fires, it will remain inactive long enough for all other neurons in the network to have had the opportunity to spike at most once. In other words, this setting guarantees that each neuron can spike at most once during the entire execution, effectively preventing feedback cycles or repeated activations.

Thus, the parameters of the neuromorphic system are configured as follows:

$$\left(\{v = 0,\ t_R = \alpha\},\ \langle \delta = h(e) + 1,\ \omega = 1 \rangle \right),$$

where each neuron has an initial membrane potential of zero and a large refractory period, while each synapse is assigned a delay of $h(e) + 1$ and a synaptic weight of 1.

Under this embedding, solving the Longest Shortest Path (LSP) problem becomes conceptually straightforward. By stimulating the source neuron v_s, spikes begin to propagate through the Spiking Neuromorphic Computing System (SNC). Neurons connected to the source via shorter paths will fire earlier due to lower cumulative synaptic delays, while neurons at the end of longer paths will spike later. The inclusion of a refractory period ensures that each neuron can spike only once, enforcing a strictly forward propagation of

the signal and effectively preventing loops and redundant activations.

This approach implicitly assumes that all nodes are reachable from the source v_s, ensuring termination in finite time. Alternatively, if reachability is not guaranteed, one may define the longest shortest path to a given node as infinite when no path exists from v_s. If the goal is to compute the longest among the shortest paths to reachable nodes, the algorithm can be executed for a maximum of α time steps, after which the last neuron to spike corresponds to the farthest reachable node in terms of shortest path length.

Additionally, as the algorithm progresses, each time a postsynaptic neuron fires the associated synaptic weight is updated. Thus, by inspecting the final state of the synaptic weights after the simulation concludes, it is possible to trace which synapses contributed to each neuron's activation, providing insight into the structure of shortest paths from the source.

In this neuromorphic setting, execution time is dictated not by the number of edges to be scanned, as is the case in classical algorithms, but rather by the synaptic delays, which are directly derived from edge weights. This stands in contrast to conventional approaches, where an algorithm must explicitly evaluate each edge regardless of its weight, leading to inefficiencies, especially in dense graphs.

By encoding edge weights as delays, the neuromorphic system effectively normalizes time complexity around the largest delay rather than the number of connections. This allows for more scalable performance, particularly when delays (i.e., weights) are bounded or normalized to prevent arbitrarily large waiting times.

Consequently, for dense graphs, this approach yields a significant advantage: the overall execution time scales with the maximum delay rather than the cardinality of the edge set. This leads to a time complexity of

$$\mathcal{O}\left(\max_{e \in E} w(e) \cdot |E|\right),$$

This approach is closely equivalent to the one presented in [53], where the SPP is addressed by using this same embedding but slightly adjusting the initial parameters of the SNC (i.e. using the w function directly instead of h). In their model, stimulating the source neuron initiates a spike propagation process through the network. The time at which each neuron fires corresponds to the length of the shortest path from the source node n_s to that neuron. If a neuron does not spike within the allocated simulation window, this indicates that no path exists from the source to that neuron.

The shortest paths across the network can then be extracted by inspecting the spiking

times and analyzing the synaptic weights (same as before). In particular, synapses that have undergone potentiation (i.e., weight increases) are those responsible for activating their postsynaptic neurons, and therefore, they form part of the shortest path from the source to that neuron. When multiple shortest paths exist (i.e., the solution is not unique), this method is capable of identifying all valid shortest paths simultaneously.

As we can see, STDP plays a crucial role in both algorithms, as it allows us to identify the traversed paths based on updated synaptic weights. However, its use in this context is essentially retrospective and deterministic: it serves to mark which synapses were responsible for activating a postsynaptic neuron after the fact. That is, once a neuron spikes, the synapse whose weight has changed is assumed to be the one that caused the activation, thereby allowing us to reconstruct the path taken by the spike.

Nevertheless, this application diverges from the biological purpose of STDP. In biological neural systems, STDP is a dynamic and continuous process that modifies synaptic strengths over time based on the relative timing of neuronal spikes, thereby enabling learning and adaptation. In contrast, these approaches employ STDP in a constrained and one-off manner, using it as a static tagging mechanism after simulation. While effective for extracting structural information from the graph, this use does not fully leverage STDP's nature as a learning rule, it is reduced to a tool for retrospective identification rather than real-time adaptation.

These examples represent just some among many approaches in which neuromorphic principles are applied to graph-theoretic problems, driven by their strong conceptual analogy. Beyond the models discussed here, alternative approaches have been proposed that extend the expressivity and flexibility of the neuromorphic framework. Some works, for instance, incorporate heterogeneous synaptic dynamics [12], while others explore probabilistic neuromorphic computation to tackle complex optimization problems such as the Travelling Salesman Problem (TSP) [32, 36].

In this work, however, we focus specifically on this embedding approach outlined above, as it offers both clarity and practical relevance for our purposes. Not only does it provide an elegant translation of weighted graph structures into neuromorphic hardware, but it also directly inspires our own proposal for a novel neuromorphic algorithm based on similar principles.

2.5.3. Constraint Satisfaction Problems

Another computational paradigm that shares a strong conceptual affinity with neuromorphic systems is that of *constraint satisfaction problems* (CSPs) [4]. Although our

discussion so far has primarily emphasized excitatory synapses with positive weights, it is important to note that synapses in biological systems frequently exhibit negative weights as well, commonly referred to as *inhibitory synapses.* These play a crucial role in shaping neural dynamics and are fundamental to maintaining balance and information flow in real neural circuits.

In the context of neuromorphic computing, inhibitory synapses allow for the implementation of mutually exclusive constraints, which are central to CSPs. Neurons connected through inhibitory synapses can effectively prevent one another from firing simultaneously, enabling the system to represent and enforce logical constraints in a naturally parallel and biologically plausible manner.

In the following section, we delve deeper into the role of inhibitory dynamics in neuromorphic architectures. Through a review of selected works, we illustrate how CSPs can be systematically translated into spiking neural circuits. These case studies highlight both the expressivity and computational potential of inhibitory synapses for solving complex constraint-based problems using neuromorphic hardware.

The starting point of our discussion is the paper [19], which introduces the neuromorphic computing paradigm and explores its relationship with Constraint Satisfaction Problems (CSPs). The authors propose a methodology for translating a CSP into a Spiking Neural Network (SNN). To fully understand the proposed approach, we must first define what a CSP is in formal terms.

A CSP is defined as a triple $\langle X, D, C \rangle$, where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is the set of variables,
- $D = \{D_1, D_2, \ldots, D_n\}$ is the set of domains, with each D_i representing the domain of variable x_i,
- C is the set of constraints that restrict the allowable combinations of variable assignments.

A CSP may admit one or more valid solutions, and in some cases, we may be interested in finding the optimal one according to some criteria.

The authors propose a step-by-step transformation of the CSP into an SNN:

1. **Neuron population assignment:** For each variable and each value in its domain, a population of n neurons is created. The size n should be sufficiently large to average out the effects of stochastic neural activity. Each population is stimulated with excitatory Poisson noise, which is essential for initiating a stochastic search

over the solution space. Additionally, inhibitory noise may be applied to refine the search process if needed.

2. **Domain competition through inhibition:** Within each domain, mutual inhibitory connections are established among all different domain values. This ensures that, on average, only one domain value is active (or selected) per variable at any given time.

3. **Constraint encoding:** For each constraint, if it is *positive*, that is, encouraging a relation between two variable assignments, an excitatory synapse is created between the corresponding neuron populations. If the constraint is *negative*, an inhibitory synapse is established instead.

The resulting SNN is designed to operate as a winner-takes-all system. As more constraints are satisfied, the inhibitory influence exerted by the currently dominant population on the others reinforces its own activation, stabilizing the network around satisfying configurations. However, to prevent convergence to suboptimal local minima and to allow continued exploration of the state space, the level of excitatory noise is set sufficiently high. This ensures the network continues its stochastic search until a valid solution satisfying all constraints is found, at which point it naturally settles into equilibrium.

Using this methodology, the authors successfully propose solutions to various problems, including graph coloring. However, several important limitations must be noted. 1) This bio-inspired heuristic is fundamentally quasi-probabilistic in nature. Although the neuromorphic network is designed to converge toward optimal solutions through guided stochastic exploration, there is no formal guarantee of convergence. While the underlying neural dynamics can be described using differential equations, predicting the system's behavior with precision remains highly challenging. This is primarily due to the arbitrary choices made during network design, such as the magnitude of excitatory noise or the number of neurons per subpopulation, which significantly influence the network's performance. These factors introduce a high degree of non-determinism and hinder formal analysis. In fact, due to this stochastic nature, the system must continue searching even after discovering a solution, meaning it never fully converges to an optimal solution with complete certainty.

2) Although the model captures some biologically plausible dynamics, it omits others that are critical in neuroscience. For instance, STDP, a mechanism central to learning and memory processes in biological systems, is entirely absent here. This omission contrasts with the graph-based problems discussed in the previous section, where STDP was essential to the network's functionality. There, STDP acted as a flagging mechanism to

identify traversed paths. In the current model, such functionality is not explicitly implemented. However, a natural next step for improving this heuristic would be to incorporate STDP as a learning mechanism. This addition could help the network gradually converge toward an optimal solution over time.

Nonetheless, the model presents promising advantages. Most notably, the energy consumption required to solve the proposed problems is approximately four orders of magnitude lower than that of quantum counterparts [19], and between two to three orders of magnitude lower than classical chips [39]. While the approach does not achieve competitive performance in terms of time when compared to specialized solvers, it does reach levels comparable to classical meta-heuristics applied to constraint solving, such as genetic algorithms or tabu search.

We have also attempted to replicate some of the results presented in various papers that employ this methodology of translating CSPs into SNNs to solve problems such as Sudoku [46, 47] in a python neuromorphic framework called NEST [21] . The underlying logic remains the same: constraints on the solution space are implemented as inhibitory synapses between neuronal populations representing different variables.

In our work, we applied this computational approach to explore classical graph problems such as the Hamiltonian Cycle and the Traveling Salesman Problem (TSP). However, although neuromorphic computing is indeed appealing for addressing NP-complete problems due to its inherent parallelism, we chose to focus primarily on the SPP instead of the TSP or Hamiltonian Cycle for two main reasons: (i) the SPP serves as a more accessible introduction to graph optimization, allowing us to investigate the neuromorphic paradigm more thoroughly without the additional complexity of problem-specific difficulties, and (ii) the TSP has already been explored in the context of neuromorphic computing [36], and our preliminary implementation would merely replicate previous experimental results without offering substantial innovation. Anyway, for a complete presentation of our experimental results, please see the appendix.

2.6. Neuromorphic Computing: Strengths and Limitations

This section provides a contextualized synthesis of the key points discussed throughout this work. One of the most prominent advantages of neuromorphic computing is its attempt to overcome the bottleneck inherent to von Neumann architectures by integrating memory and computation. This integration enables operations to be significantly more energy-efficient, although not necessarily faster. It also enhances scalability and parallelism while reducing latency, as operations are asynchronous and require only spike-based signaling. Moreover, neuromorphic systems exhibit inherent noise robustness, a property

naturally derived from neuron-like processing, and an implicit learning capability that encourages applications in artificial intelligence and robotics.

On the other hand, the main limitations are predominantly technical. Programming neuromorphic models remains a considerable challenge due to the substantial heterogeneity of current architectures and frameworks. The absence of a clear consensus on what precisely defines "neuromorphic" computing results in excessive flexibility, which can hinder the development of standardized methodologies. Likewise, there are no universally accepted performance metrics, as the field is still in a formative stage. At the hardware level, constraints such as limited bus connectivity, routing inefficiencies, and the maximum number of neurons per chip can significantly restrict the theoretical parallelism promised by neuromorphic designs. Furthermore, there is substantial variability across hardware platforms, for instance, depending on whether adaptive transistors such as memristors are employed, which complicates reproducibility and benchmarking. Another critical limitation lies in the reduced numerical precision and unconventional data representations often used in neuromorphic systems to improve energy efficiency; while beneficial for speed and power consumption, these choices can limit the accuracy and generalizability of computations in certain application domains.

3. Our Spiking Neural Network for Graph Problem Solving

Our case study begins with the *Shortest Path Problem* (SPP), one of the most well-known and extensively studied problems in graph theory. Although it is a polynomial-time problem that does not require alternative computing paradigms for efficient solutions, it serves as an excellent starting point for exploring the neuromorphic computing approach. The SPP allows us to examine the paradigm shift in computation, understand its internal mechanisms, and evaluate both its strengths and limitations when addressing graph-based problems. All source code developed for this study is publicly available in a GitHub repository [66].

To formalize our approach, we first introduce the notion of a directed weighted graph, adopting the same notation as in the papers seen in Section 2.5.2, which will serve as the foundation for our model.

Let $G = (V, E, w)$ be a directed graph with weights, where:

1. V is a finite set of vertices or nodes.
2. $E \subseteq V \times V$ is a finite set of directed edges, i.e., ordered pairs of vertices.

3. $w : E \to \mathbb{R}$ is a weight function that assigns a real-valued cost to each edge.

Throughout this work, we adopt the following notational conventions:

- $V := \{v_1, v_2, \ldots, v_n\}$ denotes the set of vertices.
- $E := \{e^{(1)}_{i_1 j_1}, \ldots, e^{(m)}_{i_m j_m}\}$, where each edge $e^{(t)}_{i_t j_t} = (v_{i_t}, v_{j_t}) \in E$, represents a directed connection from node v_{i_t} to node v_{j_t}.

We will use this foundation to develop our own algorithm for solving the SPP. In order to provide a comprehensive approach to this challenge, we also consider classical algorithms, allowing us to assess the advantages offered by this type of computation in comparison to traditional approaches, as well as to other neuromorphic algorithms proposed in recent literature.

This guided construction allows us to clarify the conceptual underpinnings of the task before addressing the technical details of its realization. Such a methodology ensures that the neuromorphic implementation remains aligned with the original problem formulation and helps us identify where and how neuromorphic dynamics can offer real advantages.

Accordingly, we divide this section into two subsections:

1. We begin by outlining the problem at an abstract level, followed by an analysis of the associated time complexity. We also examine in greater detail the referenced work [53], highlighting its core assumptions and commenting on its main limitations.
2. We then introduce our chosen neuromorphic tools, explain the rationale for their selection, and argue why they represent a competitive alternative. Additionally, we propose some software simulations and compare every approach.

The foundational paper [53], as analyzed in Section 2.5.2, employs mechanisms that we consider neuromorphic in spirit: it uses LIF neurons with refractory periods, and even STDP. Moreover, their algorithm runs in $\mathcal{O}(|E|)$ time; however, it relies on a critical hardware assumption: one must be able to read the weights of all synapses in order to detect which connections have been modified by STDP, since this information is required to reconstruct the path. After reading each synaptic weight, one must compare it to the corresponding edge weight in the original graph; and, as we will later see, use a back-tracing mechanism to reconstruct the shortest path. This requirement adds both design complexity and additional computational overhead that is not accounted for in the original complexity analysis. We will discuss this later on.

An important point to consider is whether this neuromorphic paradigm and the proposed solutions represent viable alternatives to traditional approaches. For this reason, we later

introduce a standard implementation of Dijkstra's algorithm [31]—one of the most well-known algorithms in graph theory—in order to compare it against both the algorithm presented in the referenced paper and our own, with respect to execution time, spatial complexity, and energy consumption.

3.1. Functional level

First, we will outline our algorithm's functional structure at a high level. Analogously to the methodology proposed in Section 2.5.2, we embed the graph directly into a SNC composed of neurons and synapses. However, our approach introduces key conceptual differences that significantly alter the structure and behavior of the algorithm.

Functionally, the algorithm proceeds by constructing the SNC, mapping each vertex to a neuron and each directed edge to a synapse. The system parameters are initialized as

$$\Big(\{v = 0,\ t_R = 0\},\ \langle \delta = w(e) + 1,\ \omega = 1 \rangle\Big).$$

Algorithm 1 Shortest Path Problem on a SNC

Require: Directed graph $G = (V, E, w)$, source node v_s, target node v_d
Ensure: Shortest path from v_s to v_d

```
1: while v_d ≠ v_s do
2:     Run the SNC
3:     if v_d spikes then
4:         Record v_pred, the presynaptic neuron that triggered the spike
5:         Update v_d ← v_pred
6:     end if
7: end while
8: return all the predecesor neurons
```

We begin by entering a loop in which our SNC is executed iteratively. Each run of the SNC consists of two main steps: (i) stimulating the source neuron v_s, and (ii) propagating all spikes through the network until the destination neuron v_d fires for the first time.

Once v_d spikes, we identify the presynaptic neuron responsible for triggering it and record this connection as part of the reconstructed path; this is done by listening to the multiple inputs of v_d so that, when it spikes, we instantly retrieve the ID of the presynaptic neuron. The process is repeated with the newly identified neuron as the next destination, continuing until the source and destination neurons coincide. This marks the completion of the execution.

Unlike previous algorithms, our method does not rely on STDP to identify the path. Instead, we leverage the fact that neurons, at the hardware level, can distinguish between their multiple input synapses. By simply inspecting the postsynaptic neuron, we can infer which presynaptic neuron caused its activation. Thus, our algorithm eliminates the need for both refractory periods and STDP-based path tracing.

This behavior could also be replicated in software implementations (without requiring hardware-level changes). If spikes were allowed to carry numerical information, using any of the spike-coding schemes discussed in previous sections, we could encode a unique identifier within each spike. Upon reception, the postsynaptic neuron could decode this information to unambiguously determine which presynaptic neuron triggered its activation.

The solution proposed in [33, 53] is theoretically consistent with the inherent parallelism of spiking neural systems. Like our approach, they employ the standard LIF architecture and leverage features such as STDP to reconstruct the path.

One might ask why they do not adopt the same strategy we use, namely, listening to the spiking neurons to identify their presynaptic neurons on the fly and storing them in a dedicated register, either neuron-specific or associated with the processing unit. However, this requirement would break the simplicity of the LIF model and introduce additional custom spatial complexity. It would necessitate dynamically storing and updating the presynaptic neurons during execution, in order to perform immediate back-tracing without having to compare all edges from neurons belonging to the path, as is required when using STDP.

In the other hand, as discussed in Section 2.5.2, their method leverages global refractory periods α that are directly derived from the edge weights. This design ensures that a single spike originating at the source neuron will propagate through the network, with faster synaptic connections reaching subsequent neurons first, effectively 'closing' them and guiding the signal propagation. The shortest path is thus guaranteed to emerge naturally, with computational complexity depending solely on the maximum time required for every neuron to spike, which is given by α, defined in Equation (9).

In our approach, however, we adopt an iterative strategy in which spikes are propagated without predefined refractory periods, allowing neurons to fire multiple times. This behavior can be modified if needed, particularly because we must also compute the value of α to establish an execution time bound and prevent the network from remaining active indefinitely. Moreover, it may be more efficient in some scenarios to allow a neuron to remain idle rather than firing unnecessarily.

If we choose to have no refractory periods, this feature becomes especially advantageous in dynamic graph settings, where nodes may be added or removed. Under the method proposed in [33], such modifications would require reconfiguring the entire network to update the global refractory parameter α. While this dependency could theoretically be mitigated by setting $\alpha = \infty$ to ensure each neuron fires only once, doing so undermines one of the key advantages of using α in the first place, namely, its role as an upper bound on execution time based on the total weight distribution of the graph. A finite α allows us to precalculate the worst-case runtime of the algorithm. By contrast, choosing an unbounded refractory period would lead to overly conservative timing assumptions, making it impossible to determine whether or when the algorithm will terminate.

Moreover, at the hardware level, simulating infinite refractory periods would require neurons to maintain internal counters for extended durations, leading to inefficiencies and idle neurons that are temporarily excluded from computation.

For these reasons, defining α as a function of the graph's weight distribution improves predictability and runtime guarantees. While this choice introduces additional complexity in configuring and statically constructing the network (preprocessing in the order of $\mathcal{O}(|E|)$), it ensures a more robust and efficient execution model. In our case, the introduction of an explicit and arbitrary runtime bound serves a similar purpose.

Consequently, our approach partially 'departs' from the neuromorphic paradigm. It serves as a hybrid strategy, bridging a classical iterative implementation with the architectural advantages of spiking neural circuits. The trade-off lies in reduced algorithmic elegance but also reduced technical complexity, as our model does not require either refractory dynamics or, as we discussed before, STDP.

In summary, our algorithm stands by these fundamental principles:

- **Unrestricted spiking activity.** Unlike the previously studied SNC models, our neurons are not constrained by a refractory period t_R. They are permitted to spike repeatedly without limitation.
- **Direct topological embedding.** We do not rely on fractional offsets or intermediate transformations. The circuit construction maps the graph's topology directly, preserving its original structure without auxiliary modifications, at least during the initial stages.
- **Iterative circuit simulation.** Our algorithm requires multiple executions of the SNC due to removing STDP for path reconstruction. Consequently, the total computation time scales linearly with the length of the computed path, reflecting the

iterative nature of signal propagation and feedback through the circuit.

In [53], the shortest-path computation on a neuromorphic computer requires $\mathcal{O}(|E|)$ time when synaptic weights are bounded, which means that we assume a maximum weight that is constant for our instances. Our method then similarly propagates spikes to the destination in $\mathcal{O}(|E|)$ simulation time in the first iteration, giving us the shortest path's time, but then must reconstruct the actual hop-by-hop path (what we will call the '*back-tracing* phase'), which was not accounted for in the referenced paper.

In our SNC-based back-tracing algorithm, each iteration consists of two steps: (i) executing full spike propagation, whose duration is proportional to the *remaining* minimum synaptic delay in the network, and (ii) virtually removing the incoming edges of the current destination vertex by setting the destination neuron to be the presynaptic neuron that first triggered the previous destination neuron.

It is important to note that the cost of each iteration decreases over time and is always bounded by the remaining minimum delay. This is achieved by construction: once the source neuron fires, it propagates its spike through the whole network, from which we can deduce that the first spike to reach our destination neuron is responsible for being part of the shortest path to that neuron, without regard for any spikes that may come next. We then remove that postsynaptic neuron from the path which effectively eliminates the edge with the shortest delay connecting it to its presynaptic neuron.

Overlapping spikes are not a concern here, even in the absence of refractory periods, as the network dynamics ensure that the fastest path always activate the destination neurons first. In contrast, the method presented in the referenced paper does not allow such unconstrained spiking because allowing unrestricted spiking would distort the STDP-driven flagging process and contaminate the network information essential to their approach.

3.1.1. Time complexity

We then wish to analyze the time required for the *back-tracing* phase once the destination vertex v_d spikes. To that end, suppose that the shortest path from the source v_s to the destination v_d consists of exactly h edges. Let $|E|$ denote the total number of edges in the graph. Since we assume that each edge has a constant upper-bounded delay the total propagation time incurred along any path is proportional to the number of edges it contains. In particular, the number of edges in any path is at most $|E|$, and thus we use $|E|$ as an upper bound on the total propagation time.

Initially, this assumption allows us to upper bound the time needed for the back-tracing procedure by $\mathcal{O}(|E|)$, since each edge in the path contributes a bounded delay. Then, we

assume our path of length h has the following delays:

$$\delta_1,\ \delta_2,\ \ldots,\ \delta_h.$$

For convenience we define

$$\delta_0 \ := \ 0,$$

After the initial spike at v_s, the propagation time to reach v_d is $|E|$. Once v_d spikes for the first time, we remove all incoming edges to v_d. Consequently, in the next iteration the propagation time is reduced by δ_1, the delay of the edge that led into v_d. After that spike, we back-trace to the predecessor of v_d along the shortest path, and remove the incoming edges to that predecessor, reducing the next iteration's propagation time by δ_2, and so on.

Hence, the propagation times at each of the h iterations are

$$\big|E\big|, \quad \big|E\big| - \delta_1, \quad \big|E\big| - \big(\delta_1 + \delta_2\big), \quad \ldots, \quad \big|E\big| - \sum_{i=1}^{h-1} \delta_i,$$

Therefore, the total propagation time over all h hops is

$$\sum_{k=0}^{h-1} \Big(|E| \ - \ \sum_{i=0}^{k} \delta_i \Big)$$

Since $\delta_0 = 0$, the term $\sum_{i=0}^{k} \delta_i$ simplifies to $\sum_{i=1}^{k} \delta_i$, and we may safely shift the inner summation index to start from 1. Furthermore, $|E|$ is constant across all k, so we can pull it out of the summation:

$$\sum_{k=0}^{h-1} \left(|E| - \sum_{i=1}^{k} \delta_i \right) = h \cdot |E| - \sum_{k=1}^{h-1} (h-k)\, \delta_k.$$

Next, observe that for each $1 \le k \le h-1$,

$$(h-k)\, \delta_k \ \le \ h\, \delta_k,$$

because $k \ge 1$ implies $h - k \le h$. Therefore,

$$\sum_{k=1}^{h-1} (h-k)\, \delta_k \ \le \ \sum_{k=1}^{h-1} h\, \delta_k \ = \ h \sum_{k=1}^{h-1} \delta_k \ \le \ h\, |E|,$$

where the last inequality follows from the fact that the sum of all bounded delays along

the shortest path is at most $|E|$. Consequently, this establishes the desired upper bound. In Big-O notation,

$$\sum_{k=1}^{h-1} (h-k)\,\delta_k \;=\; \mathcal{O}\Big(h\,|E|\Big).$$

Therefore, the total time for the back-tracing phase satisfies

$$h\,|E| \;-\; \sum_{k=1}^{h-1} (h-k)\,\delta_k \;=\; h\,|E| \;-\; \mathcal{O}\Big(h\,|E|\Big) \;=\; \mathcal{O}\Big(h\,|E|\Big).$$

In other words, even though each successive iteration's simulation time shrinks by the delay of the edge just removed, the cumulative propagation cost remains on the order of $\mathcal{O}\Big(h \times |E|\Big)$. Finally, since always $h \leq |V|-1$, the worst-case running time can be written as

$$\mathcal{O}\Big(h\,|E|\Big) \;\subseteq\; \mathcal{O}\Big((|V|-1)\,|E|\Big) \;=\; \mathcal{O}\Big(|V|\,|E|\Big),$$

If weights were not bounded we would take a similar approach as in [33]; denoting the maximum synaptic delay in the graph as $\delta_{\max}$, then we would also have the alternative bound

$$\mathcal{O}\Big(\delta_{\max} \cdot h|E|\Big)$$

This higher complexity reflects the inherently iterative nature of our back-tracing method: we explicitly recover each neuron on the shortest path one hop at a time. By contrast, the algorithm in [53] stops after the initial wavefront propagation and does not explain how the actual path is reconstructed. The authors suggest that one should identify exactly those synapses whose weights were modified by STDP during the simulation, but in order to do so, one must compare each synapse's final weight against its original value to determine which edges were used.

This implicit 'compare-against-original' step introduces nontrivial overhead that is never analyzed in [53]. Concretely, once the wavefront propagation finishes, one must identify exactly which of the $|E|$ synapses had their weights modified by STDP in order to recover the shortest-path edges. If one claims that the total runtime (forward propagation plus reconstruction) remains $\mathcal{O}(|E|)$, then this post-processing stage itself must also complete in $\mathcal{O}(|E|)$ time or faster. In practice, achieving, or even approaching, that bound requires at least two costly hardware strategies:

1. *On-chip storage of both "original" and "current" synaptic-weight tables:* To compare each synapse's final weight against its original, the chip must retain a full copy of the original weights in memory alongside the mutable weights. If the original design already used $\Theta(|E|)$ bits to store the dynamic weights, adding a duplicate of the same

size raises spatial complexity from $\Theta(|E|)$ to $\Theta(2\,|E|)$. While this still collapses to $\mathcal{O}(|E|)$, it effectively doubles (or more, depending on bit-width) the on-chip memory requirement.

2. *A massively parallel comparison network (comparators + interconnect):* Even if the chip has both tables in memory, performing $|E|$ weight-vs-original comparisons one after another would cost $\Theta(|E|)$ cycles in a purely sequential loop. To avoid this extra $\Theta(|E|)$ latency and 'hide' it under the asymptotic $\mathcal{O}(|E|)$ of the forward pass, one would need:

    - A separate comparator circuit for each of the $|E|$ synapses, so that all comparisons can occur in the same cycle; and
    - A bus or reduction tree wide enough to collect $|E|$ one-bit comparison results and produce, in $\mathcal{O}(1)$ or $\mathcal{O}(\log|E|)$ time, the set of edges whose "original" and "current" weights differ.

    Both components, the $|E|$ parallel comparators and the $\Theta(|E|)$–wide interconnect, consume $\Theta(|E|)$ gates and wiring resources. In a graph where $|E|$ is around 10^5–10^6, dedicating that much area to comparator arrays and a full-width bus is simply infeasible: the total chip area would scale linearly with $|E|$, multiplying by large constant factors (comparator bit-width, wire pitch, metal layers), which no commercial design can accommodate.

If one accepts those two provisions, and thus manages to keep the post-processing in $\mathcal{O}(|E|)$ time, so that

$$\underbrace{\mathcal{O}(|E|)}_{\text{Forward propagation (wavefront)}} \quad + \quad \underbrace{\mathcal{O}(|E|)/\mathcal{O}(1)}_{\substack{\text{Post-processing (reconstruction) time} \\ \text{(either in sequential or parallel fashion)}}} \quad = \; \mathcal{O}(|E|).$$

, then the required hardware area is prohibitively large for any chip of practical size. In other words, although the *time* complexity remains $\mathcal{O}(|E|)$, the spatial complexity and wiring complexity grow by significant constant factors (indeed, linear in $|E|$ with nontrivial per-edge overhead), which [53] does not address. Consequently, any implementation that tries to reconstruct the path by comparing synaptic weights must either:

- Accept a strictly sequential post-processing loop (and pay an additional $\Omega(|E|)$ time penalty to read and compare each synapse one by one, on top of the forward pass); or
- Pay a large area penalty (embedding $|E|$ comparators and a $\Theta(|E|)$–wide reduction

network) in order to perform all comparisons in parallel, as one would expect from the neuromorphic paradigm.

Both choices illustrate that recovering the shortest path carries its own nontrivial cost, either in cycles or in chip area, that cannot be ignored. We will delve deeper into our own spatial complexity in the following section.

But there is two last issues surrounding this approach; consider now, for example, this scenario in which the directed graph has multiple shortest-path candidates with same delays. For the sake of concreteness, suppose that:

1. There are n vertices labeled $\{v_1, \ldots, v_n\}$, with $v_1 = v_s$ as the source and $v_n = v_d$ as the destination.
2. The source node v_s is connected to every intermediate vertex $\{v_2, \ldots, v_{n-1}\}$, and each of these intermediate vertices is in turn connected to the destination v_d.

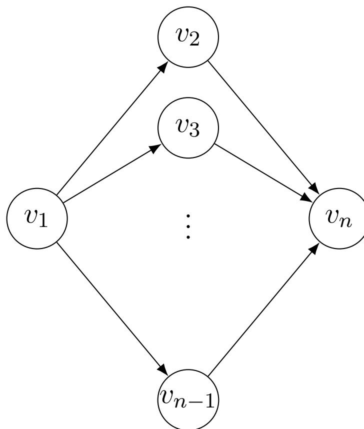


Figure 5: A directed graph with multiple possible paths: the source $v_1 = v_s$ connects to all intermediate nodes, which then connect to the destination $v_n = v_d$. Every edge has $w(e) = 1$.

In this configuration, every two-hop path of the form

$$v_s = v_1 \to v_i \to v_n = v_d \quad \text{for } i = 2, 3, \ldots, n-1$$

is a valid candidate for the shortest path. Thus, the number of distinct two-hop paths from v_s to v_d is $n-2$.

Although this graph remains within $\mathcal{O}(|E|)$, the presence of multiple candidate paths invalidates the assumption that STDP alone determines the unique shortest path. Indeed, after α time steps, the synaptic weights associated with all $n-2$ two-hop paths may have been modified, in which case we should decide what path to take.

In the case of our algorithm (as well as in the simulations we will later conduct), this ambiguity is resolved through non-deterministic path selection. That is, when multiple equally optimal paths exist, one is chosen arbitrarily, typically the first one to be reached during the spike propagation process.

But, if we want to assume that [53] applies weight offsets analogously to [33], then more complexity is added due to the preprocessing of recalculating every weight with cost $\mathcal{O}(|E|)$. So, we would end up with the following complexity,

$$\underbrace{\mathcal{O}(|E|)}_{\text{Offsets preprocessing}} + \underbrace{\mathcal{O}(|E|)}_{\text{Wavefront propagation}} + \underbrace{\mathcal{O}(|E|)/\mathcal{O}(1)}_{\text{Path retrieval (seq. or parallel)}} = \mathcal{O}(|E|)$$

Despite these offsets (which can be perfectly omitted), sometimes we will find ourselves in a situation in which even paths that do not ultimately reach v_d will have undergone weight changes during the α time steps; making it harder to filter the neurons that are actually part of the shortest path's solution, as we would need to concurrently consider all possible paths (modified synapses) from the source neuron, or do a similar back-tracing mechanism.

Figure 6 illustrates this important limitation: when applying STDP, the fastest synapses may correspond to alternative paths, while the actual shortest path to the target node v_n could be the slowest among them. This figure presents an exaggerated scenario to demonstrate that all synapses in the graph will be modified, as all neurons are eventually forced to spike. Upon stimulating neuron 0 (or analogously, vertex v_1), propagation will occur through all three available paths. Since the delays in the alternative paths are shorter than the one leading to v_n, we can expect all neurons in the graph to spike eventually.

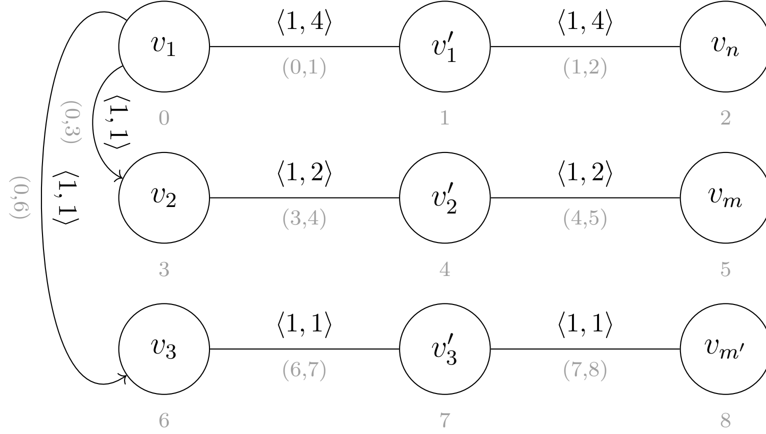


Figure 6: A spiking neural network (SNN) with multiple independent paths is illustrated, where neuron and synapse identifiers are shown in grey (as in [14]), vertex identifiers inside the neurons are displayed in black, and synapse parameters are also denoted in black. Among all possible paths, only the top path $v_1 \rightarrow v'_1 \rightarrow v_n$ is valid as the shortest path from v_1 to v_n.

This poses a challenge: it is not enough to simply read the modified synapses. Instead, we must necessarily start from the destination neuron and examine its incoming synapses to determine which one was modified. This introduces a back-tracing step similar to the one we propose, but in their case, STDP is used as a flag indicating the path along which the spike traveled most quickly.

In the end, since multiple synapses must be checked for each vertex that may belong to the shortest path, the complexity of the reconstruction phase can reach $\mathcal{O}\Big(h \cdot |E|\Big)$, where h is the length of the reconstructed path. This increases the overall asymptotic complexity of the algorithm, reaching, in the worst case, $\mathcal{O}\Big(|V| \cdot |E|\Big)$, just like our proposed algorithm. So their complexity ends up being:

$$\underbrace{\mathcal{O}(|E|)}_{\text{Offsets preprocessing}} + \underbrace{\mathcal{O}(|E|)}_{\text{Wavefront propagation}} + \underbrace{\mathcal{O}(h|E|)}_{\text{Back-tracing Path retrieval}} = \mathcal{O}(h|E|)$$

All of this highlights how our proposed algorithm may not only be more physically feasible, but also potentially faster in some cases, as it requires only $\mathcal{O}\Big(h\,|E|\Big)$ time without the need for any extra pre- or post-processing steps. Nevertheless, it is true that, in order to propose a correct execution timeout and determine whether a path exists or not, the preprocessing of α—the sum of all delays in the graph—becomes essential for both our approach and that of the referenced paper. This introduces an additional complexity

of order $\mathcal{O}(|E|)$. However, since this preprocessing is a shared requirement, we will not include it in our subsequent performance analyses.

3.1.2. Spatial Complexity

The spatial complexity of our model, following the heuristic proposed in [14], is straightforward to determine: since the graph-to-network mapping is one-to-one, we will have $\mathcal{O}(|V|)$ or $\mathcal{O}(N)$ vertices alongside $\mathcal{O}(|E|)$ edges. In the worst-case scenario, for a fully connected graph, this results in a spatial complexity of $\mathcal{O}(N^2)$. The model proposed in the referenced paper exhibits the same spatial complexity, but with the added non-trivial overhead of storing a copy of the original graph in memory. While this could potentially be mitigated through specialized hardware mechanisms, it would require re-engineering components such as LIF neuron capabilities to accommodate such a design.

Table 1: Complexity comparison between our model, the referenced neuromorphic algorithm, and Dijkstra's algorithm.

	Our Model	**Neuromorphic Paper [53]**	**Dijkstra**
Spatial Complexity	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$	$\mathcal{O}(\|V\| + \|E\|)$
Time Complexity	$\mathcal{O}(h\,\|E\|)$	$\mathcal{O}(h\|E\|)$	$\mathcal{O}(\|E\| + \|V\| \log \|V\|)$

3.2. Neuromorphic level

After examining both approaches, ours and the one presented in the referenced paper, we now formalize the type of neuron that our model employs:

As previously mentioned, our algorithm requires only a minimal neuromorphic substrate. Neurons must be capable of (i) firing a spike once a threshold is reached, and (ii) upon firing, identifying which presynaptic neuron triggered the spike. Synapses, in turn, need only delay the spike before it reaches the postsynaptic neuron, as previously discussed.

This behavior aligns well with that of Integrate-and-Fire (IF) or Leaky Integrate-and-Fire (LIF) neurons, which are widely accepted and implemented both in theoretical frameworks and in state-of-the-art neuromorphic hardware.

In our approach, we eliminate the need for STDP mechanisms and refractory periods, thereby simplifying the implementation of the neuromorphic network.

However, one might argue that this implementation is redundant when compared to classical algorithms. After all, shortest-path problems can be efficiently solved using Dijkstra's algorithm, which has a time complexity of $\mathcal{O}(|V| + |E|)$, compared to our $\mathcal{O}(h\,|E|)$ solution.

Nevertheless, a key conceptual shift lies in the fact that we no longer rely on explicit arithmetic operations or distance comparisons. Instead, we reconstruct the path through neuronal dynamics and synaptic delays alone.

The main advantage, therefore, does not lie in time complexity but rather in energy consumption. Our neuromorphic approach is expected to consume orders of magnitude less energy to solve the shortest-path problem, an expectation supported by prior estimates in [53], as neuromorphic operations are more energy efficient. We will later provide an energy estimate based on the power consumption data for neurons and synapses reported in the referenced article.

3.2.1. Software Simulations

For now, we will propose a set of software-based simulations. Our objective is to observe the behavior of our algorithm, characterize its performance, and verify whether our theoretical assumptions and following developments hold in practice.

As previously stated, we will also implement the algorithm presented in the reference paper [53], incorporating the STDP-based back-tracing mechanism that we consider essential for the path reconstruction. And, finally, we will include a standard implementation of Dijkstra's algorithm for comparison.

These simulations are designed to replicate, to a reasonable extent, the neuronal dynamics of our model. However, since our goal is not to achieve full biological or technical fidelity, we allow certain simplifications.

To obtain a reasonably realistic estimate, we will construct one model to help determine both the execution time on a neuromorphic chip and the corresponding energy consumption. The first approach will provide an idealized estimate, disregarding factors such as bus delays or other hardware-related constraints. In contrast, the second approach will aim to offer a more realistic approximation, incorporating additional details informed by current literature.

Our Algorithm

Our algorithm uses two main loops: the outer loop iterates over the number of hops h required by the network, while the inner loop iteratively simulates the behavior of the spiking neural network using counters as delays to propagate spikes.

Algorithm 2 Shortest Path via SNC Simulation (Ours)

Require: Directed graph $G = (V, E, w)$, source v_s, target v_d, timeout α
Ensure: Shortest path from v_s to v_d

```
 1: path ← [ ]
 2: while v_d ≠ v_s do
 3:     Initialize active set with {(u, w(v_s, u), v_s) | (v_s, u) ∈ E}
 4:     step ← 0, propagated ← false
 5:     while not propagated do
 6:         step ← step + 1
 7:         if step > α then
 8:             return ∅                                              ▷ timeout
 9:         end if
10:         Decrement delays in active; collect ready nodes
11:         for all ready node (u, p) in active do
12:             if u = v_d then
13:                 propagated ← true, pred ← p, step ← 0
14:             else
15:                 Enqueue each (v, w(u, v), u) into next active
16:             end if
17:         end for
18:         if no active nodes remain then
19:             return ∅                                              ▷ unreachable
20:         end if
21:     end while
22:     Prepend v_d to path; set v_d ← pred
23: end while
24: Prepend v_s to path
25: return path
```

As illustrated in the pseudocode, we use a boolean flag to indicate whether the spike has successfully propagated from the destination node v_d back to the source v_s. This flag signals the termination of the execution. We also rely on the precomputed α, which represents the sum of all finite delays in the graph, to define a timeout that ensures termination even in the worst-case scenario.

During each iteration of the inner loop simulating the SNC, we proceed as follows: (i) we gather all currently active nodes , those receiving a spike with a remaining delay , and decrement their delay by one. This is implemented using dictionaries where the destination node is the key, and the value is a tuple consisting of the origin node and the current delay; (ii) any node whose delay reaches zero is moved to a list of *ready* nodes; (iii) we then process each ready node by checking whether it is equal to v_d. If so, this signals the end of the current propagation cycle: the node is added to the solution path, its presynaptic neuron becomes the new v_d, and the network simulation resumes

until $v_d = v_s$. Otherwise, we propagate spikes to its neighbors, updating the active set accordingly.

An important remark is that in our simulation, one unit of time corresponds to a single iteration of the inner loop, during which all relevant reads and writes take place. As a result, edge weights must be represented as natural numbers in the range $[1, \infty)$. While continuous-time (floating-point) delays are theoretically possible, we adopt discrete time steps to enhance clarity and implementation simplicity.

Using edge weights as synaptic delays, however, introduces a serious practical limitation. In reality, the time complexity is effectively dictated by $\mathcal{O}\Big(\delta_{\max} \cdot h \cdot |E|\Big)$, since the number of iterations (i.e., spike propagations with respect to delay) is proportional to the edge weights. This raises concerns about scalability: if the magnitude of the weights lies within very large ranges, then the number of operations and iterations grows accordingly, potentially rendering the simulation computationally expensive.

To mitigate this, we propose a straightforward yet effective optimization: we compute the greatest common divisor (GCD) of all edge weights and divide each weight by this value. Since the division is guaranteed to be exact, this transformation preserves the relative timing structure of the graph while reducing the absolute size of the delays. Consequently, the overall number of simulation steps can be significantly reduced, leading to a substantial improvement in performance. In addition to applying the GCD method, we will test various heuristics and compression functions that bound the weight values. We will analyze the trade-off between performance gains and loss of precision in following sections.

Paper's Algorithm

The simulated approach of [53] is quite similar to ours. In both cases, signal propagation occurs through sequential updates of delay counters and dictionaries, effectively simulating unitary time steps. However, a key addition is the use of STDP-based back-tracing. To complete the proposed algorithm, it is necessary to track the edges through which propagation has occurred. This is because the fastest synapse to reach a neuron will trigger its spike and subsequently inhibit further activation due to its refractory period, marking that same synapse via STDP.

Note in the pseudocode that the simulation now consists of a single loop, which emulates the SNC dynamics and is analogous to our previous implementation. Nevertheless, no offsets are introduced here as they are not needed for determining ambiguous paths. We

Algorithm 3 Shortest Path via SNC Simulation (Paper)

Require: Directed graph $G = (V, E, w)$, source v_s, target v_d, timeout α
Ensure: Shortest path from v_s to v_d

1: $\mathit{fired}[v] \leftarrow$ False $\quad \forall v \in V, \quad \mathit{fired}[v_s] \leftarrow$ True
2: Initialize active set with $\{(u,\ w(v_s, u),\ v_s) \mid (v_s, u) \in E\}$
3: step $\leftarrow 0$
4: **while** v_d not reached **do**
5:     step $\leftarrow$ step $+ 1$
6:     **if** step $> \alpha$ **then**
7:         **return** $\emptyset$ ▷ timeout
8:     **end if**
9:     Decrease all counters in *active* by 1
10:     *ready* $\leftarrow \{v \in \mathit{active} \mid \text{counter} = 0\}$
11:     **for all** $v \in \mathit{ready}$ **do**
12:         **if** $\mathit{fired}[v] =$ False **then**
13:             $\mathit{fired}[v] \leftarrow$ True
14:             Mark edge (u, v): $w(u, v) \leftarrow -w(u, v)$
15:             **if** $v = v_d$ **then**
16:                 **break loop**
17:             **end if**
18:             **for all** $v' \in$ neighbors(v) **do**
19:                 **if** $\mathit{fired}[v'] =$ False $\wedge\ w(v, v') > 0$ **then**
20:                     Update $\mathit{active}[v']$
21:                 **end if**
22:             **end for**
23:         **end if**
24:     **end for**
25:     **if** *active* is empty **then**
26:         **return** $\emptyset$ ▷ unreachable
27:     **end if**
28: **end while**

29: **Backtracing Phase:**
30: Initialize $\mathit{path} \leftarrow [v_d]$, $\mathit{current} \leftarrow v_d$
31: **while** $\mathit{current} \neq v_s$ **do**
32:     Find u such that $w_{\text{orig}}(u, \mathit{current}) > 0 \wedge w(u, \mathit{current}) < 0$
33:     **if** such u exists **then**
34:         Prepend u to *path*
35:         $\mathit{current} \leftarrow u$
36:     **else**
37:         **return** `failure`
38:     **end if**
39: **end while**
40: **return** *path*

then introduce several key changes. First, we define a map called `fired` that records whether a neuron has spiked. This prevents a neuron from firing again, mimicking the refractory period.

Additionally, when a postsynaptic neuron spikes, the corresponding synaptic weight is marked as modified, simulating the effect of STDP. This is done by simply negating the weight, effectively using the sign as a flag to indicate that this synapse has been traversed.

Finally, the last major addition is the *Backtracing Phase*, which we have already explained. This step reconstructs the path by checking which synapses have been modified during the simulation.

This algorithm is similarly affected by the limitations imposed by delays being proportional to edge costs. Therefore, a preliminary scaling of the weights is also applied to ensure they remain within a relatively low and manageable range when possible.

Dijkstra's Algorithm

For the implementation of Dijkstra's algorithm, we employ a standard and widely adopted approach from the literature [3, 25], which relies on a heap-based priority queue. The time complexity of this implementation follows the well-established model of $\mathcal{O}(|E| + |V| \log |V|)$.

For energy estimations, we will compute within the simulation the number of heap accesses and the corresponding operations performed, providing a proxy for energy consumption in classical architectures.

Algorithm 4 Shortest Path via Dijkstra's Algorithm

Require: Directed graph $G = (V, E, w)$, source node s, target node t
Ensure: Shortest path from s to t

```
 1: Initialize dist[v] ← ∞ for all v ∈ V; dist[s] ← 0
 2: Initialize prev[v] ← null for all v ∈ V
 3: Initialize a min-heap H ← {(0, s)}
 4: while H is not empty do
 5:     Pop (d, u) from H
 6:     if d > dist[u] then
 7:         continue
 8:     end if
 9:     if u = t then
10:         break
11:     end if
12:     for all (v, w) ∈ neighbors(u) do
13:         if dist[u] + w < dist[v] then
14:             dist[v] ← dist[u] + w
15:             prev[v] ← u
16:             Push (dist[v], v) into H
17:         end if
18:     end for
19: end while

20: Path Reconstruction:
21: Initialize path ← [ ], u ← t
22: while u ≠ s do
23:     Prepend u to path
24:     u ← prev[u]
25:     if u = null then
26:         return ∅                                    ▷ no path found
27:     end if
28: end while
29: Prepend s to path
30: return path
```

3.2.2. Weight compression methods

To overcome the limitations associated with having synaptic delays strictly proportional to the edge costs of a graph, we propose a set of compression techniques aimed at reducing the absolute values of delays while preserving their relative order, or at least minimizing any resulting imprecision in that ordering.

As previously discussed, we initially employed the greatest common divisor (GCD) of all delays as a baseline method. However, this approach often results in no effective

reduction, especially in the presence of prime numbers, in which case the GCD equals 1, a rather common scenario. Therefore, in what follows, we put aside the GCD and introduce alternative techniques.

Mapping

The mapping method is the simplest of all proposed techniques. Let $D = \{d_1, d_2, \ldots, d_n\}$ be the multiset of synaptic delays associated with the edges of the graph, and let $\mu = |\{\text{unique elements of } D\}|$. We first extract the set of unique delays,

$$U = \{u_1, u_2, \ldots, u_\mu\},$$

and sort it in ascending order:

$$u_{(1)} < u_{(2)} < \cdots < u_{(\mu)}.$$

We then define a bijective mapping

$$f : U \longrightarrow \{1, 2, \ldots, \mu\}$$

by

$$f\Big(u_{(i)}\Big) = i, \quad \forall\, i \in \{1, \ldots, \mu\}.$$

Finally, each original delay $d_j \in D$ is replaced by its rank $f(d_j)$.

This method guarantees preservation of the relative ordering, as the mapping function f is strictly increasing over the domain U. Consequently, the error introduced in terms of ordering is exactly zero.

Compression is effective only when two conditions are satisfied. First, the number of unique delays μ must be significantly smaller than the total number of delays n. Second, the index range produced by the mapping, namely $[1, \mu]$, should be substantially smaller than the original range $[1, n]$.

In our scheme, the mapping step replaces each distinct delay value with an integer label in $\{1, 2, \ldots, \mu\}$. If both conditions hold, the representation depends only on the number of unique values rather than on the total size of the graph, which results in a meaningful reduction in storage requirements. However, in the worst case where $\mu = n$ and its ranges are equal or similar, every edge has a distinct delay value within similar magnitudes. In this situation where the mapped labels practically span the same range as the original

values, the compression ratio becomes negligible.

The preprocessing time complexity of this technique is $\mathcal{O}(n + \mu \log \mu)$, accounting for the cost of extracting and sorting the unique elements in U, as well as constructing the bijective mapping and applying it to the original set of delays.

However, although this technique appears to preserve relative ordering, i.e., if $a < b$, then $f(a) < f(b)$, this property does not extend to addition. For example, consider the mapping $1 \mapsto 1$, $3 \mapsto 2$, and $10^7 \mapsto 3$ (therefore with $\mu = 3$). Under this transformation, a path such as $2 \to 2 \to 2$ would have a total cost of 6, which is actually greater than the cost of the path $3 \to 1 \to 1$, which sums to 5 under this mapping, when in reality it is not.

This outcome is misleading, as the first path traverses edges that are several orders of magnitude smaller than those in the second. Specifically, the contribution of the edge with original weight 10^7 is severely understated. As a result, the summed mapped weights are no longer a meaningful proxy for actual cost, and the transformation discards essential magnitude information, ultimately distorting path evaluation. However, we will implement this weight compression method to address its efficiency and loss of precision.

Logarithmic Scale

This compression technique is widely known, as applying logarithmic functions flattens large values while preserving the relative scale of smaller ones, thanks to the properties of logarithms.

In our case, we apply the logarithm to the weights of the graph. We chose the base-10 logarithm, which provides a relatively strong compression effect. However, if a more moderate compression is desired, one can simply reduce the base of the logarithm accordingly.

$$\tilde{w}_i = \log_{10}(w_i) \tag{10}$$

However, although logarithmic transformation preserves the relative order of weights, as in the previous method, since $w_i > w_j \Rightarrow \log(w_i) > \log(w_j)$, it does not preserve additive relationships. As a result, path comparisons based on the sum of logarithmic weights can introduce false ties or produce virtual paths that appear shorter than others, even when they are not in the original weight space. This may distort the precision and resolution of the solution to the SPP.

Truncation

The truncation method consists of reducing the precision of edge weights by mapping them to the nearest lower multiple of a given compression factor q. This is achieved by first scaling each original weight w by q^{-1}, truncating the result (i.e., taking the floor), and then scaling it back by multiplying with q. This process introduces an unavoidable loss of precision determined by the value of q: the larger q is, the more severe the compression and the coarser the resulting weight distinctions.

The choice of q may be guided by external constraints, such as a maximum path length $L_{\max}$ and the number of bits available for representing weights (i.e., a maximum representable range R). The goal is to ensure that the accumulated truncation error over any path of length at most $L_{\max}$ remains within acceptable bounds.

The truncation mapping is defined by the following equation:

$$\tilde{w} = q \cdot \left\lfloor \frac{w}{q} \right\rfloor$$

where w is the original weight, and w' is the compressed (truncated) weight.

Optionally, if one prefers rounding to the nearest multiple of q rather than truncation, the mapping can be modified as:

$$\tilde{w} = q \cdot \left\lfloor \frac{w + q/2}{q} \right\rfloor$$

This variant reduces the maximum absolute error per weight from q to $q/2$, though it may still affect shortest path consistency depending on the graph structure and weight distribution. Unfortunately, we do not implement this approach in our experiments, as we reserve it for future work.

3.2.3. Hypotheses

Once the theoretical foundations have been established and implemented, we can formulate a set of hypotheses based on the nature of the different approaches. In our case:

1. **Expected advantage in large and dense graphs.** We anticipate an advantage in large graphs with high density. This is because our method does not require inspecting all the synapses along the path, nor does it perform computations proportional to the number of edges or vertices (but rather the delays), as in Dijkstra's algorithm. Instead, we execute the network h times, and the check of each neuron

that belongs to the resulting path after each execution is performed in constant time $\mathcal{O}(1)$.

2. **Expected disadvantage with large or highly dispersed weights (unbounded synaptic delays).** A potential drawback arises when edge weights are highly dispersed or large in magnitude. In such cases, our method may be less efficient than both the baseline paper (since we must execute the network h times and accumulate delay-related costs) and Dijkstra's algorithm (which is independent of the actual edge weights).

3. **Expected disadvantage in small or sparse graphs.** For small graphs or graphs with very low density, our algorithm may also underperform compared to the other two approaches. However, this disadvantage is likely to depend on the specific distribution and magnitude of the edge weights as well.

4. Experimental methodology

In this section, we distinguish between different experimental objectives depending on the nature of the comparison being pursued. (i) The first objective is to evaluate the efficiency of each algorithm, considering both temporal and energy aspects. This involves assessing the overall execution time and estimating the energy consumption under a variety of network configurations. (ii) The second objective focuses on the study of weight compression heuristics and their impact on performance. Specifically, we investigate how these heuristics affect the trade-off between computational efficiency and the precision of the results.

Each of these objectives will be addressed through dedicated analyses, where we will specify the experimental conditions, expected outcomes, and the criteria for evaluation. Additionally, we will define the metrics and estimation models used throughout the study. To ensure statistical robustness and generalization, we also describe the types of graphs used in the evaluation, along with the details of their pseudo-random generation process.

4.1. Graph Generation

To ensure fair and reproducible comparisons across a wide variety of graph types, we developed a script that generates a large set of graphs based on several parameterized characteristics. The full parameter grid is shown in Table 2. For each unique combination of parameters, the number of graphs to generate and a random seed must also be specified, enabling consistent replication of results.

Table 2: Parameter grid for synthetic graph generation.

Parameter	Value	Description
size	small	20 nodes
	medium	50 nodes
	large	200 nodes
	ultraLarge	1000 nodes
density	superSparse	Edge probability = 0.01
	sparse	Edge probability = 0.05
	medium	Edge probability = 0.25
	dense	Edge probability = 0.5
	superDense	Edge probability = 0.9
weight_dist	uniform	Uniform distribution $\mathcal{U}(1, 10)$
	exp	Exponential distribution Exp(5)
	const	Constant weights equal to 1
	highdelays	Uniform distribution $\mathcal{U}(100, 1000)$

This setup yields a total of 400 graphs, as we generate 5 distinct graphs for each parameter combination. These graphs will be used to evaluate the performance of each proposed approach to solving the SPP.

4.2. Estimations

In this subsection we will delve into our estimates from the simulations.

4.2.1. Time estimates

We begin defining two different execution time estimates: an ideal and a realistic one. The ideal simulation assumes infinite parallelism, no delays, and one-to-one connectivity. Under these assumptions, the execution time can be calculated based on the number of ticks (i.e., the delay cycles occurring on the chip) and the neuron update rate, also referred to as the cycle time. All hardware implementation data are based on Intel's Loihi neuromorphic architecture.

We define the ideal time as:

$$t_{\text{ideal}} = ticks \times \tau \times 10^3 \ (ms), \quad \tau = \frac{1}{10MHz} \tag{11}$$

With τ being the reported clock cycle for Loihi chips.

Now, to obtain a realistic estimate, based on the previous equation, several factors must

be taken into account. First, each tick can produce a varying number of spikes depending on neural activity and network topology. However, for simplicity, given the diversity of graphs, we assume this value to be constant. Thus, the number of spikes generated per tick is modeled as:

$$E = \alpha N \tag{12}$$

where E denotes the number of spike events per tick, α is the neuronal activity factor, and N is the total number of neurons in the network. We set α as the average spiking neuronal activity reported in each graph case per cycle.

Next, we must consider one of the main practical limitations of neuromorphic systems: the size of the communication bus and potential congestion. Intel's neuromorphic chips (e.g., Loihi) consist of 64 to 128 processing elements (PEs), which are small computational units responsible for managing neuron states and spike communication. Accordingly, we assume that the bus can handle between 64 and 128 events per communication cycle without major contention.

Moreover, congestion and arbitration introduce additional latency per group of spikes. Based on the estimates provided in [40], we consider a bus latency in the range of 50 to 70 nanoseconds per group of events.

Since we assume that all neurons are processed within a single neuromorphic chip (e.g., Loihi), the number of chips is not a contributing factor to communication latency in our setup. This is justified because our graphs are small enough to fit within a single chip, and Loihi can easily accommodate the total number of neurons required.

Given these considerations, we model the bus latency per tick as follows:

$$B = \frac{E + (C - 1)}{C} \times \ell_{\text{bus}} \tag{13}$$

where $\ell_{\text{bus}} = 50$ ns represents the latency per communication cycle, and $C = 64$ is the bus capacity in number of events per cycle. This expression captures the cost of transmitting E events across the bus, assuming each bus cycle can carry up to C events.

Next, we must take into account the internal latencies associated with memory access, synchronization, and routing. Each event incurs a non-negligible processing delay, which must be explicitly modeled in the execution time. According to [10], neuromorphic FPGA architectures can process one event every 7 clock cycles on average, considering all internal operations involved.

Based on this, we define the computation cost per tick as:

$$C_{\text{comp}} = \varsigma \cdot \frac{E}{C} \tag{14}$$

where C_{comp} is the total number of computation cycles required per tick, $\varsigma = 7$ is the number of processing cycles per event as reported in [10], E is the number of events generated per tick, and $C = 64$ denotes the number of PEs available in the system.

Finally, we incorporate the clock cycle duration τ described earlier and compute the actual time per tick. This includes both the communication cost B and the computational cost C_{comp}, as well as a fixed initialization and synchronization overhead:

$$t_{\text{real}} = \varrho \times (\textit{ticks} \times (B + C_{\text{comp}} \cdot \tau \cdot 10^3) + \vartheta) \tag{15}$$

Here, ϑ denotes a fixed overhead of 300 ns, which also accounts for accesses to the neurons state necessary for path reconstruction. And ϱ accounts for some initial synchronization factor which we set to 1.25. This formulation provides a relatively realistic estimate of the execution time of our neuromorphic algorithm.

Although the estimation method that we use for the reference paper follows a similar formulation, it includes an additional refinement. Specifically, we must include the temporal complexity of the back-tracing phase in the paper's estimation, which involves accessing each neuron to compare all synapses and determine which ones were modified by STDP. Therefore, in addition to the initial propagation function previously described, we define a back-tracing function that estimates the approximate number of synapses to be checked:

$$N_{\text{syn}} = h \cdot \delta_g \cdot N, \quad \delta_g \in [0, 1] \tag{16}$$

where δ_g is the graph density, i.e., the proportion of synapses per node (edges divided by $V \times (V - 1)$), and h is the length of the shortest path, i.e., the number of neurons to traverse.

We then apply the delay introduced by the bus, considering its capacity and the costs associated with arbitration and routing:

$$B_2 = \frac{N_{\text{syn}} + (C - 1)}{C} \times \ell_{\text{bus}} \tag{17}$$

Finally, we calculate the computational and comparison cost:

$$C_{\text{comp2}} = \varsigma \times \frac{N_{\text{syn}}}{C} \tag{18}$$

The execution time of this phase is then given by:

$$t_{\text{real}} = \varrho \cdot (\underbrace{ticks \times (B + C_{\text{comp}} \cdot \tau \cdot 10^3)}_{\text{Initial propagation}} + \underbrace{(B_2 + C_{\text{comp2}} \cdot \tau \cdot 10^3)}_{\text{Backtracing phase}} + \vartheta) \tag{19}$$

We will use equations (15) and (19) as the time performance estimates for our algorithm and the reference paper's algorithm, respectively, from this point onward.

4.2.2. Energy consumption estimates

We base our energy consumption estimate on the per-event costs reported in [53] (Table 3).

Table 3: Energy per event type, reproduced from [53].

Event	**Neuron (pJ)**	**Synapse (pJ)**
Accumulation	9.81	1.45
Fire	12.5	—
Learning (STDP)	—	2.58
Idle	7.20	0.07

In the spiking neuromorphic hardware benchmark described in [53], energy usage is inferred from counts of accumulation, firing, and idle events for both neurons and synapses over a discrete-event simulation. In our model, synaptic potentiation (STDP) is omitted. Let $|V|$ and $|E|$ denote the total numbers of neurons and synapses, respectively, and let β be the total number of simulated clock cycles. If a fraction α of neurons fire each cycle, then the total numbers of events satisfy

$$N_{\text{fire}} \;=\; \alpha\,\beta\,|V|, \quad N_{\text{acc}} \;=\; \alpha\,\beta\,|E|,$$

$$N_{\text{idle,neuron}} \;=\; \beta\,|V| - N_{\text{fire}}, \quad N_{\text{idle,syn}} \;=\; \beta\,|E| - N_{\text{acc}}.$$

Once again, we set α to the mean of the spikes per cycle, consistent with our temporal cost estimate, and identify $\beta = \texttt{ticks}$, the number of simulation steps. Using the per-event energies from Table 3,

$$E_{\text{fire}} = 12.50 \text{ pJ}, \quad E_{\text{acc}} = 1.45 \text{ pJ}, \quad E_{\text{idle,neuron}} = 7.20 \text{ pJ}, \quad E_{\text{idle,syn}} = 0.07 \text{ pJ},$$

the total energy consumed over β cycles is:

$$E_{\text{total}} = N_{\text{fire}}\, E_{\text{fire}} + N_{\text{acc}}\, E_{\text{acc}} + N_{\text{idle,neuron}}\, E_{\text{idle,neuron}} + N_{\text{idle,syn}}\, E_{\text{idle,syn}} \tag{20}$$

To enable a fair comparison between our neuromorphic implementation and a classical approach, we estimate the energy cost of Dijkstra's algorithm based on values reported in the literature, particularly by Horowitz [28]. According to these measurements, a single DRAM access consumes approximately 1 to 2 nJ on average, which includes the cost of fetching data (typically 32 bytes) from off-chip memory along with I/O and bus overheads. Arithmetic or comparison operations typically require around 100 pJ each, which includes the energy consumed by instruction fetch and decode, as well as execution in the arithmetic logic unit (ALU).

Heap operations such as `pop` and `push`, which are central to Dijkstra's algorithm, involve traversing a binary heap with $\mathcal{O}(\log N)$ complexity. Each level of the heap requires a few SRAM accesses (with a typical energy cost of 5–20 pJ each depending on the memory hierarchy), a comparison, and control logic. For graphs with up to 10^3 nodes (which are the largest from our dataset), the number of heap levels is limited to around 10, leading to an estimated energy cost of approximately 1 nJ per heap operation. This value balances the cost of multiple memory accesses, ALU instructions, and overhead per level, and is consistent with reported microarchitectural energy costs.

These estimates provide a reasonable and literature-grounded framework for quantifying the energy consumption of conventional Dijkstra-based pathfinding algorithms. While actual costs may vary depending on the specific processor architecture and memory subsystem, this modeling allows us to compare relative energy efficiencies between neuromorphic and traditional computing paradigms under realistic assumptions.

Based on the energy estimates discussed above, we model the total energy consumption of Dijkstra's algorithm using the following expression:

$$E_{\text{total}} = (N_{\text{read}} + N_{\text{write}}) \cdot E_{\text{mem}} + N_{\text{cmp}} \cdot E_{\text{cmp}} + (N_{\text{heap_pop}} + N_{\text{heap_push}}) \cdot E_{\text{heap}}, \tag{21}$$

where $E_{\text{mem}} = 2000\,\text{pJ}$ represents the energy per memory access (read or write), $E_{\text{cmp}} = 100\,\text{pJ}$ corresponds to the energy cost of a simple arithmetic or comparison operation, and $E_{\text{heap}} = 1000\,\text{pJ}$ reflects the estimated cost of a heap operation such as `pop` or `push`. The variables $N_{\text{read}}, N_{\text{write}}, N_{\text{cmp}}, N_{\text{heap_pop}}$, and $N_{\text{heap_push}}$ denote the number of each type of operation executed by the algorithm.

4.3. Performance experiments

To assess correctness, we simply run the simulations and verify whether the shortest paths obtained match those produced by Dijkstra's algorithm. This validation can be performed either by comparing the exact sequence of nodes or by checking whether the alternative paths found by our method yield the same cumulative cost, thus demonstrating functional equivalence.

In terms of runtime performance, we apply the time estimation framework detailed in Section 4.2 and compare the results across all approaches. Specifically, we examine three aspects: first, in how many graphs each approach is faster; second, the relative percentage improvement achieved by our method with respect to the others; and third, a statistical characterization of the observed improvements. For the latter, we compute descriptive statistics including means, medians, inter-quartile ranges, skewness, and kurtosis. This allows us to analyze the global performance distribution and identify systematic trends or outliers in the comparative results.

To validate our hypotheses regarding performance advantages under specific graph topologies, we design a set of experiments aimed at identifying the types of graphs where our approach exhibits superior behavior. We categorize graphs according to the parameter grid shown in Table 2, and analyze the resulting performance patterns through statistical distribution analysis.

Furthermore, we employ dimensionality reduction techniques such as Principal Component Analysis (PCA) to identify the most influential topological features, as we also investigate partial correlations between graph properties and performance metrics. These tools help us capture nonlinear dependencies and highlight which graph characteristics most significantly affect algorithmic efficiency.

In conclusion we have 4 main experiments:

Table 4: Performance experiments.

Experiment	Subexperiment	Description
Correctness	Path equivalence	$\text{Correctness} = \begin{cases} 1 & \text{if } \texttt{path}_{\text{our}} = \texttt{path}_{\text{ref}} \\ 1 & \text{if } \texttt{cost}_{\text{our}} = \texttt{cost}_{\text{ref}} \\ 0 & \text{otherwise} \end{cases}$ Path = sequence; Cost = total weight.
Runtime	Faster instances	$\text{Faster}_{\text{our}} = \sum_{g=1}^{G} \mathbb{I}(t_{\text{our}}^{(g)} < \min(t_{\text{paper}}^{(g)}, t_{\text{dijkstra}}^{(g)}))$
	Relative improvement	$\Delta t_{\text{vs } b} = \frac{t_b - t_{\text{our}}}{t_b} \times 100$
	Distribution statistics	Median: $Q_{0.5}$, IQR: $Q_{0.75} - Q_{0.25}$ Skewness: $\frac{\mathbb{E}[(X-\mu)^3]}{\sigma^3}$, Kurtosis: $\frac{\mathbb{E}[(X-\mu)^4]}{\sigma^4} - 3$
Category analysis	Parameter grid profiling	Performance = f(size_cat, density_cat, dist_cat)
Multivariate analysis	PCA (reduction)	$\mathbf{Z} = \mathbf{XW}$ (via eigenvectors)
	Partial correlations	$\rho_{xy\cdot z} = \frac{\rho_{xy} - \rho_{xz}\rho_{yz}}{\sqrt{(1-\rho_{xz}^2)(1-\rho_{yz}^2)}}$

4.4. Energy experiments

For the energy experiments, we will compute a set of relevant statistical measures to better understand the neuromorphic energy consumption in comparison to that estimated for Dijkstra's algorithm. In both cases, we rely on a series of estimations whose parameters are derived from the previously described simulations, based on existing literature. Specifically, we will calculate the mean, median, standard deviation, and interquartile ranges for both approaches, and highlight the most notable differences between them in terms of microjoules.

4.5. Weight compression tests

The compression tests are limited to the *mapping* and *logarithmic scaling* techniques, selected for their relative simplicity and their ability to address the limitations imposed by synaptic delays proportional to edge costs in this type of neuromorphic approach. For both techniques, the same simulations described in Section 4.2 will be executed, applying one of the aforementioned scaling strategies (which is not included in either the temporal or energy estimations).

For comparison purposes, we use the original GCD-based heuristic as the baseline, alongside the *ranking* or *mapping* heuristic, and the *logarithmic scaling* approach. We will compute average and total improvements, among other metrics, as well as performance

ratios that relate the efficiency gains to the corresponding loss in precision. Using these statistical measures, we will analyze the results and observe how graph distributions in terms of time and energy are affected by the compression methods, thereby enabling a more refined characterization of their behavior.

5. Results and discussion

Here we discuss the three experiments presented in the previous section, focusing on the results in terms of temporal efficiency, energy consumption, and the trade-offs introduced by the synaptic-delay compression techniques.

5.1. Performance results

The results concerning correctness show that, across all 400 graphs, our algorithm consistently finds paths that match those of Dijkstra's algorithm. In the few cases where the paths do not coincide, the alternative found is of equal cost.

Regarding speed, using the estimations from equations (15) for our algorithm and (19) for the reference algorithm from the paper, we observe the following: our algorithm is faster in 141 cases (35.25%), the reference algorithm is faster in 234 cases (58.50%), and Dijkstra's algorithm is faster in 25 cases (6.25%).

Table 5: Summary of Top-5 improvements/degradations against Reference Paper and Dijkstra's algorithm. Results are filtered to exclude extreme outliers, ensuring that improvements or degradations above 1000% do not skew the metrics.

(a) Top 5 Improvements vs. Paper

Set	ID	% Imp.	N	Density
ultraLarge_superDense_const	1	62.96	1000	0.90
ultraLarge_superDense_const	2	62.96	1000	0.90
ultraLarge_superDense_const	4	62.96	1000	0.90
ultraLarge_superDense_const	5	62.96	1000	0.90
ultraLarge_superDense_exp	3	62.96	1000	0.90

(b) Top 5 Degradations vs. Paper

Set	ID	% Imp.	N	Density
ultraLarge_superSparse_hd	4	−964.14	1000	0.01
ultraLarge_sparse_hd	2	−939.73	1000	0.05
ultraLarge_medium_hd	1	−932.73	1000	0.25
large_superSparse_uniform	1	−932.44	200	0.01
ultraLarge_superSparse_exp	1	−919.09	1000	0.01

(c) Top 5 Improvements vs. Dijkstra

Set	ID	% Imp.	N	Density
ultraLarge_superDense_const	5	100.00	1000	0.90
ultraLarge_superDense_const	4	100.00	1000	0.90
ultraLarge_superDense_const	2	100.00	1000	0.90
ultraLarge_superDense_const	1	100.00	1000	0.90
ultraLarge_superDense_const	3	100.00	1000	0.90

(d) Top 5 Degradations vs. Dijkstra

Set	ID	% Imp.	N	Density
medium_sparse_hd	1	−981.81	50	0.05
medium_sparse_hd	3	−886.66	50	0.05
small_superSparse_hd	2	−836.82	20	0.05
small_medium_hd	5	−634.46	20	0.26
small_medium_hd	4	−326.88	20	0.24

As illustrated in Table 5, we begin to identify the graph families where our method performs better or worse compared to other approaches.

In the comparison with the reference algorithm, we see a significant advantage on very large graphs of great densities with constantly distributed weights, i.e., low-variability and low-magnitude weights. Against Dijkstra's algorithm, this trend becomes even more consistent: the five most significant improvements also correspond to graphs with 1000 nodes, the largest in our dataset, again across great density types and primarily featuring low, constant weights. In general, the relative gains are greater against Dijkstra, reaching up to 100% improvement, whereas against the reference algorithm the top gain is 63%, decreasing thereafter.

This is because Dijkstra is at a disadvantage in our evaluation: its performance is based on actual runtime in Python rather than a custom hardware execution estimate. In reality, Dijkstra could gain a competitive edge if implemented natively in hardware, where the cycles and operations required by conventional computing are orders of magnitude higher than in neuromorphic equivalents.

In the case of Dijkstra, the worst degradations occur on small, sparse graphs with high delays, which aligns well with our initial hypotheses. A more detailed analysis is still needed to fully assess performance across all algorithms, but this already highlights its main disadvantages and improvements.

In the case of the reference paper, the worst performance degradations occur in large graphs, which may initially seem counterintuitive. This phenomenon is explained by the significant sparsity (very low density (1-25%)) observed in the worst-performing graphs: in such cases, the reference algorithm only needs to verify a minimal number of synapses during the back-tracing step, whereas our approach requires executing the network h times. Another possible contributing factor is the lower average congestion on the communication bus. This is likely due to the refractory period established by the neurons in the reference model, which effectively limits the neuronal activity during our estimation of α, reducing the average spiking frequency of the neural populations per cycle.

Both Table 6 and Figure 7 provide a detailed characterization of the performance differences with respect to the baseline algorithm from the reference paper. We observe a highly negative mean improvement of -155.93%, along with a negative median, indicating that in most cases, the reference paper achieves better results. However, the distribution also exhibits considerable variability, as reflected by a large standard deviation and a kurtosis value of 11.2, suggesting the presence of heavy tails and extreme outliers.

Figure 7 provides a visual summary of both performance improvements and degradations. The histogram reveals a bimodal distribution, supported by a skewness value of

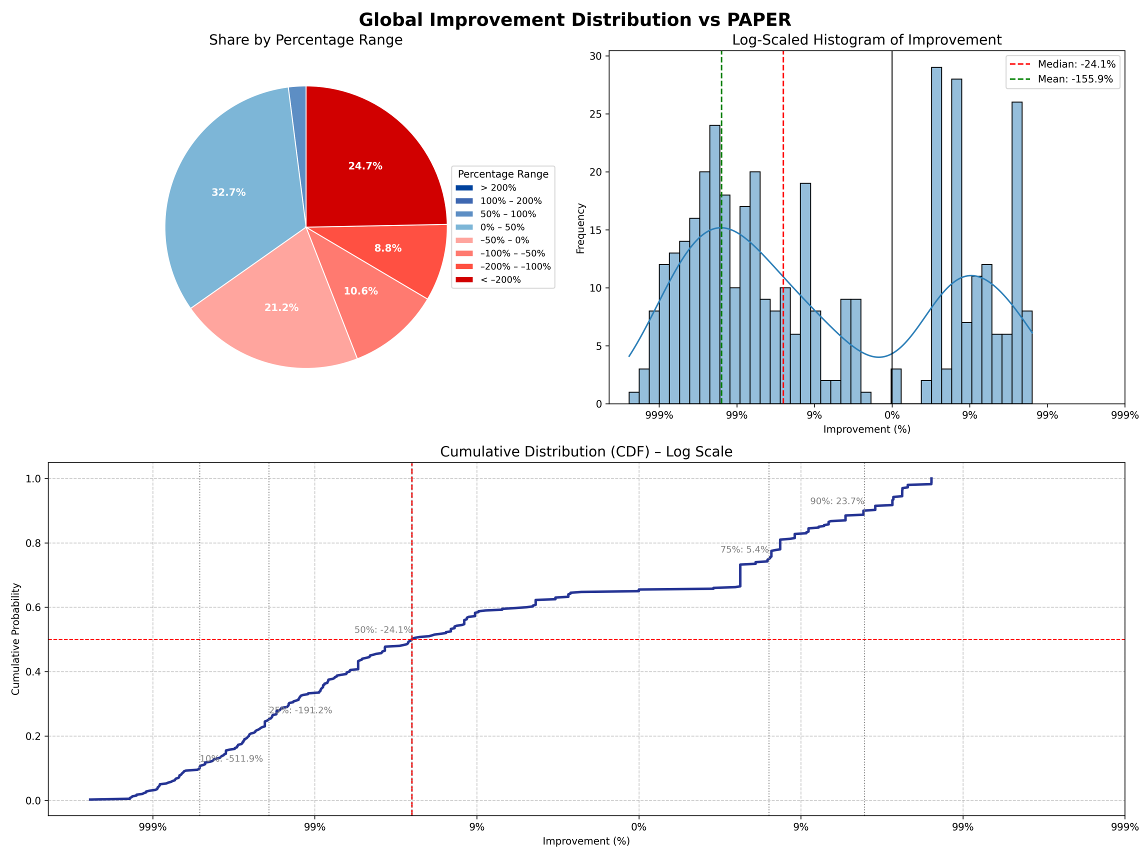


Figure 7: Global improvement distribution compared to the baseline algorithm from the reference paper. The figure includes: (Left) a pie chart representing the proportion of graphs across predefined improvement/degradation ranges, (Top right) a histogram using a symmetric logarithmic scale to visualize the frequency of improvements, and (Bottom) a cumulative distribution function (CDF) showing the overall improvement trend. The log scale mitigates the influence of extreme outliers (improvements or degradations exceeding 1000%), while median and mean markers aid in interpreting the central tendency and skewness of the distribution.

Table 6: Descriptive statistics and improvement distribution against the reference paper.

Statistic	**Value (%)**
Summary statistics	
Mean improvement	–155.93
Median improvement	–24.13
Standard deviation	302.50
Interquartile range (IQR)	196.52
Skewness	–2.90
Kurtosis	11.20
Improvement distribution	
Graphs with improvement	35.2 (141 graphs)
Graphs with degradation	64.8 (259 graphs)
Significant improvement (>50%)	2.0
Significant degradation ($< -50\%$)	43.8
Extreme cases ($\|\Delta\| > 200\%$)	24.5

–2.9, indicating a strong leftward bias. Two distinct peaks are evident, reflecting the approximate 60–40 split between degraded and improved instances. Notably, the peak in the negative region is both higher and located farther from the origin (0%), suggesting that degradations are not only more frequent but also more severe in magnitude than the observed improvements. This visual interpretation is consistent with the summary statistics, reinforcing the distribution's asymmetry and heavy-tailed nature.

Now, performing the same analysis with respect to Dijkstra yields the more favorable statistics presented in Table 7, where the majority of improvements hover around 98%. However, as shown in Figure 8, the distribution exhibits heavy tails, corroborated by a very high kurtosis value of 71.18, due to a few cases of markedly severe degradations. Similar to the paper comparison, there exist instances where the magnitude of deterioration far exceeds the percentage of the improvements.

However, in this case, the frequency of improvements is substantial, overwhelmingly surpassing the cases of degradation. Nearly 90% of the cases demonstrate improvement. It is important to bear in mind that this estimation pertains to neuromorphic hardware, which is a direct implementation of the algorithm; in reality, Dijkstra should operate faster on conventional processors, as they achieve much higher speeds than neuromorphic chips. Regardless, this analysis serves better as a characterization method rather than an absolute benchmark, since the relative improvement depends on the graph type, providing deeper insight into the behavior of our algorithm.

We now proceed to dissect these data by categories, starting with size, followed by density,

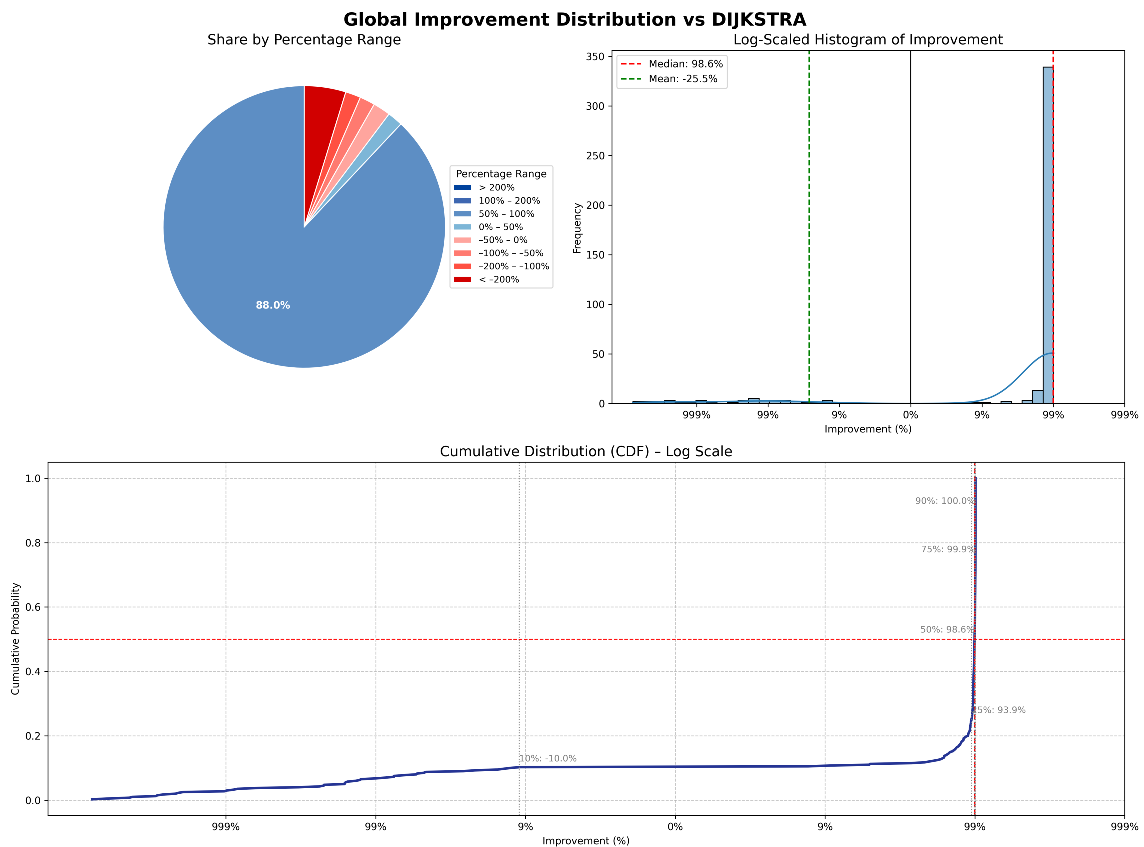


Figure 8: Global improvement distribution compared to Dijkstra algorithm The figure includes: (Left) a pie chart representing the proportion of graphs across predefined improvement/degradation ranges, (Top right) a histogram using a symmetric logarithmic scale to visualize the frequency of improvements, and (Bottom) a cumulative distribution function (CDF) showing the overall improvement trend. The log scale mitigates the influence of extreme outliers (improvements or degradations exceeding 1000%), while median and mean markers aid in interpreting the central tendency and skewness of the distribution.

Table 7: Descriptive statistics and improvement distribution against Dijkstra's algorithm.

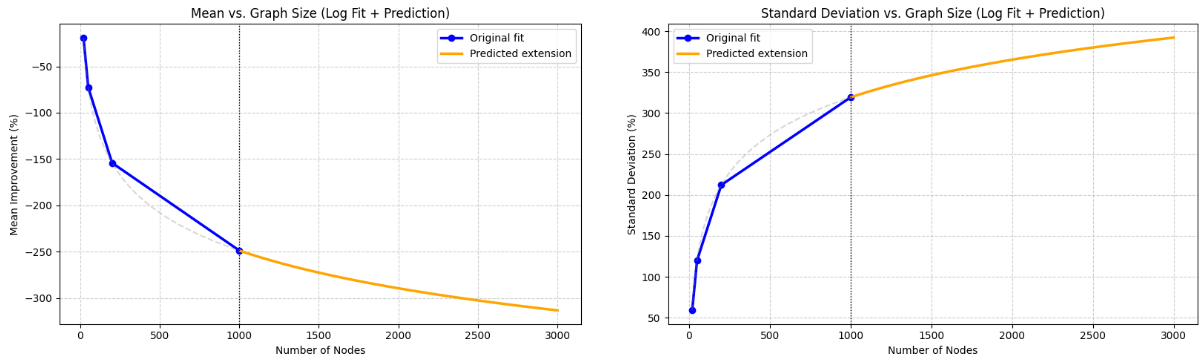
Statistic	**Value (%)**
Summary statistics	
Mean improvement	–25.47
Median improvement	98.56
Standard deviation	669.24
Interquartile range (IQR)	5.93
Skewness	–7.89
Kurtosis	71.18
Improvement distribution	
Graphs with improvement	89.8 (359 graphs)
Graphs with degradation	10.2 (41 graphs)
Significant improvement (>50%)	88.0
Significant degradation ($<-50\%$)	8.2
Extreme cases ($\lvert\Delta\rvert > 200\%$)	4.8

and concluding with weight distribution:

Table 8: Performance vs Paper Summary by `SIZE` Category (Filtered Extremes)

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
Small	100	-2.27	-27.68	55.92	42.00	0.00	56.00	39.00
Medium	100	-24.29	-75.12	129.28	29.00	13.00	71.00	59.00
Large	99	-40.91	-128.55	201.82	36.36	35.35	62.63	60.61
UltraLarge	89	-121.13	-263.21	323.01	34.83	21.35	65.17	65.17

All percentages refer to filtered graphs. Positive ($\Delta > 0$), Significant Improvement ($\Delta > 50\%$), and Severe Degradation ($\Delta < -50\%$) rates shown per category.



(a) Mean Improvement (%) vs. N. of Nodes

(b) Standard Deviation vs. N. of Nodes

Figure 9: (a) Mean improvement percentage with logarithmic fit and prediction. (b) Standard deviation behavior with logarithmic fit and prediction. Both plots show trends up to 3000 nodes.

It is insightful to examine Table 8 alongside Figure 9, where a clear pattern emerges: both the median and mean improvements are negative for smaller graphs, but they increase proportionally with the number of nodes. The same trend holds for the standard deviation. This suggests that, in smaller graphs, performance differences are less pronounced and remain relatively close to the mean within a certain range.

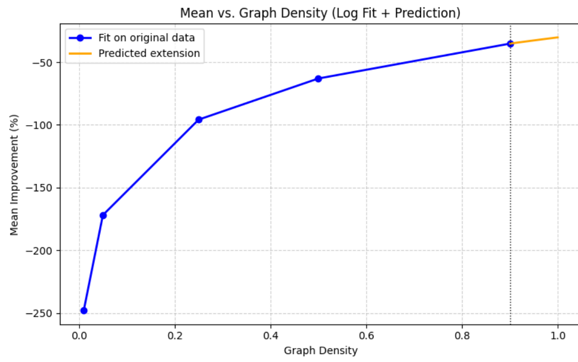
The growth of the mean and median appears to plateau, following an inverse logarithmic trend. In contrast, the standard deviation follows a logarithmic pattern, increasing steadily but at a slower rate than the number of nodes.

Table 8 further indicates that the highest proportion of significant improvements occurs predominantly in large and ultralarge graphs. However, these categories are also where the most substantial degradations are observed, highlighting the increasing variability in performance as graph size grows.

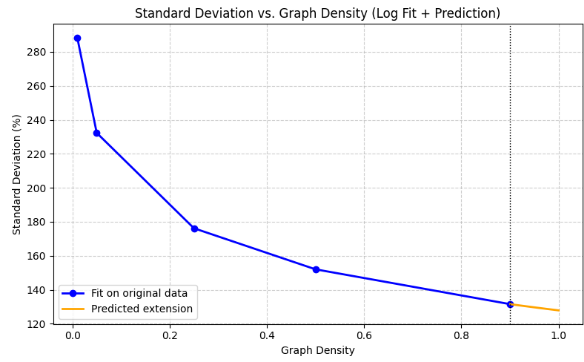
Table 9: Performance vs Paper Summary by `DENSITY` Category (Filtered Extremes)

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
SuperSparse	71	-50.00	-236.49	311.18	46.48	8.45	53.52	50.70
Sparse	77	-143.98	-187.16	189.82	18.18	0.00	81.82	79.22
Medium	80	-39.39	-105.48	192.34	3.75	2.50	92.50	91.25
Dense	80	-3.33	-55.93	151.13	43.75	20.00	56.25	32.50
SuperDense	80	10.14	-28.98	135.98	66.25	53.75	33.75	25.00

All percentages refer to filtered graphs. Positive ($\Delta > 0$), Significant Improvement ($\Delta > 50\%$), and Severe Degradation ($\Delta < -50\%$) rates shown per category.



(a) Mean Improvement (%) vs. Density



(b) Standard Deviation vs. Density

Figure 10: (a) Mean improvement percentage with logarithmic fit and prediction. (b) Standard deviation behavior with logarithmic fit and prediction. Both plots show trends up to density = 1 (Complete graph).

For the *density* category, as shown in Table 9 and Figure 10, the functions behave inversely compared to the *size* category, which is to be expected. This strongly supports the idea that the higher the density, the more edge performance we obtain. The mean improvement increases logarithmically, while the standard deviation decreases in an inversely logarithmic manner.

Moreover, as observed in Table 9, the median improvement becomes positive in the case of the superdense graphs. The majority of significant improvements occur predominantly in dense and superdense graphs, whereas the worst results are found in supersparse, sparse, and medium-density graphs. This aligns with the observation that the low standard deviation in dense graphs makes improvements more typical, while degradations become increasingly atypical.

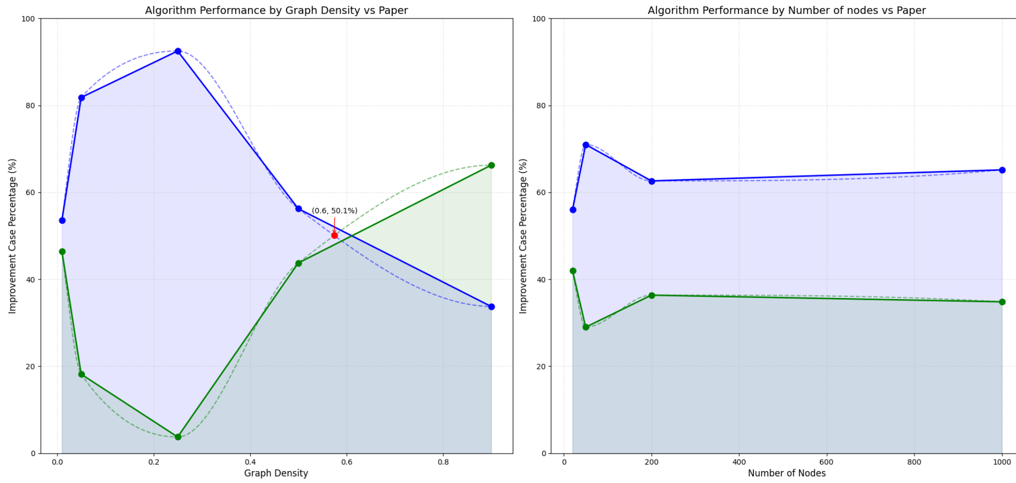


Figure 11: Algorithm performance vs paper distribution by graph density (Left) and by number of nodes (Right). Green regions indicate cases where our algorithm shows superior performance ($\Delta > 0$), while blue regions favor the reference implementation ($\Delta < 0$). The red marker identifies the crossover point where performance dominance shifts.

Note: All percentages reflect filtered data. The left figure points the intersection on 0.6 density.

As illustrated in Figure 11, our algorithm demonstrates a clear trend of increasing relative performance as graph density grows. A notable inflection point occurs around a density of 0.6, beyond which the performance advantage becomes significant when compared to the baseline algorithm. This highlights graph density as a key topological feature that critically influences the effectiveness of our approach, whereas in the number of nodes stays relatively constant.

Table 10: Performance vs Paper Summary by `DIST` Category (Filtered Extremes)

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
const	100	-2.80	-58.48	141.04	46.00	26.00	53.00	44.00
exp	99	-3.33	-82.74	204.79	41.41	24.24	56.57	43.43
uniform	96	-8.76	-81.60	173.65	41.67	17.71	58.33	48.96
highdelays	93	-171.21	-264.08	265.63	11.83	0.00	88.17	88.17

All percentages refer to filtered graphs. Positive ($\Delta > 0$), Significant Improvement ($\Delta > 50\%$), and Severe Degradation ($\Delta < -50\%$) rates shown per category.

As shown in Table 10, the most advantageous weight distribution is the `const` setting, as it substantially reduces graph delay. In contrast, the `highdelays` distribution yields

the worst performance, with a severe degradation observed in 88% of the graphs. These findings are consistent not only with our initial hypotheses outlined in Section 3.2.3, but also with the results of previous experiments.

Regarding Dijkstra, we observe very similar comparative results in Table 11, following the same logarithmic trend shown in the previous figures, which is why we omit them here. There remains a clear predominance of improvement in very dense and large graphs, whereas the most significant performance degradations are found in small graphs. Additionally, the `highdelays` distribution stands out as the only case under these estimations where performance becomes strongly negative.

Table 11: Performance vs Dijkstra Summary by `SIZE`, `DENSITY` and `DIST` Categories (Filtered Extremes)

(a) `SIZE` Category

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
small	100	96.47	46.71	142.65	85.00	85.00	15.00	15.00
medium	94	97.77	63.02	151.63	94.68	92.55	5.32	5.32
large	95	99.81	87.67	39.90	94.74	94.74	5.26	4.21
ultraLarge	100	99.94	89.93	38.36	95.00	95.00	5.00	5.00

(b) `DENSITY` Category

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
superSparse	72	96.85	68.98	117.08	91.67	91.67	8.33	8.33
sparse	77	96.94	57.03	168.82	90.91	90.91	9.09	7.79
medium	80	98.72	64.95	108.33	90.00	88.75	10.00	10.00
dense	80	99.42	77.54	69.24	93.75	92.50	6.25	6.25
superDense	80	99.62	89.49	31.99	95.00	95.00	5.00	5.00

(c) `DIST` Category

Category	n	Median (%)	Mean (%)	Std (%)	$\%\Delta > 0$	$\%\Delta > 50$	$\%\Delta < 0$	$\%\Delta < -50$
const	100	99.54	98.13	3.76	100.00	100.00	0.00	0.00
exp	100	99.34	97.07	7.70	100.00	100.00	0.00	0.00
uniform	100	99.07	96.18	8.44	100.00	100.00	0.00	0.00
highdelays	89	74.11	-13.73	205.52	66.29	64.04	33.71	32.58

If we aim to quantify the relative importance of different features, various techniques are available. One common approach is to train a gradient boosting regression model in order to identify which features contribute most significantly to the model's performance. This yields an initial overview of the relative weight of each graph-based dimension in explaining algorithmic performance.

Table 12: Importance of Features vs Paper Comparison through Gradient Boost Regression

Feature	**Importance**
`path_len`	0.292985
`n`	0.250284
`density`	0.247004
`mean_weights`	0.110020
`disp`	0.099707

In Table 12, we observe that one of the most influential, yet previously unmentioned, features is the shortest path length. This aligns with our earlier hypothesis that this characteristic can become a drawback, particularly in small graphs or in scenarios where a single long path exists despite otherwise favorable topological properties. In such cases, the complexity of the model, which grows linearly with the graph's height h as $\mathcal{O}(h|E|)$, becomes a limiting factor due to the need for the network to run for h iterations. This justifies why the shortest path length stands out as the most important feature, with a relative importance close to 0.3.

Following this, the number of nodes and edges appear as the next most influential features, which is to be expected given their inherent correlation. Finally, the edge weights and their dispersion rank lower in importance. This result is unsurprising, as high delays also negatively impact the performance of the baseline algorithm discussed in the reference paper, given that synaptic delays are also proportional to edge costs.

Table 13: Importance of Features vs Dijkstra Comparison through Gradient Boost Regression

Feature	**Importance**
`mean_w`	0.290436
`path_len`	0.204676
`n`	0.178812
`density`	0.163704
`disp`	0.162371

If we examine Dijkstra's algorithm, we observe a shift in feature importance. In this case, the mean edge weight ranks first in the importance table (see Table 13), which

is consistent with our expectations, as the delay is again the dominant factor in the algorithm's asymptotic behavior. The shortest path length follows closely, while the number of nodes and the graph's density appear lower in the ranking. These results are fully aligned with our initial hypotheses and reinforce the emerging picture provided by this feature-based characterization.

Table 14: Partial Correlations of Graph Features with Respect to Time Improvement

(a) vs Paper

Feature	**Correlation**
`n`	−0.070
`density`	0.516
`disp`	−0.171
`mean_w`	−0.163
`path_len`	−0.673

(b) vs Dijkstra

Feature	**Correlation**
`n`	0.758
`density`	0.522
`disp`	−0.232
`mean_w`	−0.310
`path_len`	−0.397

We now aim to assess whether each feature contributes positively or negatively to the algorithm's performance, as well as how much variability in performance these features account for. To this end, we refer to Table 14, which presents the partial correlations and the principal component decomposition of the feature space.

The partial correlations indicate the effect of each feature while holding all others constant. We observe that increasing the graph's density is associated with a consistent and significant improvement in performance, confirming earlier findings regarding the strong advantage of our algorithm over the baseline in densely connected graphs. On the other hand, shortest path length once again emerges as the strongest negative factor, showing the most substantial detrimental effect on performance. In addition, both the mean edge weight and its dispersion exhibit modest negative effects, suggesting that greater cost and heterogeneity among connections also limit the algorithm's gains, but could also improve it if the costs are low. The number of nodes, by contrast, appears to have only a minor influence, with a partial correlation of just –0.07, indicating that, when controlling for all other factors, increasing node count results in only a small performance penalty.

However, in comparison to Dijkstra's algorithm, although graph density remains a positively correlated factor (0.52), the number of nodes exhibits a strong partial correlation (0.758) with performance. This observation aligns with the fact that Dijkstra's asymptotic complexity depends primarily on the number of vertices, whereas our approach depends inversely on the number of edges, making the number of nodes a relative advantage in our

case. The most negatively correlated factors are, in fact, the average edge weight (−0.31) and the path length (−0.39); both directly influence our algorithm's linear growth in complexity.

Table 15: Principal Component Analysis (PCA) of Graph Features

Component	**Top Feature Loadings**
PC1 (72.9% var)	`mean_w` (0.76), `disp` (0.65), `path_len` (0.03)
PC2 (13.0% var)	`n` (0.92), `path_len` (0.38), `density` (0.08)
PC3 (9.4% var)	`path_len` (0.87), `n` (0.39), `density` (0.30)

PCA, in contrast, helps identify the main directions of variance in the feature space (Table 15). Here we observe that edge weight properties, namely, their average and variability, dominate the first principal component, suggesting that these factors are the primary sources of variation across graphs. The number of nodes and shortest path length appear as key contributors to the second and third components, respectively. This indicates that performance variability arises more from the cost structure of the network than from its specific topological shape.

5.2. Energy results

Similarly, by applying our previously developed energy estimation equations for both our algorithm (Equation (20)) and Dijkstra's algorithm (Equation (21)), we observe highly favorable results for our neuromorphic algorithm. It achieves superior energy efficiency in approximately 92.25% of the cases (369 graphs), while the remaining 7.75% (31 graphs) correspond to scenarios where Dijkstra's algorithm performs better. These edge cases occur with very small graphs coupled with high delays, causing the number of cycles in our algorithm to increase significantly relative to the few operations required by Dijkstra's algorithm.

Table 16: Comparison of estimated energy consumption (in microjoules) between our neuromorphic approach and the classical Dijkstra algorithm across 400 graphs.

Statistic	**Our Approach (µJ)**	**Dijkstra (µJ)**
Mean	1.71	390.24
Std. Dev.	5.03	1071.53
Min	0.00015	0.0223
25%	0.00583	0.704
Median	0.05841	5.44
75%	0.68259	116.14
Max	48.85	5494.58

As shown in Table 16, our average energy consumption in microjoules is lower by two orders of magnitude in most of the cases. On average, we obtain an energy efficiency advantage of approximately 228× (390.24 / 1.71), which aligns precisely with energy efficiency gains reported in the literature for neuromorphic chips across various tasks [29], such as in [39], where improvements of two to three orders of magnitude are documented.

By dividing the standard deviation by the mean to obtain the coefficient of variation, we observe that our algorithm (2.94) exhibits slightly higher relative variability compared to Dijkstra's algorithm (2.74). Nevertheless, these statistical results overall demonstrate a significant improvement in energy efficiency.

5.3. Weight compression results

We begin by analyzing how many shortest paths differ from those produced by Dijkstra's algorithm when using the three weight encoding methods introduced in Section 4.5: the original GCD-based method, the ranking (or mapping) approach, and logarithmic scaling. Based on our simulations, the original GCD encoding results in 0 out of 400 paths differing from Dijkstra's algorithm, the ranking method yields 47 out of 400 differing paths (11.75%), and the logarithmic method produces 177 out of 400 differing paths (44.25%).

Having established the deviation rates, we now examine the efficiency improvements introduced by each method in terms of execution time and energy consumption. Tables 12a and 12b summarize the results. All percentages for time and energy improvement are calculated with respect to the original GCD method and represent the sums across the entire set of execution times.

The average precision cost represents the mean of the pointwise differences between the costs of a given method and the original baseline. In contrast, the total precision loss is computed as the sum of all these differences divided by the sum of the original costs, providing a more global measure of the overall deviation.

(a) Ranking (Mapping) Variant

Metric	**Value**
Average time saved (ms)	0.0488
Average energy saved (μJ)	0.8980
Average precision loss (cost units)	26.24
Total time improvement (%)	76.33
Total energy improvement (%)	52.57
Total precision loss (%)	14.45
Time improvement-to-loss ratio	5.28
Energy improvement-to-loss ratio	3.64

(b) Logarithmic Scaling Variant

Metric	**Value**
Average time saved (ms)	0.0593
Average energy saved (μJ)	1.5335
Average precision loss (cost units)	83.66
Total time improvement (%)	92.75
Total energy improvement (%)	89.78
Total precision loss (%)	46.09
Time improvement-to-loss ratio	2.01
Energy improvement-to-loss ratio	1.95

Figure 12: Efficiency and precision trade-offs of weight encoding methods.

As shown in Tables 12a and 12b, the absolute gains in performance are notable, though not overwhelming in either approach. In relative terms, however, the precision loss becomes significantly more pronounced: paths computed using the ranking method are on average 26.24 units less costly than those produced by the original algorithm, while logarithmic scaling leads to an average decrease of 83.66 cost units.

On a global scale, the ranking and logarithmic methods achieve overall execution time improvements of 76.33% and 92.75%, respectively. Although this represents a considerable temporal gain of 16.42 percentage points in favor of logarithmic scaling, the trade-off does not completely justify its use. Both the total and average precision loss for the logarithmic method—expressed as a percentage and in units of edge cost, respectively—are approximately four times higher than those observed with the ranking method.

This suggests that logarithmic scaling is a more aggressive transformation that significantly distorts the resulting paths. Despite its flattening effect on weights, it fails to yield proportional time benefits. A plausible explanation is that, in graphs where shortest paths are already relatively short, logarithmic scaling disproportionately reduces the weights of edges that are unlikely to be part of the optimal path. Consequently, this leads to a loss in

resolution for genuinely short paths, amplifying imprecision without commensurate gains in runtime.

However, when we shift our focus to energy consumption, the improvements associated with logarithmic scaling are more substantial, nearly twice those achieved with ranking. Nevertheless, even in this domain, the magnitude of precision loss outweighs the gains. This conclusion is further supported by the efficiency ratios, which consistently indicate that the ranking (or mapping) method offers a more favorable balance between performance improvement and precision degradation, with an approximate 2× advantage.

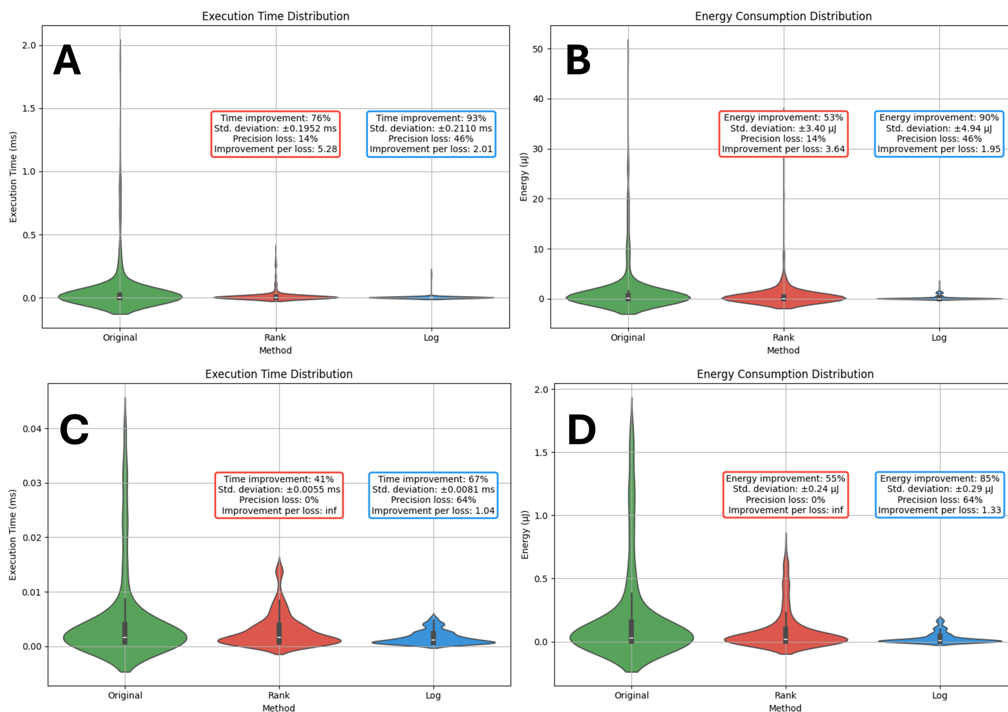


Figure 13: The top panels (A & B) show the distributions of execution time and energy consumption, respectively, including outliers, i.e., the most improbable samples within the distributions. The lower panels (C & D) display the same distributions filtered by interquartile ranges to remove these outliers. Each image highlights the computed mean improvements and their standard deviations for each method, framed in colored boxes: red for the ranking method and blue for the logarithmic scaling method.

Focusing now on relative changes and the distributions of execution times and energy consumption, Figure 13 provides a clear visualization. Panels A and B precisely reflect the previously reported statistics, clearly illustrating the flattening effect produced by the proposed compression methods. Both execution time and energy consumption decrease and compress, resulting in a lower absolute value and a broader distribution.

What is truly noteworthy is that, after filtering outliers in panels C and D, although the

percentage improvement in time diminishes considerably, the standard deviations also decrease substantially, which is expected. Importantly, for the ranking method, there is no loss of precision, meaning that within the most common cases inside the interquartile ranges, this method achieves an average 41% time improvement without any path errors, alongside an average 55% reduction in energy consumption.

Conversely, the logarithmic scaling method shows reduced gains and increased precision loss, indicating that this more aggressive approach performs better in distributions with atypical weights and highly dispersed large values. This advantage primarily arises in the presence of outliers; when these edges are removed, the total precision declines.

6. Conclusions and Future Work

We have reviewed key literature in the field of neuromorphic computing to build a comprehensive initial intuition about the nature of these bioinspired architectures. We discussed the functioning of neurons and synapses, as well as their applications, including frameworks that enable us to understand and model them. The implementation of these novel computational primitives to form Turing-complete systems and generate new computational models is a cornerstone of this work, as it enables the wide range of use cases we have reviewed, from deep learning to our focus on neuromorphic algorithms for solving graph problems.

We have also developed a novel algorithm in depth, aligned with current trends in the literature, and expanded upon existing work. Not only have we implemented software simulations of our algorithm and proposed conceptual solutions to challenges inherent in neuromorphic graph processing, such as delay compression techniques, but we have also extended the work of [53]. We have studied both approaches and compared them with Dijkstra's algorithm to justify and explore the paradigm shift from classical to neuromorphic computation.

And we have ultimately addressed one of the main challenges in encoding graphs as SNNs: the use of synaptic delays proportional to edge weights. Through both theoretical analysis and empirical experimentation, we have shown how this approach negatively impacts algorithmic performance, introducing additional real complexity. To mitigate this limitation, we proposed several delay compression techniques and evaluated their respective advantages and limitations, providing an initial framework for practical optimization that resulted on the evaluation of two main methods, the mapping and the logarithmic scale.

These studies have yielded highly relevant insights into the characterization of our algorithm, bridging the theoretical and practical aspects of neuromorphic computing in a more

detailed manner. We have functionally defined the key components of our algorithm, the reference paper, and Dijkstra's algorithm, and we have analyzed their algorithmic complexities, ultimately establishing a temporal complexity of $\mathcal{O}(h|E|)$ for our solution. The simulations, estimates, and experiments we conducted provided empirical evidence that helped validate our theoretical developments and facilitated practical comparisons.

As a result, we obtained a more refined characterization of our algorithm, which proved to be consistently faster on large and dense graphs, showing improvements of up to 66.25% in SuperDense graphs and 36.36% in Large graphs. However, it exhibited clear disadvantages compared to both the reference algorithm in [53] and Dijkstra's algorithm on small, sparse graphs or those with very high delays.

The distributions of performance gains and losses reveal how the algorithm behaves across different graph categories. Our analysis using partial correlations and PCA identified the most relevant features. Specifically, the path length h, which directly affects the asymptotic bound of our algorithm, emerges as a major limiting factor. Additionally, the average edge weight and its dispersion also negatively affect performance, albeit to a more moderate extent. In contrast, graph density is the most beneficial feature for our approach, consistently improving its performance.

Regarding energy consumption, we observed a substantial improvement over classical approaches, achieving a gain of 2 to 3 orders of magnitude, with an average improvement in energy efficiency of approximately 228×. Not only does our method result in significantly lower mean energy consumption, but it also exhibits a smaller standard deviation. However, proportionally, its variability is slightly higher than that of the classical approach (2.74 vs. 2.94). This is attributed to our algorithm's strong dependence on graph topology, including delays and the length of the shortest path. Even in dense graphs, high neuronal activity per cycle or delays that cause synapses to retain spikes for multiple ticks can increase the energy cost, making the algorithm slightly more variable in energy performance under such conditions.

Nonetheless, a key limitation must be acknowledged: while our estimations aim to be as comprehensive as possible, they remain approximations. The actual results may vary depending on the specific neuromorphic chip used or the experimental conditions under which the algorithm is executed. For this reason, particularly in the temporal analysis, these estimates should be understood primarily as a tool for algorithmic characterization, allowing us to analyze its behavior across different types of graphs, rather than as precise predictions of computation time.

Future work could further investigate our delay compression techniques to reduce the

practical time complexity of neuromorphic graph algorithms and validate these improvements through comprehensive experimental evaluation. In addition, reducing the gap between theoretical and functional implementations would be highly valuable. This includes directly implementing these algorithms on neuromorphic hardware or FPGAs to unequivocally identify the common mechanisms enabling this type of computation and to perform more precise comparative analyses.

Lastly, exploring alternative methodologies for designing graph algorithms presents a promising direction. Although the current one-to-one mapping between graphs and SNNs is intuitive and straightforward, more efficient representations or embeddings may exist that offer enhanced computational efficiency and energy performance.

References

[1] Peter Ashwin, Stephen Coombes, and Rachel Nicks. Mathematical frameworks for oscillatory network dynamics in neuroscience. *The Journal of Mathematical Neuroscience*, 6:1–92, 2016.

[2] Samuel López Asunción. *FPGA-Based Acceleration for Emerging Neuromorphic Computing Paradigms*. Doctoral thesis, Universidad Politécnica de Madrid, Escuela Técnica Superior de Ingenieros de Telecomunicación, Madrid, Spain, 2023.

[3] Michael Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, 47(2):263, 2002.

[4] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European journal of operational research*, 119(3):557–581, 1999.

[5] James J Buckley and Yoichi Hayashi. Fuzzy neural networks: A survey. *Fuzzy sets and systems*, 66(1):1–13, 1994.

[6] Martin V Butz. Toward a unified sub-symbolic computational theory of cognition. *Frontiers in psychology*, 7:925, 2016.

[7] György Buzsáki. *Rhythms of the Brain*. Oxford university press, 2006.

[8] Gyorgy Buzsaki and Andreas Draguhn. Neuronal oscillations in cortical networks. *science*, 304(5679):1926–1929, 2004.

[9] Jérémie Cabessa. Turing complete neural computation based on synaptic plasticity. *PloS one*, 14(10):e0223451, 2019.

[10] Louis-Charles Caron, Michiel D'Haene, Frédéric Mailhot, Benjamin Schrauwen, and Jean Rouat. Event management for large scale event-driven digital hardware spiking neural networks. *Neural networks*, 45:83–93, 2013.

[11] Joni Dambre, David Verstraeten, Benjamin Schrauwen, and Serge Massar. Information processing capacity of dynamical systems. *Scientific reports*, 2(1):514, 2012.

[12] Anup Das. Neuromorphic computing for graph analytics. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pages 1–7, 2024.

[13] Prasanna Date, Bill Kay, Catherine Schuman, Robert Patton, and Thomas Potok. Computational complexity of neuromorphic algorithms. In *International Conference on Neuromorphic Systems 2021*, pages 1–7, 2021.

[14] Prasanna Date, Thomas Potok, Catherine Schuman, and Bill Kay. Neuromorphic computing is turing-complete. In *Proceedings of the International Conference on Neuromorphic Systems 2022*, pages 1–10, 2022.

[15] Suman Datta, Nikhil Shukla, Matthew Cotter, Abhinav Parihar, and Arijit Raychowdhury. Neuro inspired computing with coupled relaxation oscillators. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[16] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, et al. Loihi: A neuromorphic manycore processor with on-chip learning. *Ieee Micro*, 38(1):82–99, 2018.

[17] Weiping Ding, Mohamed Abdel-Basset, Hossam Hawash, and Ahmed M Ali. Explainability of artificial intelligence methods, applications and challenges: A comprehensive survey. *Information Sciences*, 615:238–292, 2022.

[18] Yan Fang, Zheng Wang, Jorge Gomez, Suman Datta, Asif I Khan, and Arijit Raychowdhury. A swarm optimization solver based on ferroelectric spiking neural networks. *Frontiers in neuroscience*, 13:855, 2019.

[19] Gabriel A Fonseca Guerra and Steve B Furber. Using stochastic spiking neural networks on spinnaker to solve constraint satisfaction problems. *Frontiers in neuroscience*, 11:714, 2017.

[20] Pascal Fries. Rhythms for cognition: communication through coherence. *Neuron*, 88(1):220–235, 2015.

[21] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[22] Matthieu Gilson, Anthony Burkitt, and J Leo van Hemmen. Stdp in recurrent neuronal networks. *Frontiers in computational neuroscience*, 4:23, 2010.

[23] Cornelius Glackin, Liam McDaid, Liam Maguire, and Heather Sayers. Implementing fuzzy reasoning on a spiking neural network. In *Artificial Neural Networks-ICANN 2008: 18th International Conference, Prague, Czech Republic, September 3-6, 2008, Proceedings, Part II 18*, pages 258–267. Springer, 2008.

[24] Kurt Gödel. On undecidable propositions of formal mathematics systems. 1934.

[25] Andrew V Goldberg and Robert E Tarjan. Expected performance of dijkstra's shortest path algorithm. *NEC Research Institute Report*, 1996.

[26] Hayato Goto. Universal quantum computation with a nonlinear oscillator network. *Physical Review A*, 93(5):050301, 2016.

[27] Henry Han and Xiangrong Liu. The challenges of explainable ai in biomedical data science. *BMC bioinformatics*, 22(Suppl 12):443, 2022.

[28] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14. IEEE, February 2014.

[29] Murat Isik, Karn Tiwari, Muhammed Burak Eryilmaz, and I Can Dikmen. Accelerating sensor fusion in neuromorphic computing: A case study on loihi-2. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2024.

[30] Eugene M Izhikevich. Computing with oscillators. *Neural Networks*, 5255:1–30, 2000.

[31] Adeel Javaid. Understanding dijkstra's algorithm. *Available at SSRN 2340905*, 2013.

[32] Zeno Jonke, Stefan Habenschuss, and Wolfgang Maass. Solving constraint satisfaction problems with networks of spiking neurons. *Frontiers in neuroscience*, 10:118, 2016.

[33] Bill Kay, Prasanna Date, and Catherine Schuman. Neuromorphic graph algorithms: Extracting longest shortest paths and minimum spanning trees. In *Proceedings of the 2020 Annual Neuro-inspired Computational Elements Workshop*, pages 1–6, 2020.

[34] Samuel Lanthaler, T Konstantin Rusch, and Siddhartha Mishra. Neural oscillators are universal. *Advances in Neural Information Processing Systems*, 36:46786–46806, 2023.

[35] Fang Liu, Jie Yang, Witold Pedrycz, and Wei Wu. A new fuzzy spiking neural network based on neuronal contribution degree. *IEEE Transactions on Fuzzy Systems*, 30(7):2665–2677, 2021.

[36] Wolfgang Maass. To spike or not to spike: that is the question. *Proceedings of the IEEE*, 103(12):2219–2224, 2015.

[37] Danijela Marković, Alice Mizrahi, Damien Querlioz, and Julie Grollier. Physics for neuromorphic computing. *Nature Reviews Physics*, 2(9):499–510, 2020.

[38] Natasza Marrouch, Joanna Slawinska, Dimitrios Giannakis, and Heather L Read. Data-driven koopman operator approach for computational neuroscience. *Annals of Mathematics and Artificial Intelligence*, 88(11):1155–1173, 2020.

[39] Daniel Marti, Matteo Rigotti, Minyoung Seok, and Stefano Fusi. Energy-efficient neuromorphic classifiers. *Neural Computation*, 28(10):2011–2044, 2016.

[40] Paul A Merolla, John V Arthur, Bertram E Shi, and Kwabena A Boahen. Expandable networks for neuromorphic chips. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 54(2):301–311, 2007.

[41] Justin Morris, Hin Wai Lui, Kenneth Stewart, Behnam Khaleghi, Anthony Thomas, Thiago Marback, Baris Aksanli, Emre Neftci, and Tajana Rosing. Hyperspike: hyperdimensional computing for more efficient and robust spiking neural networks. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 664–669. IEEE, 2022.

[42] Hamid Nasiri and Mohammad Mehdi Ebadzadeh. Mfrfnn: Multi-functional recurrent fuzzy neural network for chaotic time series prediction. *Neurocomputing*, 507:292–310, 2022.

[43] Emre O Neftci, Charles Augustine, Somnath Paul, and Georgios Detorakis. Event-driven random back-propagation: Enabling neuromorphic deep learning machines. *Frontiers in neuroscience*, 11:324, 2017.

[44] NEST Initiative. Nest simulator. `https://www.nest-simulator.org/`, 2024. Accessed: 2025-04-21.

[45] Anh Nguyen, Jason Yosinski, and Jeff Clune. Understanding neural networks via feature visualization: A survey. *Explainable AI: interpreting, explaining and visualizing deep learning*, pages 55–76, 2019.

[46] Christoph Ostrau, Christian Klarhorst, Michael Thies, and Ulrich Rückert. Comparing neuromorphic systems by solving sudoku problems. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 521–527. IEEE, 2019.

[47] Christoph Ostrau, Christian Klarhorst, Michael Thies, and Ulrich Rückert. Benchmarking neuromorphic hardware and its energy expenditure. *Frontiers in neuroscience*, 16:873935, 2022.

[48] Priyadarshini Panda, Jason M Allred, Shriram Ramanathan, and Kaushik Roy. Asp: Learning to forget with adaptive synaptic plasticity in spiking neural networks. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 8(1):51–64, 2017.

[49] Riccardo Pignari, Vittorio Fra, Enrico Macii, and Gianvito Urgese. Efficient solution validation of constraint satisfaction problems on neuromorphic hardware: the case of sudoku puzzles. *IEEE Transactions on Artificial Intelligence*, 2025.

[50] Nitin Rathi, Indranil Chakraborty, Adarsh Kosta, Abhronil Sengupta, Aayush Ankit, Priyadarshini Panda, and Kaushik Roy. Exploring neuromorphic computing based on spiking neural networks: Algorithms to hardware. *ACM Computing Surveys*, 55(12):1–49, 2023.

[51] Francisco A Rodrigues, Thomas K DM Peron, Peng Ji, and Jürgen Kurths. The kuramoto model in complex networks. *Physics Reports*, 610:1–98, 2016.

[52] Kaushik Roy, Akhilesh Jaiswal, and Priyadarshini Panda. Towards spike-based machine intelligence with neuromorphic computing. *Nature*, 575(7784):607–617, 2019.

[53] Catherine D Schuman, Kathleen Hamilton, Tiffany Mintz, Md Musabbir Adnan, Bon Woong Ku, Sung-Kyu Lim, and Garrett S Rose. Shortest path and neighborhood subgraph extraction on a spiking memristive neuromorphic implementation. In *Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop*, pages 1–6, 2019.

[54] Catherine D Schuman, Thomas E Potok, Robert M Patton, J Douglas Birdwell, Mark E Dean, Garrett S Rose, and James S Plank. A survey of neuromorphic computing and neural networks in hardware. *arXiv preprint arXiv:1705.06963*, 2017.

[55] Teresa Serrano-Gotarredona, Timothée Masquelier, Themistoklis Prodromakis, Giacomo Indiveri, and Bernabe Linares-Barranco. Stdp and stdp variations with memristors for spiking neuromorphic learning systems. *Frontiers in neuroscience*, 7:2, 2013.

[56] Abhishek A Sharma, James A Bain, and Jeffrey A Weldon. Phase coupling and control of oxide-based oscillators for neuromorphic computing. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 1:58–66, 2015.

[57] Gopalakrishnan Srinivasan, Priyadarshini Panda, and Kaushik Roy. Stdp-based unsupervised feature learning using convolution-over-time in spiking neural networks for energy-efficient neuromorphic computing. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 14(4):1–12, 2018.

[58] Klaus M Stiefel and G Bard Ermentrout. Neurons as oscillators. *Journal of neurophysiology*, 116(6):2950–2960, 2016.

[59] Gouhei Tanaka, Toshiyuki Yamane, Jean Benoit Héroux, Ryosho Nakane, Naoki Kanazawa, Seiji Takeda, Hidetoshi Numata, Daiju Nakano, and Akira Hirose. Recent advances in physical reservoir computing: A review. *Neural Networks*, 115:100–123, 2019.

[60] Jacob Torrejon, Mathieu Riou, Flavio Abreu Araujo, Sumito Tsunegi, Guru Khalsa, Damien Querlioz, Paolo Bortolotti, Vincent Cros, Kay Yakushiji, Akio Fukushima, et al. Neuromorphic computing with nanoscale spintronic oscillators. *Nature*, 547(7664):428–431, 2017.

[61] Andrei Velichko, Maksim Belyaev, and Petr Boriskov. A model of an oscillatory neural network with multilevel neurons for pattern recognition and computing. *Electronics*, 8(1):75, 2019.

[62] Anne Wilson and James Hendler. Linking symbolic and subsymbolic computing. *Connection Science*, 5(3-4):395–414, 1993.

[63] Chris Yakopcic, Nayim Rahman, Tanvir Atahary, Tarek M Taha, and Scott Douglass. Solving constraint satisfaction problems using the loihi spiking neuromorphic processor. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1079–1084. IEEE, 2020.

[64] Zhuowen Zou, Haleh Alimohamadi, Farhad Imani, Yeseong Kim, and Mohsen Imani. Spiking hyperdimensional network: Neuromorphic models integrated with memory-inspired framework. *arXiv preprint arXiv:2110.00214*, 2021.

[65] Zhuowen Zou, Haleh Alimohamadi, Yeseong Kim, M Hassan Najafi, Narayan Srinivasa, and Mohsen Imani. Eventhd: Robust and efficient hyperdimensional learning with neuromorphic sensor. *Frontiers in Neuroscience*, 16:858329, 2022.

[66] Álvaro Sánchez-Paniagua Ríos. Neuromorphic computing tfm – source code, 2025. Available in: `https://github.com/Kokechacho/TFM`.

A. Graphical Notation for Spiking Neural Networks

For the graphical representation of the spiking neural network used in our experiments, we adopt a consistent and readable notation aligned with the conventions introduced in [14].

A.1. Representation and neuron model

Neurons are depicted as circles containing their key parameters, enclosed in curly braces: the *firing threshold* v_{th}, which specifies the minimum membrane potential required for the neuron to emit a spike, and the *leak constant* λ, which governs the rate at which the membrane potential decays back to its resting value (0). In our model, a value of $\lambda = 0$ denotes instantaneous information loss (no memory retention), whereas $\lambda = \infty$ represents perfect memory retention until the neuron fires. A unique numerical identifier is placed below each neuron to distinguish it within the network.

Synapses are represented as directed arrows connecting neurons. Below each arrow, we indicate the ordered pair (i, j) denoting the connection from presynaptic neuron i to postsynaptic neuron j. Above each arrow, we annotate the synaptic parameters enclosed within angle brackets $\langle \cdot \rangle$: the *synaptic weight* ω, which determines the strength of the transmitted spike, and the *delay* δ, which specifies the number of discrete time units required for the spike to reach the target neuron.

Parameters may include subscripts corresponding to their associated neuron or synapse. For example, the delay of the synapse from neuron 2 to neuron 3 is denoted by $\delta_{(2,3)}$. The neurons in our simulations are modeled using the classic *Leaky Integrate-and-Fire* (LIF) model, which is widely used in computational neuroscience for modeling biological neurons.

B. First Experiment: Logic Gates

To begin exploring the neuromorphic paradigm, we implemented a basic set of logic gates: NOT, AND, OR, and implication. This initial approach is motivated by the desire to investigate the computational capabilities of spiking neural networks (SNNs) from a practical perspective, by verifying whether arbitrary Boolean functions can be computed. Recall that the set {AND, OR, NOT} forms a functionally complete basis.

This first step is essential for understanding how logical reasoning can be modeled in such systems. In fact, existing approaches such as neuro-SAT solvers demonstrate how logical statements can be addressed using neural networks. However, it is worth noting that our approach here deviates, both in spirit and in structure, from the core principles of

neuromorphic computing and the ultimate goals of this appendix. Our long-term aim is to develop networks that are more parallelized, distributed, and noise-tolerant, in contrast to the inherently sequential and deterministic nature of conventional logic gates. Thus, this appendix should be viewed as a conceptual approximation to neuromorphic computing, intended to build intuition for the design of more sophisticated algorithms in later stages.

The final section of this appendix will address the last problem discussed herein, which served as our entry point into constraint-based programming within this paradigm, leveraging inhibitory synaptic connections. Specifically, we focus on the Hamiltonian cycle problem and the Travelling Salesman Problem (TSP), aiming to replicate methodologies that will be described in greater detail in subsequent sections.

B.1. AND Gate (∧)

The AND gate is implemented in a straightforward manner, inspired by [14], particularly in relation to μ-recursive functions such as the successor or constant function.

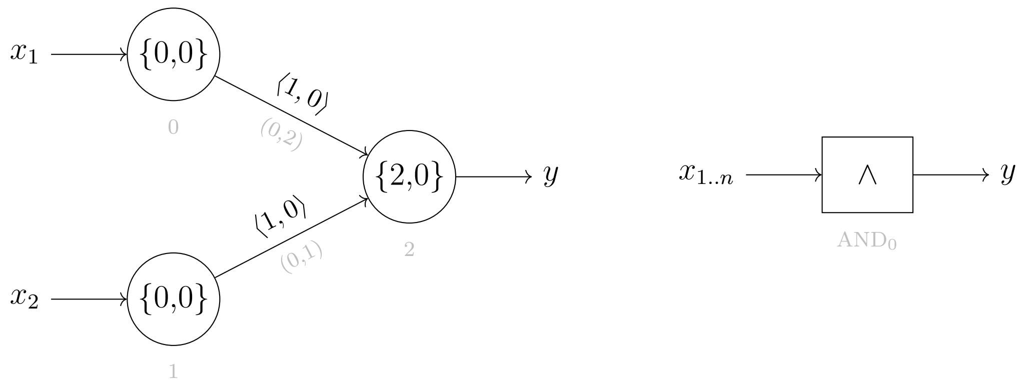


Figure 14: AND gate and its abstract representation.

The behavior is as follows: logical inputs $x_1, x_2 \in \{0, 1\}$ stimulate neurons 0 and 1, respectively, causing them to emit spikes. These spikes are transmitted via synapses of weight 1 to neuron 2. Neuron 2 will only fire if it receives both spikes simultaneously, since its activation threshold is set to 2. In that case, it emits a spike representing the output y.

B.2. OR Gate (∨)

The OR gate is built analogously to the AND gate:

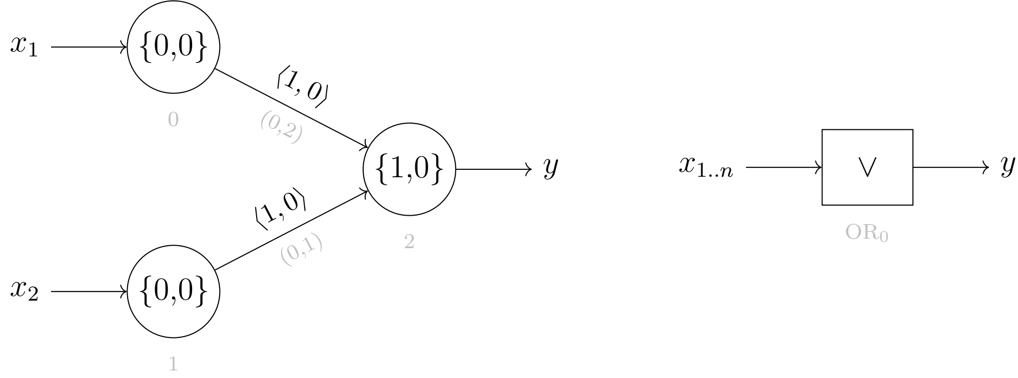


Figure 15: OR gate and its abstract representation.

The only difference from the AND gate is that neuron 2 has a reduced threshold of 1. Thus, the output neuron will fire if at least one of the inputs emits a spike, which is consistent with the semantics of logical disjunction.

B.3. NOT Gate (¬)

This gate requires a slightly more creative approach, as it introduces two essential mechanisms characteristic of this computational model: inhibitory synapses and synaptic delay δ.

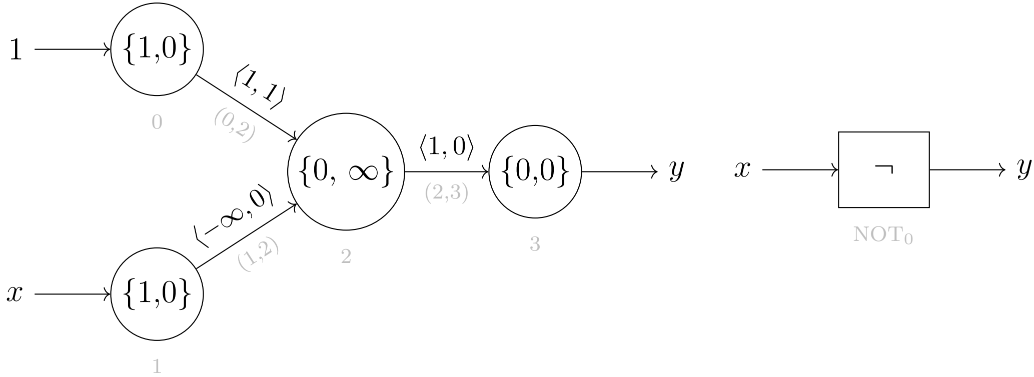


Figure 16: NOT gate: spiking neural circuit (left) and abstract representation (right).

This configuration uses two inputs: a constant 1 sent to neuron 0 (which always fires), and the logical variable $x \in \{0, 1\}$ sent to neuron 1.

- **Case** $x = 0$**.** Neuron 1 does not reach its threshold ($v_{th_1} = 1 > x$) and remains silent. Neuron 0 fires and transmits an excitatory spike to neuron 2, pushing it above its threshold. Neuron 2 fires, stimulating neuron 3, which outputs $y = 1$.
- **Case** $x = 1$**.** Both neuron 0 and neuron 1 fire, but the synaptic delays differ: $\delta_{(1,2)} < \delta_{(0,2)}$. Therefore:

1. At $t = 0$, the spike from neuron 1 arrives first. Its inhibitory weight $\omega_{(1,2)} = -\infty$ sets neuron 2's membrane potential to $-\infty$, which persists due to its leak $\lambda_2 = \infty$.
2. At $t = 1$, the excitatory spike from neuron 0 arrives with weight $\omega_{(0,2)} = +1$, but:

   $$(-\infty) + 1 = -\infty,$$

   so neuron 2 remains below threshold and does not spike.

As neuron 2 does not spike, neuron 3 is not activated, and the output is $y = 0$.

This could alternatively be implemented with $\delta_{(0,2)} = \delta_{(1,2)} = 0$, where both spikes arrive simultaneously. The inhibitory weight would dominate the sum, suppressing the excitation. However, introducing a delay makes the dynamics more intuitive and the mechanism easier to understand.

Thus, this configuration correctly implements a NOT gate, inverting the logical value of the input.

B.4. Implication Gate ($\to$)

Finally, we implement the implication gate. This is relatively straightforward if we recall that:

$$A \to B \equiv \neg A \vee B$$

Hence, the corresponding circuit can be built as a direct composition of a NOT gate followed by an OR gate. This logical equivalence allows us to reuse previously defined modules to implement implication naturally within this computational framework:

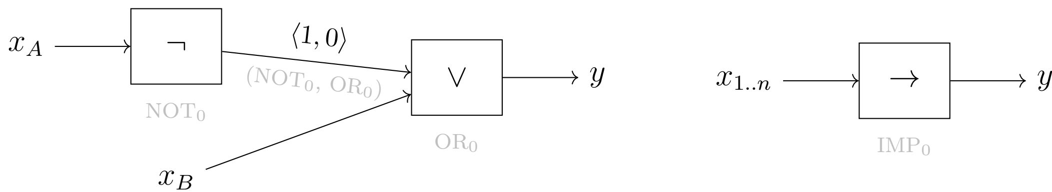


Figure 17: Implication gate: spiking circuit (left) and abstract symbol (right).

This completes the implementation of four essential logical gates, establishing a functional basis $\{\wedge, \vee, \neg\}$. These primitives suffice to express any Boolean function, highlighting the expressiveness and completeness of spiking neural circuits as a computational model.

C. Modeling

Once the dynamics of the neuromorphic paradigm are understood, we can address the final objective of this appendix: the search for a Hamiltonian cycle in a graph and, more ambitiously, the resolution of the Travelling Salesman Problem (TSP), a more complex and computationally demanding generalization.

The following simulations are carried out using the `NEST` framework, a Python-based simulator specifically designed for modeling spiking neural networks. `NEST` is primarily targeted at computational neuroscience and offers fine-grained control over both neuronal and synaptic parameters, including their temporal dynamics.

We chose `NEST` due to its flexibility and expressive power, which allowed us to accurately define our network components. Furthermore, [44] provides a comprehensive Jupyter notebook with several practical examples, including a neuromorphic Sudoku solver. This example served as both a reference and inspiration for the design of our system.

Given a directed graph $G = (V, E)$, the *Hamiltonian cycle problem* consists of finding a tour that visits each vertex exactly once and returns to the starting vertex. In contrast, the *Travelling Salesman Problem* (TSP) considers a weighted directed graph $G = (V, E, w)$, requiring not only a Hamiltonian cycle but one that minimizes the total cost, defined as the sum of the weights of the selected edges.

To approach both problems, we build upon two foundational works [44, 49], along with other studies that have explored the Hamiltonian cycle and TSP using various neuromorphic approaches [18, 36]. Both foundational works address the solution of Sudoku instances via spiking neural networks, a combinatorial problem that shares many structural similarities with those we consider here. Their methodologies and results serve as a basis for the design and adaptation of our own neuromorphic architectures tailored to Hamiltonian cycle detection and TSP optimization.

In general, the main challenges when applying this neuromorphic paradigm are twofold:

1. **Defining the computational environment.** Given the fragmented nature of the neuromorphic framework and the large number of available neuronal parameters, a crucial first step is selecting the components to include (e.g., membrane leak, refractory periods, STDP, inhibitory synapses, etc.). We must analyze how these components influence network behavior while ensuring the model retains sufficient expressiveness and formal properties to allow for complexity-theoretic analysis.

2. **Parameter tuning.** Once a basic architecture is selected, appropriate values must be assigned to all parameters to ensure the desired network behavior. As we will

show in the final implementation, this task can be formidable: ensuring properties like termination or optimality demands precise tuning of delays, synaptic weights, thresholds, leaks, and more. The search space grows rapidly, greatly complicating formal validation.

Let $G = (V, E)$ be a directed graph with $V = \{v_1, v_2, \ldots, v_n\}$ and $E \subseteq V \times V$. A *Hamiltonian cycle solution* in G is a sequence:

$$S = (s_1, s_2, \ldots, s_n)$$

satisfying:

1. S is a permutation of the vertices:

$$\{s_1, s_2, \ldots, s_n\} = V \quad \text{and} \quad s_i \neq s_j \quad \forall i \neq j.$$

2. Each consecutive pair is connected in G:

$$(s_i, s_{i+1}) \in E \quad \forall\, i = 1, \ldots, n-1.$$

3. The sequence is cyclic, i.e., the last vertex connects back to the first:

$$(s_n, s_1) \in E.$$

To model each vertex $v_i \in V$ within the neuromorphic circuit, we associate it with a population of five fully interconnected neurons, denoted $\Psi_{i,j}$. Each population $\Psi_{i,j}$ encodes the assignment of vertex v_i to position j in the cycle (i.e., $\Psi_{i,j} \equiv v_i = s_j$). Since a solution S consists of n positions, we replicate these populations to form an $n \times n$ matrix of neural populations:

$$M = \begin{bmatrix} \Psi_{1,1} & \Psi_{1,2} & \ldots & \Psi_{1,n} \\ \Psi_{2,1} & \Psi_{2,2} & \ldots & \Psi_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \Psi_{n,1} & \Psi_{n,2} & \ldots & \Psi_{n,n} \end{bmatrix}.$$

Here, row i corresponds to vertex v_i, and column j corresponds to the j-th position in the tour. Thus, each pair (i, j) uniquely identifies a population modeling the assignment of vertex v_i to position j.

C.1. Hamiltonian Cycle Algorithm

Initially, all populations $\Psi_{i,j}$ are stimulated using random noise drawn from a Poisson generator θ. This promotes exploration of the solution space through stochastic activations.

We now enforce the constraints described earlier by connecting the populations via inhibitory synapses. These connections are carefully designed with fixed parameters to guide the network's dynamics toward valid solutions, without eliminating the stochastic nature of the system. All inhibitory synapses are assigned a fixed weight $\omega_{\text{inhib}} = -0.2$.

To enforce constraint (1), no vertex appears more than once, we define two types of symmetric inhibitory connections:

- **Row-wise inhibition (duplicate vertex suppression):**

$$\forall i \in \{1, \ldots, n\},\ \forall j \neq j', \quad \omega_{(\Psi_{i,j}, \Psi_{i,j'})} = \omega_{\text{inhib}}.$$

  This prevents vertex v_i from being assigned to multiple positions.

- **Column-wise inhibition (unique position enforcement):**

$$\forall j \in \{1, \ldots, n\},\ \forall i \neq i', \quad \omega_{(\Psi_{i,j}, \Psi_{i',j})} = \omega_{\text{inhib}}.$$

  This ensures only one vertex is assigned to each position.

To enforce constraints (2) and (3), requiring connectivity between consecutive vertices and cycle closure, we add the following inhibitory connections:

- **Inhibition between consecutive positions:**

$$\forall i \neq i',\ \forall j \in \{1, \ldots, n-1\}, \quad (v_i, v_{i'}) \notin E \Rightarrow \omega_{(\Psi_{i,j}, \Psi_{i',j+1})} = \omega_{\text{inhib}}.$$

- **Inhibition for cycle closure (last to first position):**

$$\forall i \neq i', \quad (v_i, v_{i'}) \notin E \Rightarrow \omega_{(\Psi_{i,n}, \Psi_{i',1})} = \omega_{\text{inhib}}.$$

With these inhibitory patterns in place, the neuromorphic network is steered toward valid Hamiltonian cycles. The noise θ stochastically stimulates each $\Psi_{i,j}$, and once a population fires, it inhibits its competitors via ω_{inhib}. The network is simulated for a fixed number of steps α (or until a maximum time threshold), allowing the system to converge toward a valid activation pattern.

To extract a candidate solution S, we monitor the spike count of each population throughout the simulation and, for each position j, select the population $\Psi_{i,j}$ with the highest spike count. The set of most active populations defines:

$$S = \Big\{\, \Psi_{i_1,1},\ \Psi_{i_2,2},\ \ldots,\ \Psi_{i_n,n} \Big\}.$$

We then verify, in polynomial time, whether S constitutes a valid Hamiltonian cycle:

1. All coordinates (i_k, j) must be unique, ensuring a single active population per row and column.
2. For every pair $(\Psi_{i_j,j}, \Psi_{i_{j+1},j+1}) \in S$, the corresponding edge $(v_{i_j}, v_{i_{j+1}}) \in E$ must exist.
3. To complete the cycle, $(v_{i_n}, v_{i_1}) \in E$ must hold.

If all conditions are satisfied, we accept S as a valid solution; otherwise, we discard it and repeat the process.

The algorithm allows for a parameterizable selection of the source or destination node, as it is sufficient to directly stimulate, by applying a high potential, the population $\Psi_{i,j}$ that one wishes to fix at a specific position. This ability to enforce an initial choice can significantly improve the efficiency of the algorithm, since restricting a particular vertex-position combination reduces the search space by approximately a factor of n. In scenarios where both the starting and ending nodes are fixed, the reduction can reach a factor close to n^2, potentially accelerating convergence to a valid solution.

This algorithm has proven capable of finding valid Hamiltonian cycles in all tested graphs, ranging from those with $|V| = 5$ to graphs with $|V| = 20$. While we are confident that the same strategy will scale to graphs of arbitrary size, it should be noted that there is currently no formal proof guaranteeing convergence in all cases. In practice, the search relies on a stochastic process that, by pure probability, eventually identifies a Hamiltonian cycle.

The method draws inspiration from neuromorphic networks applied to Sudoku solving: when only one valid combination exists, inhibitory dynamics quickly guide the system toward the optimal solution. In such a scenario, the populations corresponding to the correct path reinforce their activation while suppressing the others, until reaching a stable state in which no other populations compete for activation.

In cases where multiple Hamiltonian cycles exist, the network may take longer to settle, as several populations simultaneously compete to be activated at different positions in the

solution. However, the configuration of the inhibitory synapses favors the persistence of populations forming part of a consistent solution, as they are not significantly inhibited by others. Across multiple runs, this dynamic of stochastic competition and suppression eventually promotes the emergence of a valid solution.

We have tested the algorithm on graphs with both a unique solution and multiple solutions, and in all cases it successfully finds a Hamiltonian cycle before reaching the iteration limit. A particularly intriguing observation is that the learning process of the proposed network follows a sigmoidal trend, aligning with the biological behavior of learning.

C.2. Algorithmic Complexity

Following the algorithmic analysis proposed in [13], we can establish that the spatial complexity of the model with respect to the number of neurons is $\mathcal{O}(n^2)$, since we employ a matrix M of size $n \times n$, where each cell represents a population composed of 5 neurons. Therefore, the total number of neurons is proportional to $5n^2$, which implies a spatial complexity of $\mathcal{O}(n^2)$.

As for the number of synapses, we can decompose it into two main contributions. The first, corresponding to constraint (1), involves inhibitory connections between populations that share the same row or column. Since each population in the matrix M connects to the remaining $n-1$ populations in its row and $n-1$ in its column, we obtain a total of $\mathcal{O}(2(n-1)n^2) = \mathcal{O}(n^3)$ synapses from this contribution.

The second contribution stems from constraints (2) and (3), namely inhibition between populations associated with vertex pairs that are not connected by an edge in the original graph. In the worst-case scenario, a completely disconnected graph, each population must inhibit all adjacent-position populations that represent unconnected vertices. This also results in a number of synaptic connections on the order of $\mathcal{O}(n^3)$. Therefore, the worst-case overall spatial complexity, considering both neurons and synapses, is cubic: $\mathcal{O}(n^3)$.

The temporal complexity of the algorithm is primarily determined by the number of iterations, or equivalently, by the maximum simulation time specified for the model execution. Recalling that we previously denoted this parameter by α, representing the total simulation time in discrete units, the temporal complexity of the algorithm is given by $\mathcal{O}(\alpha)$.

C.3. TSP Algortihm

To address the Traveling Salesman Problem (TSP), we consider a weighted graph $G = (V, E, w)$, where $w : E \to \mathbb{R}$ is a function assigning a real-valued cost to each edge.

We retain the previously proposed architecture but adapt the objective to probabilistically favor paths that correspond to lower-cost routes. To achieve this, we add $n \times n$ additional connections linking all populations to one another (one per vertex-position pair), via excitatory synapses defined as follows:

- The synaptic weight $\omega_{(i,j)}$ is set to a small positive value, sufficient to provide a slight reward to populations associated with more promising routes.
- The synaptic delay $\delta_{(i,j)}$ is defined as $\delta_{(i,j)} = w_{(v_i,v_j)}$, i.e., equal to the weight of the edge in the graph.

In this way, lower-cost routes (i.e., edges with smaller weights) induce shorter delays and therefore stimulate their target populations sooner. This mechanism introduces a temporal preference which, combined with the reward, promotes the activation of populations that could form part of more efficient solutions.

This algorithm is significantly more volatile, less refined, and remains experimental in nature. Its behavior strongly depends on the tuning of hyperparameters, as well as on the strategy used to incorporate weight information: for instance, whether a delay proportional to the edge weight is applied, or alternatively, whether a synaptic reward is modulated by the weights while keeping the delay constant. Depending on the configuration chosen, valid results may or may not emerge.

We have not yet conducted a systematic evaluation of the effectiveness of this approach, not even in comparison with the Hamiltonian cycle algorithm. Therefore, we cannot confidently assert whether the probabilistic potentiation introduced, through weight-modulated delays or reinforcements, has a genuine impact on the selection dynamics, or whether it constitutes a spurious influence that goes unnoticed during the evolution of the network.