Bachelor thesis

Artificial Intelligence

Radboud University

Complexity analysis on Dijkstra's algorithm:

Neuromorphic implementation compared against Von Neumann implementation.

Author:
Cis van Aken
S1032150
cis.vanaken@ru.nl

First supervisor:
A. Diehl MSc (Arne)
Donders Institute for Brain,
Cognition and Behaviour
arne.diehl@donders.ru.nl

Second supervisor :
dr. J.H.P. Kwisthout (Johan)
Donders Institute for Brain,
Cognition and Behaviour
j.kwisthout@donders.ru.nl

Second reader:
dr. S. Thill (Serge)
Donders Institute for Brain,
Cognition and Behaviour
serge.thill@donders.ru.nl

September 20, 2025

Abstract

The research will centre around analysing an existing implementation of Dijkstra's algorithm, adapted into a spiking neural network. A traditional implementation of Dijkstra's algorithm and a neuromorphic implementation of the same algorithm will be compared on their time, space and energy complexities. The idea is that the abstract complexities allow for such different computational philosophies to be compared more easily, by abstracting away from hardware and software design limitations and purely focusing on how an algorithm scales regarding the size of its input. Computational complexity theory does not currently provide a good framework to asymptotically approach energy complexity, so a novel approach to that topic is also included in this thesis. In the end, we found that the traditional implementation exceeds the neuromorphic implementations in most scenarios. In some scenarios, we found that the Neuromorphic implementation was slightly better, but the scenarios were very specific, and the benefit was not great enough to suggest any significant upside over traditional computing for this specific case.

Contents

Chapter 1

Introduction

Neuromorphic computing (NMC) is a field of computing science that has been around in some form or another for a while now, with computational models of biological neurons dating back to at least the 1950s[18]. However, only recently it emerged to prominence and is seen as one of the possible replacements for traditional, or "von Neumann", computing, now that we are approaching the limits of Moore's law[22, 32]. Instead of CPU and memory-based architectures, Neuromorphic systems make use of brain-inspired Spiking Neural Networks (SNNs) to compute[30]. Numerous different approaches to both hardware and software exist, differing mainly on neural models, analogue or more digital hardware and speed at which the simulation runs[15]. It is not completely clear what the capabilities of these systems are and what their eventual possibilities may lie, but several properties may provide tangible benefits concerning efficiency in specific areas of computing, without sacrificing computational power[4, 9, 10].

Since it is such a new field of research, plenty is still not fully understood. One of the areas that has not been seeing a lot of development is the field of complexity, and then specifically energy complexity. One of the main proposed benefits of neuromorphic computing is the improvement in the energy efficiency, and lower total consumption of energy, of algorithms and computers. With this paper, I will take a look at one specific algorithm, implemented in a Neuromorphic manner, and provide an extensive complexity analysis of that algorithm, covering space, time and energy complexities. The exact research question is as follows: Is there a significant improvement in complexity for an implementation of Dijkstra's algorithm in a neuromorphic simulator when compared to a traditional implementation?

Dijkstra's algorithm is a graph algorithm thought of by Edsger W. Dijkstra, to solve the single-source shortest-path (SSSP) problem[14]. The SSSP-problem is a graphing problem, in which the algorithm tries to find the shortest possible path from a start node, to a target node. Dijkstra's algorithm takes a graph $\mathcal{G} =< V, E >$, which is a tuple of the set V consisting of vertices and the set E of edges between two vertices, where node $v \in V$ and edge $e \in E$. A regular implementation of Dijkstra's algorithm is presented in algorithm 1.

Algorithm 1 Dijkstra's Algorithm

```
1: Set distance to all nodes to infinity
2: Source distance = 0
3: distances = new empty list
4: visited = new empty set
5: queue = new priority queue
6: queue.put(source, 0)
7: while Queue is not empty do
8:     state = queue.next()
9:     if state in visited then
10:        continue
11:    end if
12:    visited.add(current)
13:    for Neighbour in Graph.neighbours(current) do
14:        temporary_distance = distances[current] + Graph.distance(current, neighbour)
15:        if temporary_distance < distances[neighbour] then
16:            distances[neighbour] = temporary_distance
17:            queue.put(neighbour, distances[neighbour])
18:        end if
19:    end for
20: end while
21: Return distances
```

The rationale behind selecting Dijkstra's algorithm here lies in the similarities between the vertex-nodes model of graphs, when compared to the synapse-neuron models of Neuromorphic software, implying that the implementation of a graphing algorithm in a neuromorphic fashion should be relatively intuitive. The idea for this algorithm specifically is that it would be relatively simple to turn any graph into a spiking neural network and track a spike from the source node to the sink node, thus finding the shortest path, or the shortest path if multiple exist. Dijkstra-adjacent algorithms have also already been implemented in SNNs before, so plenty of material and possible solutions exist [23, 27].

An algorithm built and executed on a von Neumann architecture and an algorithm built and executed on neuromorphic hardware architecture cannot be compared directly, the parameters and environments are simply too different. An abstraction of the algorithm is needed, such that both implementations can be expressed in the same way and comparisons can be made. In computing science, computational complexity theory describes abstractly the fundamental resource requirements associated with different algorithms[25]. Most commonly, these are expressed in the sense of time required to find a solution, and space (e.g., memory) required to store the algorithm, the overhead and the data.

Since computers can actively forget (meaning that the previous logical state of the computer is not recoverable purely by its current state), and we know that that coincides with a certain increase in entropy, which in turn requires the consumption of energy, the energy used can also be considered such a resource[20]. This is but one possible approach to energy complexity, more elaboration on computational complexity will be given in section 2.4.

The main focus of this paper lies on the aforementioned energy complexity of the algorithm, with a small section dedicated to both the time and space complexities for Dijkstra's algorithm, both for its native implementation, as well as an adaptation into

the SNN. An increase in energy efficiency is hailed as one of the main benefits of NMC over current systems, but not a lot of research has gone into the scalability and computational complexity of the energy consumption of SNNs when compared to von Neumann implementations. This thesis aims to answer if SNNs are indeed more energy efficient in theory, also for large instances, when compared to regular von Neumann implementations, for one specific algorithm.

Chapter 2

Background

2.1 Neuromorphic Hardware

Neuromorphic computing is a relatively new field of computing science aimed at replacing traditional von Neumann computing. Instead of more traditional CPU and memory-based approaches, Neuromorphic computing takes inspiration from how the brain computes. Artificial neurons and synapses, either as physical hardware or represented in a simulation, do computations based on spiking patterns[22]. Though computational models of biological neurons date back to the 50s, and possibly earlier, only recently attempts to turn them into proper hardware or software models have become more prominent[18].

Since the 1990s, several distinct styles of hardware have emerged. These distinctions include a more analogue hardware focus vs. digital simulations, the scale of the system, the underlying neural model used, and the speed at which systems operate[15]. Additional information on different hardware can be found in section A.1, however for the thesis itself it is enough to understand that there is not one type of neuromorphic hardware, but that it is an umbrella term for a field of research into hardware. It is important to realize that each hardware architecture has distinct advantages and disadvantages, also when it comes to the complexity of its potential algorithms. This makes the choice of what hardware you use to solve a specific problem, a very important aspect of any eventual neuromorphic solution.

2.2 Spiking neural networks

Neuromorphic software mainly comes in the form of Spiking Neural Networks (SNNs). SNNs are a third-generation neural network architecture and a significant departure from traditional Deep Neural Network approaches, transmitting information through binary spikes, rather than packets of data[30, 24]. SNNs are a lot less linear than traditional first and second-generation perceptrons and artificial neural networks (ANNs), and apart from the set input and output layers, do not have any distinct layers, such as ANNs have, as shown in figure 2.1. In some cases, those input and output layers are reduced to a singular input neuron or read-out neuron, removing distinct layers entirely.

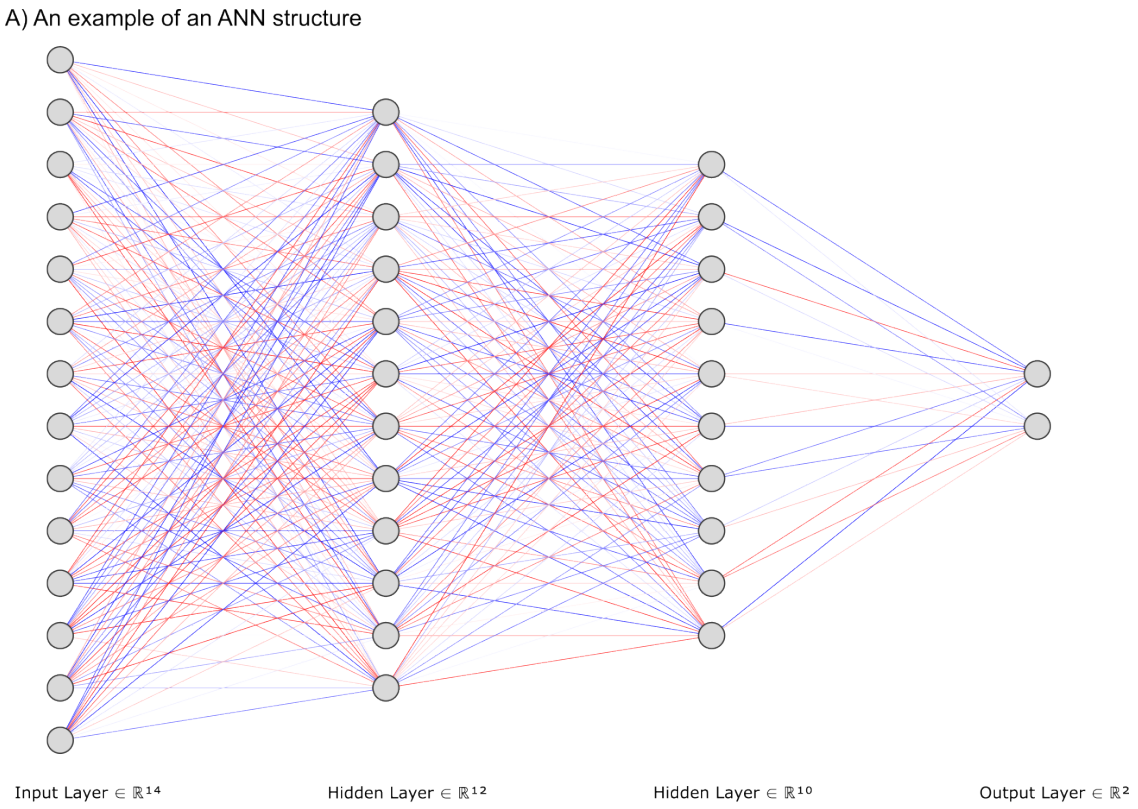

Figure 2.1:
A) A typical Artificial Neural Network structure with an input layer, two hidden layers and an output layer. The layers are fully connected. A blue connection is a connection with a positive weight, and a red connection has a negative weight. The opacity of the connections represents the size of the weight, with weights closer to zero, having a higher opacity.
B) A possible Spiking Neural Network (SNN) structure. Notice that it has a more free-flowing structure, with connections going back and forth between "layers", when compared to the ANN. Each connection has a direction, weight and delay. The yellow neurons are input neurons or input trains, and the grey neurons are the eventual readout neurons, whose spiking pattern determine the eventual output.

There are two major ways in which SNNs are used for computing, either by constructing a network and using plasticity in the neurons to have the network learn or by creating

a network and setting all the weights, voltages and thresholds in such a way, that the network can solve a specific, contextual problem[5]. The SSSP problem tackled in this thesis falls in the latter category. It is possible to take any graph with weighted edges and build a spiking network out of it by connecting pre-programmed leaky-integrate-and-fire (LIF) neurons. The LIF neuron mimics the behaviour of a biological neuron by integrating incoming signals over time. The neurons in the simulator used in this thesis, are modelled after the neurons defined by Severa et al., developed at the Sandia National Labs[28]. The "leaky" aspect refers to the gradual leakage or decay of the neuron's membrane potential when not stimulated. Once the membrane potential reaches a threshold, the neuron "fires" a spike, representing an output signal. After firing, the membrane potential is reset to a resting state[28]. Besides LIF neurons, most simulators use input neurons that spike based on provided data. Input neurons fire if and only if the data for a time step is equal to 1. Input neurons do not have a membrane voltage and only have outgoing synapses.

2.3 The SimSNN simulator

Most of the information in this section is taken directly from the SimSNN source code[3] or a Master thesis describing the simulator and a Graphical User Interface for it[31]. As far as I could find, these are the only 2 sources of information on SimSNN.

The algorithm analysed in this paper is constructed using a Python simulator called SimSNN[3], which is a simulator that focuses on networks and algorithms constructed with individual neurons, rather than groups of neurons that are interconnected, unlike several larger and more common simulators, such as PyNN[12] and Brian2[29]. The simulator itself implements the LiF neuron described in section 2.2, and allows for customization of several parameters, the most important of which are listed below.

- **Threshold (T):** T denotes the minimum value the voltage of a neuron has to have before it sends out a spike itself. A neuron spikes if and only if $V_i \geq T_i$ at time step i.
- **Reset voltage (R):** After spiking, the voltage of the neuron will return to this value.
- **Inverse leakage (M):** This parameter simulates the leakage of a LiF-neuron. When there is leakage, the voltage of a neuron gradually decreases back to its resting stage. Because we use inverse leakage, if $m = 1$ there is no leakage, and if $m = 0$, the neuron returns to its resting state in 1 time step.
- **Initial voltage** (V_0)**:** This setting allows users to set the voltage at time step T = 0 for any given neuron.
- **Minimum voltage** (V_{min}**:** This allows users to set the minimum voltage for any given neuron, the default setting is 0.
- **Constant Input Current**: Users are allowed to set an integer value that is added to the voltage of a neuron at each time step. This functions as a bias, where it increases the spiking rate of a particular neuron, giving more weight to its information output.

Every synapse establishes a connection from a pre-synaptic node to a post-synaptic node, featuring attributes such as weight (w) and delay (d). Since SimSNN is a discrete-timed simulator, the delay must be a whole integer. The duration of the simulation is determined by specifying the number of time steps to execute, during which the voltage and spiking behaviour of all nodes can be observed and analysed using detectors.

2.4 Computational Complexity Theory

This paper will touch upon 3 different types of computational complexity, 2 more common types and 1 that is less explored. As mentioned in section 1, Dijkstra's algorithm will be measured and compared on time, space and energy complexity.

There are three separate ways of talking about complexity, there are worst-case, best-case and average-case complexity, all of which say something different depending on the size and ordering of the input.

- Best-case complexity: this subclass says something about the efficiency when the algorithm is provided with the best possible input. For example, a sorting algorithm already is provided with a sorted or mostly sorted list.
- Worst-case complexity: worst-case complexity talks about how the algorithm behaves when it is provided with the worst possible input.
- Average-case complexity: average-case complexity talks about algorithms on the average. This case is often only defined with respect to a probability distribution, which describes how likely certain inputs are to appear. Often, all inputs are assumed equally likely.

For simplicity, when talking about complexity from here on out, it will be about the worst-case complexity or the upper bound. For this reason, we will be using Big-O notation, which denotes the upper bound of the scaling of an algorithm.

2.4.1 Time complexity

Time complexity is a fundamental concept in computational complexity theory that quantifies the efficiency of an algorithm by analysing the relationship between the input size and the amount of computational resources, specifically time, required for its execution. It provides a theoretical measure of how the running time of an algorithm grows as the size of the input increases. Time complexity is expressed using big-O notation, which describes the upper bound of an algorithm's growth rate. A lower time complexity signifies a more efficient algorithm, as it implies that the algorithm's execution time increases at a slower rate in response to larger inputs. Analysing time complexity is crucial for comparing and selecting algorithms that can efficiently solve computational problems across different input sizes and for predicting their scalability in real-world applications.

We take for example an algorithm that has a complexity of $\mathcal{O}(n^2)$ and an algorithm that has a complexity of $\mathcal{O}(nlog(n))$. We assume $n = 1.000.000$ and we assume that both algorithms are running on the same hardware and that that hardware can process about 1,000 operations per second. Algorithm 1 would take roughly 1^{10} seconds to complete, whereas algorithm 2 would only take a little under 2.000 seconds.

A naive implementation of Dijkstra's, using arrays to search through all nodes, is widely accepted to have a total runtime complexity of $\mathcal{O}(|V|^2)$, meaning that the runtime

increases quadratically with the number of vertices[14]. However, the version implemented in algorithm 1 makes use of priority queues to find the most likely next edge, bringing the total average-case time complexity down to $\mathcal{O}(|V| + |E| \cdot \log(|V|)$[26, 19, 7].

2.4.2 Space complexity

Similarly to time complexity, space complexity tries to quantify the efficiency of an algorithm, particularly by assessing the amount of storage and memory needed for executing the algorithm. Space complexity describes how the memory usage of an algorithm increases when the input of the algorithm is increased. Space complexity as well is also expressed in big-O notation. A lower space complexity implies a more memory-efficient algorithm.

When looking at algorithm 1, as described in section 1, we see that each node is stored once in the priority queue, before being transferred to the set "visited". We also see that each node is stored once in the list storing all the distances. Therefore, we know that each node is stored twice, and we can conclude that the space complexity is $\mathcal{O}(2|V|) \approx \mathcal{O}(|V|)$, so on the average case, the memory requirements of Dijkstra's algorithm increase linearly with the size of the input graph.

2.4.3 Energy complexity

Energy complexity is a new and relatively unexplored addition to computational complexity theory, as such, an exact framework does not yet really exist. What adds to the difficulty is that the energy usage of a system is a multifaceted concept and depending on the exact parameters of your system, different metrics may be appropriate.

One such metric that a lot of research is going towards, is that of reversible computing and its connection to energy consumption. Reversible computing is the field of computing science that explores the theoretical and practical applications of computational processes that can be fully reversed[13]. In traditional computing, the input is often transformed in a way that makes it difficult to return to a previous state. Reversible computing, on the other hand, aims to design computing systems where information loss is minimized or eliminated, allowing for the computation process to be entirely reversible. One of the primary motivations for reversible computing is the pursuit of energy-efficient computation. By minimizing information loss and adhering to the principles of reversibility, it is theoretically possible to reduce the energy dissipation associated with traditional computing. This is closely tied to Landauer's principle, which establishes a theoretical lower limit on the amount of energy dissipated during computation, as well as stating a minimum amount of energy required to delete 1 bit of information Landauer's principle implies that reversible computation has the potential to be highly energy-efficient[20]. A framework for energy complexity based on information erasure and Landauer's principle then makes sense to apply in the context of reversible computing[13]. This is a highly complex topic outside the scope of a Bachelor's thesis and mainly serves to illustrate the different avenues within energy complexity. Some more background information on Landauer's principle, reversible computing and its relevance for energy complexity is provided in the appendix A.2.

In this thesis, I use my own definition of complexity, which is inspired by the frameworks mentioned above, but more simplified. It is based less on reversible computing, in the sense that operations are not considered to be "free" if they are reversible and still add to the overall energy complexity. The exact mechanics are explained in section 3.1.1,

but it works under the assumption that each computation in an algorithm adds a small increase in energy consumption.

Chapter 3

Methods

In this section I will outline how I went about my eventual analysis and provide the frameworks I used. I will first provide a comprehensive explanation of the model for computational complexity for neuromorphic computing that I used. Since such a model is not yet fully standardized and explored, especially concerning energy complexity.

3.1 Neuromorphic complexity analysis

Before we can analyse the algorithm, we need to know how to approach neuromorphic software from a complexity point of view. Below, I will elaborate on the definitions and frameworks used for energy complexity in a similar manner as seen in section 2.4. Time and space complexity are a more straightforward conversion from traditional computing to neuromorphic computing and do not need a separate section for their explanations. For time complexity, next to taking loops in the traditional overhead into account, one should also consider the run-time of the simulation, which acts as a multiplier to potential loops happening in the simulation, since those loops happen for each time step. For space complexity, it is important to consider the size of the SNN together with the size of the graph in the von Neumann overhead.

3.1.1 Neuromorphic Energy Complexity

As stated in section 2.4, in this thesis I use my own definition of computing energy complexity, one that is more abstract and based on total computations done. This is done under the assumption that an algorithm consumes a base amount of energy and has a tiny spike in energy consumption when an operation is being performed. The figures seen in the paper by Gross et al. support this and their method is based on the framework explained in section 2.4.3[17]. A lot more factors than just computations and operations decide the actual energy consumption, such as the programming language, the hardware used and the ambient temperature of your surroundings, just to name a few[6]. However, for this thesis I assume that the energy consumption scales proportionally to the input size based on the memory used, and the operations performed, in order to abstract away from hardware limitations. This is done to be able to better compare SNNs with traditional implementations, which theoretically run on very different hardware and software environments.

3.2 Dijkstra-adjacent SNN

For the eventual analysis, I have received an existing implementation of Dijkstra's algorithm in a neuromorphic simulator. An explanation of the algorithm and its underlying principles is in order since Dijkstra is not traditionally a spiking neural network. The overarching idea behind the algorithm is to turn a graph into a spiking neural network, by converting each node into a neuron and all edges into synapses, with the weight of each edge, turned into the delay for a synapse. In algorithm 2 below, an explanation in pseudocode is given of the SSSP function that is provided in section 4 and is part of the full source code found in section 7.

Algorithm 2 Neuromorphic implementation of Dijkstra's algorithm

```
1:  function SSSP(Graph G, Source S)
2:      network = new Network()
3:      sim = new Simulator(net)
4:
5:      Neurons = []
6:      Synapses = []
7:      Destinations = []
8:      Neuron_to_dest = []
9:      Edge_neuron = []
10:     Dest_to_edge = []
11:     Neuron_to_edge = []
12:
13:     for vertex in G do
14:         Neuron = new LIF(ID = vertex.id, Threshold = 1000, V_initial = 999)
15:         Neurons.add(Neuron)
16:     end for
17:
18:     for edge in G.edges do
19:         synapse = new Synapse(pre=edge[0], post=edge[1])
20:         Synapses.add(synapse)
21:     end for
22:
23:     for pre and post in G.edges do
24:         destination = new LIF labelled pre-to-post
25:         Destinations.add(destination)
26:     end for
27:
28:     for pre and post in G.edges do
29:         synapse = new synapse from neuron "pre" to neuron "pre to post"
30:         Neuron_to_Dest.add(synapse)
31:     end for
32:
33:     for pre and post in G.edges do
34:         edgeNeuron = new LIF(Threshold = 2)
35:         Edge_neurons.add(edgeNeuron)
36:     end for
```

```
37:
38:     for edgeNeuron in Edge_neurons do
39:         synapse = new Synapse from corresponding neuron in Neurons to edgeNeuron
40:         Dest_to_edge.add(synapse)
41:     end for
42:
43:     for Neuron in Edge_neurons do
44:         synapse =new Synapse from corresponding neuron in Destinations to Neuron
45:         Neuron_to_edge.add(synapse)
46:     end for
47:
48:     setup_raster(sim, E, order) # Set up raster for simulation
49:
50:     sim.run()
51:
52:     raster_data = sim.extract_data # Retrieve raster data
53:     spiked = identify_spiked_neurons(order, raster_data) # Identify spiked neurons
    based on raster data
54:     L = map_shortest_paths(spiked) # Create a dictionary mapping each node to its
    shortest path
55:     Define lambda function extract_path_function()
56:     Return shortest_paths(source, extract_path_function())
57: end function
```

This is the function that ultimately computes the shortest path from a source neuron to all other neurons. All other blocks in the notebook are either test functions or steps in the iterative process of arriving at the full function, such as a function to compute the length of the shortest path from a source node to a single other node, and are thus not included in the text of the paper. A link to the full source code can be found in section 7. The primary idea is to represent nodes and directed edges of the graph as neurons and synapses, respectively, in an SNN. The simulation captures the dynamics of these neurons and synapses to determine the shortest paths from a designated source node to all other nodes in the graph. A step-by-step breakdown of the algorithm into its different phases is included below.

1. **Network Initialization:** A graph is created with the NetworkX python package for Graph creation[2]. Next to this, a simulator is initialized. This function comes from the SimSNN package and takes a network as input, from which it creates a raster which can then be filled with neurons and synapses[3].

   Neurons and synapses are created based on the edges and weights of the input graph and added to the raster and their respective dictionaries.

2. **Directed Edge Representation:** Additional LIF neurons are created and made to represent directed edges in the graph. Synapses are established between source neurons and these directed edge neurons, which in turn mimics the directionality of the edges.

3. **Path Length Calculation:** Next, the SNN is initialized. The membrane potential of the source neuron is set to match the spiking threshold, effectively causing it to spike and initiate the process of calculating the SSSP. The idea is that all neurons

spike only once until the target is reached. This is accomplished by initializing each neuron with an arbitrarily high threshold and setting the initial voltage 1 below that threshold. The reset voltage of the neuron, in turn, is then made to be 0, effectively making it, so each neuron spikes exactly once, forming a path from source to target, without being disrupted or restarted by interfering neurons.

4. **Path Extraction:** Raster data is collected directly from the simulation. Through this, all neurons that have spiked can be identified. If everything goes well, the shortest path can be reconstructed from this list.

5. **Path Reconstruction:** A recursive lambda expression is employed to reconstruct the paths based on the spiked neurons, determining the predecessors in the paths. In the end, we are left with a dictionary of shortest paths from the source to all other nodes.

This algorithm is not verified to be correct. However, with the small set of restricted graphs we test on, it does work correctly. This is checked by comparing the output from the neuromorphic implementation to NetworkX's direct implementation of Dijkstra's and verifying those outputs to be the same. A more in-depth analysis of the algorithm is required before anything can be stated on its correctness.

Chapter 4

Analysis

In this section, I will apply the framework I created in the methods section to the SNN implementation of Dijkstra's algorithm used in this thesis, as well as give a definition of the time and space complexity in big-O notation. The analysis will be based on the SSSP function, provided below. A more in-depth explanation of an example in pseudocode can be found in section 3.2 and the full notebook, with support functions that are not relevant to the analysis itself, is attached to the thesis.

```
1  import networkx as nx
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  from pprint import pprint
6  from random import randint
7
8  from simsnn.core.networks import Network
9  from simsnn.core.simulators import Simulator
10
11 def sssp(graph, source):
12     net = Network()
13     sim = Simulator(net)
14
15     N = {n: net.createLIF(ID=f"{n}", V_init=999, thr=1000) for n in g}
16     S = {e: net.createSynapse(ID=f"{e}", pre=N[e[0]], post=N[e[1]], d=g[e
           [0]][e[1]]["weight"]) for e in g.edges}
17
18     D = {f"{p}_d_{n}": net.createLIF(ID=f"{p}_d_{n}") for p,n in g.edges}
19     N_D = {(p,n): net.createSynapse(ID=f"{(p,n)}", pre=N[p], post=D[f"{p}_d_
           {n}"], d=g[p][n]["weight"]) for p,n in g.edges}
20
21     E = {f"{p}_e_{n}": net.createLIF(ID=f"{p}_e_{n}", thr=2, m=0) for p,n in
            g.edges}
22     D_E = {(f"{p}_d_{n}", f"{p}_e_{n}"): net.createSynapse(ID=f"({p}_d_{n},
           {p}_e_{n})", pre=D[f"{p}_d_{n}"], post=E[f"{p}_e_{n}"]) for (p,_,_,_
           ,n) in E}
23     N_E = {(int(n), f"{p}_e_{n}"): net.createSynapse(ID=f"(n,{p}_e_{n})",
           pre=N[int(n)], post=E[f"{p}_e_{n}"]) for (p,_,_,_,n) in E}
24
25     N[source].V = 1000 # Let the source neuron spike immediately, to start
           the path length calculation
26
27     order = np.array([e for e in E])
28     sim.raster.addTarget([E[e] for e in order])
29     sim.run(steps=50, plotting=False)
30
```

```
31      rasterdata = sim.raster.get_measurements().T
32      spiked = order[(rasterdata>0).any(axis=1)]
33
34      L = {int(n):int(p) for p,_,_,_,n in spiked}
35      path = lambda h, t, d: t if t[0] == h else path(h, [d.get(t[0], h)] + t,
            d)
36      return {n: path(source, [n], L) for n in L} | {source:[source]}
```

The algorithm itself consists of 3 main components, that can be analysed separately and then have their complexities added up in the end. This should make it easier to comprehend the individual steps and the eventual conclusion. The components are as follows, in order:

1. Network initialisation, from line 12 until 25.
2. Simulation, which is set up and then executed from line 27 until 29
3. Path extraction, which happens from line 31 until 36.

4.1 Time Complexity

To initialise the network, we create a new neuron for each vertex in V, as well as a synapse for each edge in E. Since we use a directed graph, neurons are also created representing each directed edge, and synapses are drawn between those. So we loop through E a total of 6 times. The creating of the neurons and synapses themselves are linear time steps, so we end with an upper bound of $\mathcal{O}(|V| + 6 \cdot |E|)$ to initialize the network, which scales the same as $\mathcal{O}(|V| + |E|)$.

Next, we run the simulation for T time steps. At each time step, all the nodes N and synapses S in the SNN are updated. This happens internally by looping over all nodes, followed by looping over all synapses, as shown in the code snippet below, taken directly from the source code[3].

```
1     def step(self):
2         for node in self.nodes:  # update all nodes
3             node.step()
4         for synapse in self.synapses:  # update all synapses
5             synapse.step()
```

Internally, the step functions for both node and synapses only contain constant time functions. This means we end up with an upper bound complexity of $\mathcal{O}(|N|+|S|)$ for each step that we run the simulation for, leading to a total time complexity of $\mathcal{O}(T \cdot (|N|+|S|)$ for the simulation.

When extracting the path, we create a new array from the dictionary E containing all neurons representing directed edges. Recall that the amount of neurons in E depends on the total number of edges in G. Creating the array called "spiked" and sorting it, as well as creating the dictionary "L" both have worst-case complexity of $\mathcal{O}(|E|)$. When we finally return the paths, we call the constant time lambda function on each element in "L". These steps together lead to a total time complexity $\mathcal{O}(E)$ for this step.

Our algorithm then has a total time complexity of $\mathcal{O}(|V|+|E|+T \cdot (|N|+|S|)+|E|)$. This can be further simplified to $\mathcal{O}(|V| + |E| + T \cdot (|N| + |S|)$. If we consider that $|N| = |V| + 2|E|$ and $|S| = 4|E|$, we can simplify the complexity further to an upper bound complexity of $\mathcal{O}(|V| + |E| + T \cdot (|V| + 6|E|) \sim \mathcal{O}(|V| + |E| + T \cdot (|V| + |E|))$

4.2 Space Complexity

To start, we store the newly created neurons and synapses in dictionaries. We create a neuron for each vertex in V, and we create a total of 6 neurons or synapses for each edge in E, leaving us with a space complexity of $\mathcal{O}(|V| + |E|)$.

Next, before we simulate, we create a new array "order". Creating the NumPy array from the keys of the dictionary E requires additional space. The space complexity for this operation is $\mathcal{O}(|E|)$, since the size of dictionary E is dependent on the amount of edges in the original graph. After this, we add references to the neurons to the raster. Adding elements to the raster target involves storing references to the neurons in the raster. The space complexity for this operation is $\mathcal{O}(|E|)$ as well, since the amount of neurons stored depends again on dictionary E.

Following this, we store yet more items, in order to correctly extract the path. First, we collect the data from our raster and store that in a new array, then we create a new array in which we store all the neurons from "order" that have spiked, and we create a new array "L" with elements based on the size of "spiked". Finally, we return our findings in another dictionary, for each element in "L". All of these new arrays and dictionaries depend ultimately on the size of dictionary E, which in turn depends on the amount of edges in the graph. In the end, this leaves us with a space complexity of $\mathcal{O}(|E|)$.

For the full algorithm, we have obtained a space complexity of $\mathcal{O}(|V| + 12|E|) \sim \mathcal{O}(|V| + |E|)$

4.3 Energy Complexity

As mentioned in section 3.1.1, I will be using total operations performed and neurons that fired in the SNN, to estimate how both the traditional algorithm and the SNN should scale in their energy consumption. I will start with applying it to the traditional algorithm as seen in algorithm 1, as that was missing from section 2.4.3. I will assume each operation adds 1 unit of energy and that each spike contributes 1 unit of energy as well.

4.3.1 Traditional algorithm

Let's denote the number of vertices as $|V|$, and the number of edges as $|E|$.

The initialisation phase requires $|V|$ operations, to set the distances to all vertices to ∞. Setting the distance to the source to 0 and initialising the distances list, visited set, and priority queue all require multiple operations, but all have a complexity of $\mathcal{O}(1)$.

The main loop contains 3 things of note for the energy complexity;

- Extracting the minimum distance node from the priority queue. The search for the minimum distance happens in $\log(V)$ operations.
- Iterating over neighbours. In the worst case, all nodes are neighbours of the current node, so the worst-case complexity for each iteration is $\mathcal{O}(|E|)$.
- Then putting the neighbour back in the queue, at the appropriate place takes $\log(V)$ operations again.

Everything considered, the total energy complexity of the algorithm would then be $\mathcal{O}(|V| + |E| \cdot \log(|V|))$

4.3.2 Spiking Neural Network

Let's designate the number of vertices as $|V|$, the number of edges as $|E|$, and the number of simulation steps as T.

Creating neurons involves $\mathcal{O}(|V|)$ operations, and creating synapses involves six operations per edge, leading to a complexity of $\mathcal{O}(6|E|)$. Since we're primarily concerned with the order of magnitude, this is considered $\mathcal{O}(|E|)$. Just as with the time complexity, creating the "order" array and adding all targets to the raster depend on dictionary E and have a complexity of $\mathcal{O}(|E|)$.

The simulation involves T steps, each requiring operations proportional to the sum of vertices and edges. This results in a complexity of $\mathcal{O}(T \cdot (|V| + |E|))$. Note that this is because the algorithm is run on traditional hardware and the complexity for simulations is assumed to equal the amount of spikes, which is at most $\mathcal{O}(|E|)$.

All following steps in the algorithm are either constant or linear with $\mathcal{O}(|E|)$. Everything added up, the total number of operations is approximately $\mathcal{O}(|V| + |E| + T \cdot (|V| + |E|))$. Or $\mathcal{O}(|V| + |E|)$ if we assume the simulation to run on true neuromorphic hardware and not rely on a traditional overhead, allowing us to focus on only the spikes of the algorithm.

Chapter 5

Discussion

I have given an in-depth comparative analysis of the complexity of Dijkstra's algorithm in both a traditional implementation and in the form of an SNN. Below I will discuss what can be concluded from this thesis, as well as potential points of limitation and possible improvements to be made.

5.1 Conclusion

The traditional implementation and neuromorphic implementation have different complexities for all 3 metrics. Let us go through them one by one.

5.1.1 Time complexity

Recall that the time complexity of the traditional implementation was $O(|V| + |E| \cdot \log(|V|))$ and the time complexity of the neuromorphic implementation was $O(|V|+|E|+T \cdot (|V| + |E|))$. Which implementation is then favourable over the other depends on a few different aspects. If we run the simulation for a short amount of time, (e.g., T is relatively small), $O(|V| + |E| + T \cdot (|V| + |E|))$ can be simplified to $O(|V| + |E|)$. This is comparable to the complexity of the traditional implementation, which may be favourable for graphs with fewer edges. When $|E|$ is significantly smaller than $|V|$, so when the graph is sparsely connected, the traditional implementation could be more favourable. When T is sufficiently large, $T \cdot (|V|+|E|)$ would dominate and the traditional implementation would be more favourable, since it would grow at a slower rate, for a larger T. The more favourable implementation regarding time complexity would be the traditional implementation in most scenarios, or it would make little difference to use one over the other.

5.1.2 Space complexity

Now recall that the space complexity for the traditional implementation is akin to $O|V|$ and the complexity for the neuromorphic simulation is $O(|V|+|E|)$. In this case, we have two scenarios.

1. $|E|$ is sufficiently large, either proportional to $|V|$ or larger: In scenarios such as this, the complexity $O(|V| + |E|)$ is larger than $O(|V|)$.
2. Graph G is not fully connected and sparse: this would mean that the number of edges $|E|$ is much smaller than the number of vertices $|V|$, then the complexity $O(|V|)$ might be larger.

Scenario 2 will not occur in a regular scenario, since we assume fully connected graphs, to be able to find a path. Therefore $|E| \geq |V| - 1$.

5.1.3 Energy complexity

For energy complexity, we found that the complexity for the traditional implementation was $O(|V| + |E| \cdot \log(|V|))$ and the complexity of the neuromorphic implementation was $O(|V| + |E| + T \cdot (|V| + |E|))$. The same scenarios apply here as for the time complexity, where the neuromorphic implementation is slightly better for a larger number of edges and a shorter simulation. The same also applies if we grab the complexity as if we would construct the SNN and let it run on neuromorphic hardware, instead of in a simulated environment, with $O(|V|+|E|)$, where the complexities are similar, unless for a sufficiently large E.

5.1.4 Final conclusion

With all of that being said, these results suggest there to be little incentive to use neuromorphic computing over von Neumann computing to implement Dijkstra's algorithm and solve the SSSP problem if your main goal is to use a more efficient and scalable algorithm.

Note that the energy complexities are very similar to the found time complexities. This does make sense in a way, since an algorithm will consume more energy the longer it runs. In some sense, you could say that ultimately the energy consumption of an algorithm is bounded by the runtime and thus its energy complexity is bounded by its time complexity.

5.2 Limitations

The first and largest limitation of this research is my own lack of knowledge on the topic before going into the thesis. Both neuromorphic computing and computational complexity theory are highly abstract and complex topics. The first months of this research were spent figuring out what the topic was and what I wanted to do within it, while having to adjust that several times along the way because I could not figure out how to do certain things I wanted to do, such as using actual neuromorphic hardware.

What we end up with is a very small-scale, minimal research project with a complexity framework that is a massive oversimplification of reality. In reality, the energy complexity of algorithms, especially SNNs, depends on a lot more than just computations and spikes. Such as, but not limited to, hardware restraints, software used, compilers used, ambient temperatures and the neural models used. Even what software is running in the background, besides your algorithm, could be of influence.

Another major limitation is the fact that the neuromorphic implementation was built in a simulator, rather than built for actual hardware. This changes the way the construction of the algorithm is approached, as well as adding extra overhead and background computations that are necessary to approach actual neurons and synapses firing.

Nevertheless, I do still think the framework I thought off of, however flawed it may be, could still provide a good diving board for someone who has more knowledge on both topics to make use of and turn into some viable metric that can be used to classify energy complexity.

5.3 Suggestions for further research

One idea for further research is to take the framework for energy complexity I proposed and refine it further. This would both allow this research to be redone and provide better results for what is now my thesis, as well as allow much easier analysis of algorithms based on their energy consumption as a whole.

One could also go in a different direction and analyse different types of algorithms, such as search algorithms, and see how novel neuromorphic implementations stack up to their traditional counterparts on that front. Or Dijkstra's could be implemented on actual neuromorphic hardware, which would provide the opportunity to verify, or disprove, my findings.

There is plenty of options for further research in this field still left.

Chapter 6

Acknowledgments

All the code used in this project has been provided to me by my supervisors. Early in the project, an attempt was made to create an SNN from scratch, several weeks were spent on this, before ultimately abandoning the idea in favour of rewriting the provided code to a package that could be compiled and executed on neuromorphic hardware. Ultimately, this plan was also abandoned and the idea of using neuromorphic hardware was scratched completely, due to it being out of scope for the level of a bachelor thesis.

Chapter 7

Data and source code

All the code used is written and accessible in the following notebook: `https://colab.research.google.com/drive/1MyiIaohoGxE0PIngQsZHOQGJBSF_LiWp?usp=sharing`

Bibliography

[1] *Human Brain Project & EBRAINS.*

[2] *NetworkX — NetworkX documentation.*

[3] *NeuromorphicComputing / simsnn · GitLab*, March 2023, Publication Title: GitLab.

[4] James B. Aimone, Yang Ho, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Yipu Wang, *Provable Advantages for Graph Algorithms in Spiking Neural Networks*, Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures (Virtual Event USA), ACM, July 2021, pp. 35–47 (en).

[5] James B. Aimone, Ojas Parekh, and William Severa, *Neural computing for scientific computing applications: more than just machine learning*, Proceedings of the Neuromorphic Computing Symposium (New York, NY, USA), NCS '17, Association for Computing Machinery, July 2017, pp. 1–6.

[6] Susanne Albers, *Energy-efficient algorithms*, Communications of the ACM **53** (2010), no. 5, 86–96.

[7] baeldung, *Understanding Time Complexity Calculation for Dijkstra Algorithm \textbar Baeldung on Computer Science*, July 2021.

[8] Harry Buhrman, John Tromp, and Paul Vitányi, *Time and space bounds for reversible simulation*, Journal of Physics A: Mathematical and General **34** (2001), no. 35, 6821 (en).

[9] Prasanna Date, Bill Kay, Catherine Schuman, Robert Patton, and Thomas Potok, *Computational Complexity of Neuromorphic Algorithms*, International Conference on Neuromorphic Systems 2021 (Knoxville TN USA), ACM, July 2021, pp. 1–7 (en).

[10] Prasanna Date, Thomas Potok, Catherine Schuman, and Bill Kay, *Neuromorphic Computing is Turing-Complete*, Proceedings of the International Conference on Neuromorphic Systems 2022 (Knoxville TN USA), ACM, July 2022, pp. 1–10 (en).

[11] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong Wang, *Loihi: A Neuromorphic Manycore Processor with On-Chip Learning*, IEEE Micro **38** (2018), no. 1, 82–99, Conference Name: IEEE Micro.

[12] Andrew Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger, *PyNN: a common interface for neuronal network simulators*, Frontiers in Neuroinformatics **2** (2009).

[13] Erik D. Demaine, Jayson Lynch, Geronimo J. Mirano, and Nirvan Tyagi, *Energy-Efficient Algorithms*, Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (Cambridge Massachusetts USA), ACM, January 2016, pp. 321–332 (en).

[14] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), no. 1, 269–271 (en).

[15] Steve Furber, *Large-scale neuromorphic computing systems*, Journal of Neural Engineering **13** (2016), no. 5, 051001 (en).

[16] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown, *Overview of the SpiNNaker System Architecture*, IEEE Transactions on Computers **62** (2013), no. 12, 2454–2467.

[17] Joshua B. Gross, Daniel Jacoby, Kevin Coogan, and Aaron Helman, *Motivating complexity understanding by profiling energy usage*, Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (New York, NY, USA), Onward! 2021, Association for Computing Machinery, October 2021, pp. 85–96.

[18] A. L. Hodgkin and A. F. Huxley, *A quantitative description of membrane current and its application to conduction and excitation in nerve*, The Journal of Physiology **117** (1952), no. 4, 500–544.

[19] Piotr Jurkiewicz, Edyta Biernacka, and Jerzy Domz, *Empirical Time Complexity of Generic Dijkstra Algorithm*, (2021) (en).

[20] R. Landauer, *Irreversibility and heat generation in the computing process*, IBM Journal of Research and Development **5** (1961), no. 3, 183–191.

[21] Yves Lecerf, *Logique mathématique : 1. Machines de Turing réversibles. Récursive insolubilité en n epsilon ny de l'équation u = theta n u, où theta est un "isomorphisme de codes". 2. Récursive insolubilité de l'équation générale de diagonalisation de deux monomorphismes de monoïdes libres phi x = psi x*, Published in **1964** in Bruxelles by Euratom (1964) (fre), Publisher: Bruxelles : Euratom.

[22] C. Mead, *Neuromorphic electronic systems*, Proceedings of the IEEE **78** (1990), no. 10, 1629–1636.

[23] E. Mérida-Casermeiro, J. Muñoz-Pérez, and R. Benítez-Rochel, *Neural Implementation of Dijkstra's Algorithm.*, Computational Methods in Neural Modeling (Berlin, Heidelberg) (José Mira and José R. Álvarez, eds.), Lecture Notes in Computer Science, Springer, 2003, pp. 342–349 (en).

[24] 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** (2023), no. 12, 243:1–243:49.

[25] Ahmet Celal Cem Say, *Energy Complexity of Computation*, Reversible Computation (Cham) (Martin Kutrib and Uwe Meyer, eds.), Lecture Notes in Computer Science, Springer Nature Switzerland, 2023, pp. 3–11 (en).

[26] Alexander Schrijver, *On the history of the shortest path problem*, Optimization Stories (Martin Grötschel, ed.), EMS Press, 1 ed., January 2012, pp. 155–167 (en).

[27] 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*, Proceedings of the 7th Annual Neuro-inspired Computational Elements Workshop (New York, NY, USA), NICE '19, Association for Computing Machinery, 2019, pp. 1–6.

[28] William Severa, Ojas Parekh, Kristofor D. Carlson, Conrad D. James, and James B. Aimone, *Spiking network algorithms for scientific computing*, 2016 IEEE International Conference on Rebooting Computing (ICRC), October 2016, pp. 1–8.

[29] Marcel Stimberg, Romain Brette, and Dan FM Goodman, *Brian 2, an intuitive and efficient neural simulator*, eLife **8**, e47314.

[30] Clarence Tan, Marko Šarlija, and Nikola Kasabov, *Spiking Neural Networks: Background, Recent Development and the NeuCube Architecture*, Neural Processing Letters **52** (2020), no. 2, 1675–1701 (en).

[31] Niels van Harten, *Creating a pipeline to graphically design and execute spiking network algorithms*, (2023).

[32] Fulai Zhu, Peiyu Xu, and Jiahao Zong, *Moore's Law: The potential, limits, and breakthroughs*, Applied and Computational Engineering **10** (2023), 307–315.

Appendix A

Appendix

A.1 Examples of Neuromorphic Hardware

As mentioned in section 2.1, several different hardware architectures exist in the field of NMC. Early on, during the process of writing the thesis, the idea was to compare an implementation of an algorithm on two different types of neuromorphic hardware, which was eventually dropped and the section on hardware was removed almost entirely. For this reason, a lot of time was put into examining different architectures, and a larger section was dedicated to it in the original outline. In later iterations, the section was removed and ultimately largely moved to the appendix, to not waste the effort put into it and because I believe that it is still useful information to contextualise the research question, the analysis and ultimately the conclusion, even if it is not necessary to fully comprehend the thesis itself.

As talked about earlier, there is not one approach to neuromorphic hardware. Rather, there are several prominent ones that I would like to talk about most, to highlight differences and potential benefits and drawbacks between the different approaches. There is Intel's Loihi and Loihi 2 neuromorphic chips, as well as the Human Brain Project's "Manchester SpiNNaker" and "BrainScaleS" in particular that I want to talk about. These are systems that all function in a vastly distinct way and serve neatly to illustrate the broadness of the field.

Intel Loihi is a neuromorphic computing chip, designed to mimic the neurobiological functioning of the brain. Loihi's architecture integrates memory and processing on the same chip, which allows for more parallel processes and distributed memory, which more closely resembles the way neurons operate in the brain. Loihi is a fully digital system, that approximates continuous time with discrete time-step models[11]. The chips also support plasticity to a certain degree, meaning that it is well suited for SNNs requiring continuous learning and updating.

SpiNNaker, which is a contraction for Spiking Neural Network Architecture, is one of the flagship neuromorphic machines created by the Human Brain Project (HBP)[1]. The physical machine is located in Manchester in the United Kingdom, and is accessible through the EBRAINS environment. The system consists of 1 million different processor cores, all interconnected and well-suited to simulate neural behaviours. The individual units are arranged in a 2D-mesh grid, with communication between units made possible by larger information routers on each core. Intel's Loihi system works with a similar mesh-based design. Instead of using physical neuron and synapse models, it simulates the behaviour of neurons on its processing cores. An illustration is provided in figure A.1. It implements a simple point neuron model, where the tree-like structure of synapses is

ignored, and all inputs are applied directly to the neuron. Cell dynamics are also not taken into account, and the complexities of the structure of a neuron is abstracted to a single "point"[16].

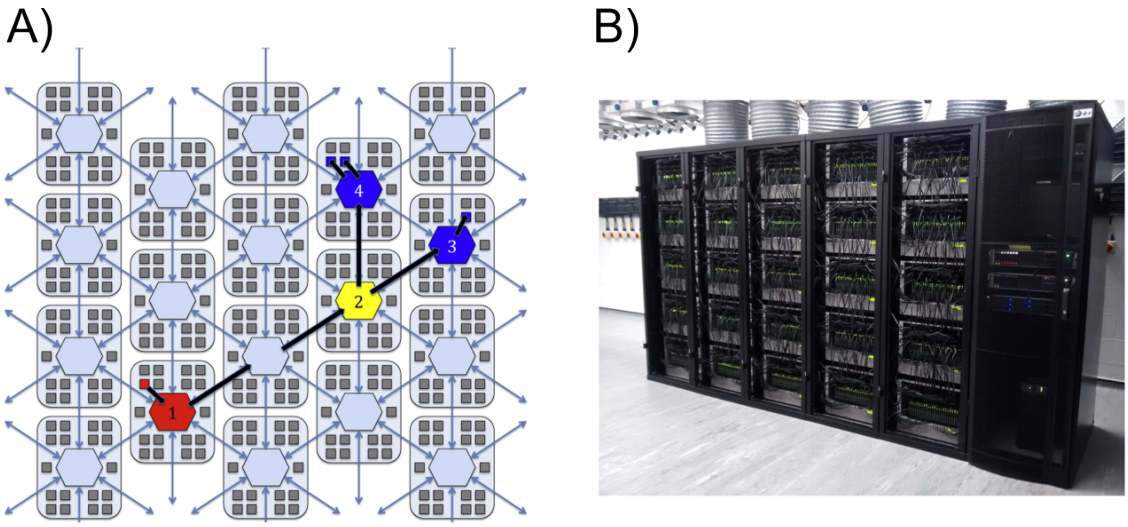

Figure A.1: The SpiNNaker system.
A) The internal system architecture works via a Mesh grid made of different processing cores. Smaller neuron cores (the grey squares) send information on spiking data to the local information router (the hexagons), which in turn communicate these information packets with other cores.
B) An older version of the SpiNNaker machine with 500,000 processing units.
Images taken from a paper by Furber et al. on large scale neuromorphic machines[15]

The BrainScaleS system is located in Heidelberg, Germany, and uses analogue circuits to simulate neurons and synapses, creating a physical model of neural computing. Its above-threshold circuitry allows the system to run at up to 10,000 times biological speeds[15]. This makes it especially adept at simulating long neural processes, such as learning over multiple years.

All architectures make use of SNNs for computation, however, they do so in different manners, which would impact the way an algorithm is written for the specific system. The different philosophies also present different challenges for energy complexities and different aspects to take into account. BrainScaleS' above-threshold circuitry would increase energy consumption since its transistors are always "on", whereas Loihi and SpiNNaker do not suffer that problem. However, Loihi and SpiNNaker make use of more digital and simulated components than BrainScaleS does, increasing energy consumption that way. BrainScaleS also offers faster simulations, whereas SpiNNaker and Loihi offer larger simulations with more Neurons and synapses. All-in-all, a lot of different considerations go into the creation of neuromorphic hardware and no system is quite the same.

A.2 A Note on Reversible computing and Landauer's principle

It was Landauer who first argued that a computational process could only be physically reversible if the process was then also logically reversible. If a computation is logically

reversible, it means that the previous state of the algorithm can be obtained purely by looking at the current state, without additional information needed. He then followed that with the observation that the erasure of 1 bit of information, requires a minimum amount of energy, which he defined as follows:

$$E = k \cdot T \cdot \ln 2 \tag{A.1}$$

k in this equation is Boltzmann's constant ($k \approx 1.38 \cdot 10^{-23} J/K$) and T is the ambient temperature. This is known as Landauer's principle, or Landauer's bound, and it denotes a theoretical lower bound for the energy consumption of computation. In essence, it shows a connection between information theory and thermodynamics and proposes a loss of energy for each bit discarded. Most CPUs discard plenty of bits, often at least one at every single logic gate. Currently, it is theorized that Landauer's principle can be circumvented entirely with logically reversible computing[20, 13].

Reversible computations are a relatively old idea within computing science, with reversible Turing machines being proven universal for the first time around 1963[21]. In later years, it was proven that any algorithm can be made reversible, but it would increase either the space complexity or the time complexity drastically[8]. This implies that there is a tradeoff between space, time and energy complexities.

The paper by Demaine et al. attempts to analyse these tradeoffs and define a metric for energy complexity based around it. In their system, they distinguish between reversible operations, that add nothing to the energy complexity, and destructive operations, that do add to the energy complexity. Through those efforts, they have defined energy complexities of several sorting and graphing algorithms, while maintaining the integrity of traditional time and space complexity[13].