Implementation of a Distributed Minimum Dominating Set Approximation Algorithm in a Spiking Neural Network

Victoria Bosch*1[0000−0001−7454−8325], Arne Diehl*1[0000−0001−9702−1083], Daphne Smits*1[0000−0002−3737−6672], Akke Toeter*1[0000−0002−9577−920X], and Johan Kwisthout2[0000−0003−4383−7786]

[1] School for Artificial Intelligence, Radboud University, Montessorilaan 3 6525 HR Nijmegen, the Netherlands
[2] Donders Center for Cognition, Radboud University, Montessorilaan 3 6525 HR Nijmegen, the Netherlands
j.kwisthout@donders.ru.nl

Abstract. Neuromorphic computing is a promising new computational paradigm that may provide energy-lean solutions to algorithmic challenges such as graph problems. In particular, the class of distributed algorithms may benefit from translation to spiking neural networks. This work presents such a translation of a distributed approximation algorithm for the minimum dominating set problem, as described by Kuhn and Wattenhofer (2005), to a spiking neural network. This translation shows that neuromorphic architectures can be used to implement distributed algorithms. Subcomponents of this implementation, such as the calculation of the minimum or maximum of two numbers and degree of a node, can be reused as foundational building blocks for other (graph) algorithms. This work illustrates how leveraging neural properties for the translation of traditional algorithms relies on novel insights, thereby contributing to a growing body of knowledge on neuromorphic applications for scientific computing.

Keywords: Neuromorphic Computing, Spiking Neural Network, Distributed Computing, Minimum Dominating Set, Graph Algorithms

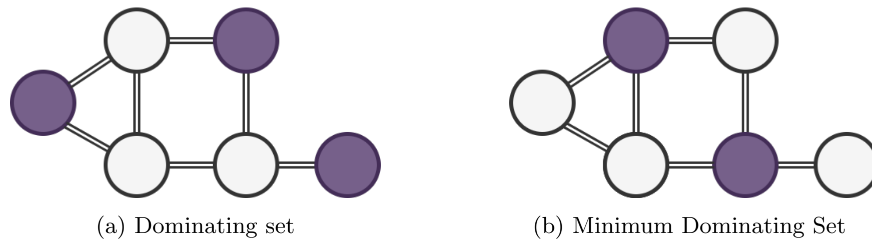
1 Introduction

Neuromorphic Computing is a relatively young field that concerns itself with emulating the brain and bringing advantages that are inherent to its structure into computational devices and programs. These advantages include parallel architecture, co-location of memory and computation, and high energy efficiency [14]. Moving away from the traditional von Neumann-architecture is an avenue from which various areas of research and development could benefit. In particular, the use of neuromorphic architectures has prompted the development of new methods and algorithms for scientific computing [16].

* equal contribution

The basic architecture and computational model underlying neuromorphic hardware [4, 2] is the spiking neural network (SNN). SNNs offer a great potential in finding solutions to graph problems. The translation process of graph algorithms to SNNs is relatively new, and there is still much knowledge to be gained about the optimal conversion and optimisation. The structure of SNNs (neurons and synapses) is similar to that of graphs (vertices and edges), allowing any graph to be represented by a spiking neural network. Thus, one possible approach is to use the inherent structure of the graph to solve various problems by letting the vertices communicate with spikes and spike timing (message-passing). Recent work that takes this approach is the partial translation of an algorithm for the max network flow problem to neuromorphic hardware [2], implementation of a SAT solver [21], and an exploration of neuromorphic algorithms for the longest shortest path and minimum spanning tree [9]. Another approach for the usage of SNNs for graph problems is to view every neuron as a computational unit. Manually programming and designing the network may be unconventional, yet this approach enables increased control in tailoring of SNNs for various (graph) problems. For example, Aimone et al. present a conversion method for the class of dynamic programs [1].

However, there are more classes of algorithms that may benefit from neuromorphic architectures, especially the class of distributed algorithms. This is because distributed computing traditionally requires multiple CPUs, whereas the neurons in an SNN can function as a population of computational units within one device.



(a) Dominating set (b) Minimum Dominating Set

Fig. 1: *Examples of a dominating set and a minimum dominating set of a graph (shown in purple).*

To demonstrate the potential of programming SNNs for distributed graph algorithms, we show a conversion of a distributed approximation algorithm for the *minimum dominating set* (MDS) graph problem. Kuhn and Wattenhofer [11] have constructed a distributed and constant-time approximation algorithm for the MDS problem. The MDS of a graph is a smallest subset of the vertices in G, such that for every vertex it either is in the dominating set, or one of its direct neighbours is (see fig. 1). The Kuhn-Wattenhofer algorithm is a parallelised greedy algorithm and achieves an expected approximation ratio of $\mathcal{O}(k\Delta^{(2/k)}log\Delta)$ in $\mathcal{O}(k^2)$ rounds where k is an arbitrary parameter that denotes the number of iterations of the approximation algorithm and Δ is the maximum

degree in graph G. Because of its distributive nature and constant-time complexity, we have found the algorithm fit to be effectively implemented in a spiking neural network. Thus, we present a spiking neural network implementation of the Kuhn-Wattenhofer approximation algorithm for the minimum dominating set problem, in order to show the potential of programming SNNs.

2 Preliminaries

2.1 Graph Definitions

A graph $G = (V, E)$ with edges $(u, v) \in E$, consists of vertices $V = (v_1, ..., v_n)$ and edges $E = (e_1, ..., e_m)$, where n and m represent the number of vertices and edges in graph G respectively. The degree δ_i represents the number of connected vertices for an arbitrary vertex v_i with $i \in [1, n]$. Alternatively, $\delta_i^{(1)}$ and $\delta_i^{(2)}$ denote the maximum degree in a one- and two-step neighbourhood of the vertex v_i respectively. Δ denotes the maximum degree in the graph G.

2.2 Minimum Dominating Set

The functional *minimum dominating set problem* is defined as follows:

> **Minimum Dominating Set**
> **Input:** Undirected graph $G = (V, E)$.
> **Output:** Subset D in which $D \subseteq V$ if $v \in V$ is in D or adjacent to D, and no subset of D is a dominating set of G.

The minimum dominating set problem has historical roots in the k-queens problem and is related to the set cover problem.The set cover problem can be considered equivalent to MDS under L-reduction [7]. The MDS problem is one of the first graph problems shown to be NP-hard [5]. The best logarithmic approximations of the MDS are achieved by hybrid algorithms that make use of greedy algorithms and LP-rounding [13]. For a recent overview of the performance of various approximation algorithms for the MDS problem, we refer to [13]. Potential applications for algorithms that solve the minimum dominating set problem include the clustering of wireless devices in a network [8] and automatic text summarisation [20].

2.3 Neural Model

Here we define a spiking neural network as a finite directed graph with vertices and edges, where the vertices are neurons and the edges function as synapses. We make use of the leaky-integrate-and-fire (LIF) neural model, which is commonly used in neuromorphic hardware. A LIF-neuron is defined by its initial voltage (V_{init}), the activation threshold (thr), the amplitude of the spike that occurs when threshold is met (amp), the leakage constant (m) which decreases the voltage over time, and the reset voltage to which the neuron returns after

spiking (V_{reset}). Neurons can either be deterministic or stochastic, which determines their spiking behaviour. A deterministic neuron spikes when its voltage has reached its threshold. A stochastic neuron however, will spike according to some probability distribution p. Neurons are connected by synapses, which are defined by the pre- and post-synaptic neuron that they connect, the weight (w) of their connection, and the time delay (d) of the signal. Spiking neural networks can take input from various sources, for example, the programmed V_{init} of a neuron, or from neurons that are programmed to spike at a certain time. The graphical notation for spiking neural networks in this paper is defined in fig. 2 and is based on the notation presented in [2].

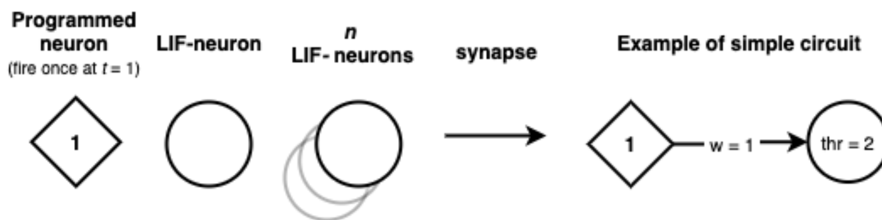


Fig. 2: *Notation for the graphical representation of spiking neural networks. If the values are set to default values, they are omitted from the shown graph.*

2.4 Kuhn-Wattenhofer algorithm

The Kuhn-Wattenhofer algorithm consists of two parts. It commences by solving the LP-relaxation of the problem (in alg. 2), and then uses the solution, the α-approximation ($\underline{x}^{(\alpha)}$) of LP_{MDS}, to approximate the integer program (in alg. 1). Using distributed randomised selection, a solution $\underline{x}_{DS}$ for the integer program (IP_{MDS}) is found, where $\underline{x}_{DS}$ consists of a binary list indicating which vertices are in the dominating set that approximates the minimum dominating set.

The approximation of LP_{MDS} (alg. 2) contains the main functionality of the Kuhn-Wattenhofer algorithm, as it returns the approximation of the related linear programming solution LP_{MDS}. It is a distributed greedy algorithm, where each vertex v_i dynamically calculates the α-approximation x_i for the solution to LP_{MDS}. To that end, each vertex also has a variable *colour*, which is initially white and turned grey if the vertex is considered covered. Each vertex also has a variable dynamic degree $\tilde{\delta}(v_i)$, which is equal to the number of vertices in the closed neighbourhood (that includes the vertex itself) that are white. As initially all vertices are white, the dynamic degree is initialised to the number of vertices in its closed neighbourhood ($\delta_i + 1$). The algorithm contains two nested loops. For every iteration, the vertices with a dynamic degree $\tilde{\delta}(v_i)$ above the threshold, raise x_i. Next, the dynamic degree $\tilde{\delta}(v_i)$ is updated according to the

Algorithm 1: Kuhn-Wattenhofer - $LP_{MDS} \rightarrow IP_{MDS}$

input : feasible solution $\underline{x}^{\alpha}$ for LP_{MDS}
output: IP_{MDS}-solution $\underline{x}_{DS}$ (dom. set)

1 calculate $\delta_i^{(2)}$ // Each step is computed for all vertices $v_i \in V$ simultaneously.
2 $p_i := min\{1, x_i^{\alpha} \cdot ln(\delta_i^{(2)} + 1)\}$
3 $x_{DS,i} := \begin{cases} 1 & \text{with probability } p_i \\ 0 & \text{otherwise} \end{cases}$
4 **send** $x_{DS,i}$ to all neighbours
5 **if** $x_{DS,j} = 0$ *for all* $j \in N_i$ **then**
6 | $x_{DS,i} := 1$
7 **end**

neighbouring colour values, which are then updated according to the x values of neighbouring vertices v_i.

These two algorithms work under the assumption that all vertices have knowledge of the maximum degree Δ. There is a third algorithm available in [11], which describes an adaptation of the LP_{MDS} approximation in which this knowledge is not assumed. However, for the scope of this research, only the first two algorithms are implemented. For a more detailed explanation of the workings of the Kuhn-Wattenhofer algorithm, proofs, and the third algorithm, we refer to the original paper [11].

Algorithm 2: Kuhn-Wattenhofer - LP_{MDS} approximation

1 $x_i := 0$;
2 $\tilde{\delta}(v_i) := \delta_i + 1$
3 **for** $l :=$ *k-1 to 0 by -1* **do**
4 | **for** $m :=$ *k-1 to 0 by -1* **do**
5 | | **if** $\tilde{\delta}(v_i) \geq (\Delta + 1)^{l/k}$ **then**
6 | | | $x_i := max\{x_i, \frac{1}{(\Delta+1)^{m/k}}\}$
7 | | **end**
8 | | **Send** $colour_i$ to all neighbours
9 | | $\tilde{\delta}(v_i) :=| \{j \in N_i \mid colour_j = \text{'white'}\} |$
10 | | **Send** x_i to all neighbours
11 | | **if** $\Sigma_{j \in N_i} x_j \geq 1$ **then**
12 | | | $colour_i :=$ 'gray'
13 | | **end**
14 | **end**
15 **end**

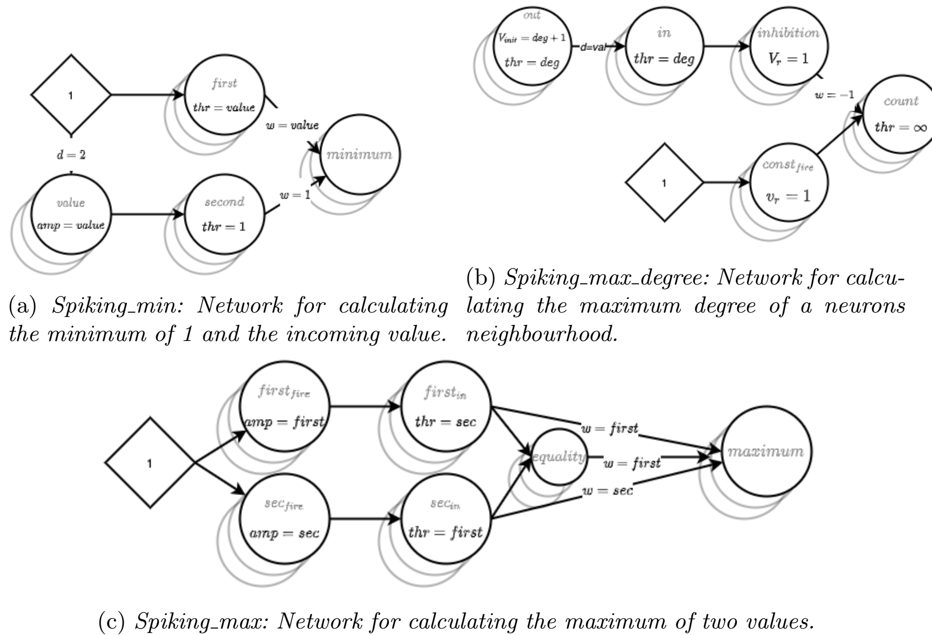
3 Spiking Implementation

In this section, we present the spiking implementation of the Kuhn-Wattenhofer algorithm and the details of the various functions that enable the calculation. The original algorithm can be viewed in section 2.4. The spiking implementation consists of multiple spiking neural networks, that each handle specific calculations and implement different functionalities. Some of these functions are called multiple times, such as the calculation of the degree of the neurons. Programming the SNN is achieved by setting and defining the variables of the neurons, such as their thresholds, delays and spiking amplitude, and the weights of the synapses. The values are defined by e.g. information acquired from the input graph structure, and several networks take the measured resulting voltage of another network as input.

3.1 LP-relaxation

The spiking implementation of the LP-relaxation (alg. 1) consists of six spiking networks. The first function, *spiking_degree*, calculates the degree of each neuron. This is done by creating neurons for all vertices and bidirectional synapses for all edges. All neurons spike once, and the resulting voltage of each neuron then represents the degree of that neuron. Then, *spiking_max_degree*, calculates the maximum value of each neurons neighbourhood. The constructed network of this function can be seen in fig. 3b. It is implemented by creating an *out* and *in*-neuron for each neuron, where for each edge, a synapse is created between the *out*-neuron and the *in*-neuron. The *out*-neurons spike once, with a delay equal to their value. The *in*-neurons have a threshold equal to their degree and no leakage, which ensures that they spike when the last spike has arrived. The spike-timing of the *out*-neuron therefore represents the maximum value of the neighbourhood. This is then converted to a voltage-representation using a separate *count* neuron, which adds one to its voltage until the *in*-neuron fires.

Afterwards, *spiking_multiplication*, calculates an element-wise multiplication of two arrays. This is implemented using one synapse per element, where the initial spike amplitude represents the first value and the synaptic weight represents the second value. The voltage of the post-synaptic neurons then represents the result of the multiplication. Then, the *spiking_minimum* network calculates the element-wise minimum of an element in an array and 1. The constructed network of this function can be seen in fig. 3a. The minimum is calculated using the following function: $((1 > value) \cdot value) + ((value > 1) \cdot 1)$. The neuron *first* handles the first condition $(1 > value)$ by receiving an input spike of amplitude 1 and having a threshold equal to the provided value. The neuron *second* handles the second condition $(value > 1)$ by receiving an input spike of amplitude equal to the value and having a threshold of 1. Both neurons are connected to a final neuron *minimum*, with a synaptic weight equal to *value* for the first neuron and 1 for the second. The voltage of this last neuron then represents the minimum value.



(a) *Spiking_min: Network for calculating the minimum of 1 and the incoming value.*

(b) *Spiking_max_degree: Network for calculating the maximum degree of a neurons neighbourhood.*

(c) *Spiking_max: Network for calculating the maximum of two values.*

Fig. 3: *Various spiking neural networks used as modules in the spiking implementation of the Kuhn-Wattenhofer algorithm.*

The fifth function, *spiking_sampling*, samples according to the given probabilities. This is implemented by creating a stochastic neuron for every vertex, which spikes with the given probabilities. The spike represents whether a neuron is considered in the dominating set or not.

And lastly, the *spiking_summation* network checks for all neurons whether one of their neighbours is in DS, and adds the neuron v_i to DS if this is not the case. This is accomplished by creating neurons for all vertices, and bidirectional synapses for all edges. Each dominating set vertex is represented as a programmed neuron that is constantly spiking. All other vertices are represented as LIF-neurons with a threshold of 1 and a constant input voltage of 1, which initiates them to spike constantly. The weight of the synapse is negative (-1) if the presynaptic neuron is in the dominating set, and 0 otherwise. This way, the LIF-neurons that have a neighbour in the dominating set will be inhibited, while the programmed neurons always keep spiking. The spikes thus represent whether a vertex is considered to be in the dominating set or not.

3.2 Approximating LP_{MDS}

The spiking implementation of the LP_{MDS} approximation (alg. 2) consists of three different spiking networks, of which one is the network utilised in the LP-relaxation to calculate the degree. The second is the *spiking_update* function,

which is depicted in fig. 4 and consists of three main steps. The first step is the updating of the x-values. The *check_dd* neuron checks if the first if-statement is met, by setting its threshold to $(\Delta + 1)^{(l/k)}$ and setting the initial voltage to the old dynamic degree ($\tilde{\delta}(v_i)$) values. If this neuron spikes, the x-value is updated. The calculation of the new x-value is done using the *spiking_max* function, which will be described below. The computed x-values are contained in the x neurons and sent to the *check color* neuron. This neuron checks if the second if-statement is met. If this neuron fires, a silencer neuron is activated, which turns the *color* neuron grey by inhibiting it. The dynamic degree is updated by adding the outputs of these *color* neurons. Because the dynamic degree needs to be updated before the *color* neurons are updated, we read out the dynamic degree after step 2 of the simulation. The x and *color* values are saved after three more simulation steps. As indicated in the figure, all of these neurons are created once for every vertex in the input graph.

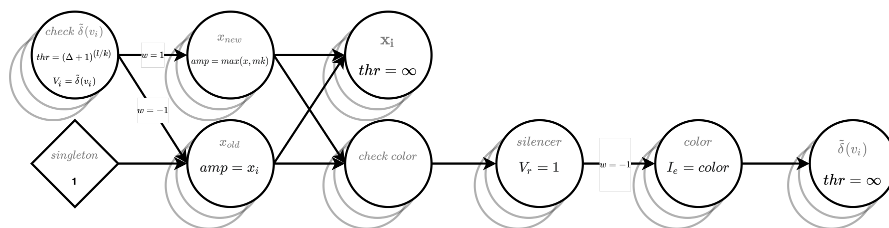


Fig. 4: *Graphical representation of the update function in the approximation of LP_{MDS}. The input values to this function are the previous values of x, color and $\tilde{\delta}(v_i)$ (dynamic degree). These are used in the model as initial voltage V_i, amplitude and input current. The output consists of the voltage levels of the x, color and $\tilde{\delta}(v_i)$ neurons, which are used in the LP-relaxation.*

The *spiking_max* network used in the update function calculates the maximum of two values. The network is depicted in fig. 3c. The maximum is calculated using the following formula: $((a > b) \cdot a) + ((b > a) \cdot b)$, and is thus implemented in a similar manner as the *spiking_minimum* network. One addition is made to ensure a correct computation if both values are equal. In that case, the *maximum* neuron spikes and is reset to 0. The *equality* neuron checks if the inputs are equal and sets the *maximum* neuron to the first value.

These functions contain some commonly used principles, such as the implementation of if-else statements using the threshold value of the neurons. For an example, see the descriptions of the *spiking_minimum* and *spiking_maximum* functions.

However, there are a few functions that have not been converted to the SNN, such as the calculation of the natural logarithm. An attempt was made to perform this calculation with a Taylor approximation in an SNN, but this lead

to the need of more standard Python or package functions. Future work may allow for a complete conversion.

3.3 Neuromorphic Implementation

The translation of the Kuhn-Wattenhofer algorithm has been implemented in the SNN Simulator [19], which simulates a spiking neural network. Additionally, the functions for the calculation of the degree of a neuron and the maximum of two numbers have been implemented in the LAVA framework by Intel [6]. In future work, these functions may be ran on the Intel Loihi neuromorphic chip. Code and documentation can be found at [3].

4 Complexity Analysis

In this section we compare the complexity of our implementation with that of the Kuhn-Wattenhofer algorithm.

Traditional computational complexity analysis observes time complexity in terms of the number of operations that are performed and space complexity as amount of utilised memory. As the Kuhn-Wattenhofer algorithm is a distributed algorithm, our complexity analysis of their algorithm follows the measures as defined by Kshemkalyani and Singhal [10]. According to this metric, time and space complexity are computed both per vertex and system-wide. Additionally, the message complexity is computed in terms of message size, number of messages and number of communication rounds.

Because of their novel structure, spiking neural networks require new measures of complexity for their neuromorphic computation [12]. For SNNs, the time complexity can be measured as time to convergence, the space complexity as network size, and energy complexity as the total number of spikes, according to Kwisthout and Donselaar [12]. Because our unconventional implementation makes use of voltage-based computation, a hybrid complexity analysis is performed. The creation of the network is not performed by the network itself, therefore we also separate the network generation from the simulation of the network in this analysis.

4.1 Space Complexity

Space complexity is defined as the amount of memory that is needed for a computation, apart from the input [17]. However, for spiking neural networks it is defined as network size [12]. For this project, we report on both. As we also make use of non-spiking functions, providing the standard space complexity may provide a more accurate picture. If implemented on neuromorphic hardware, the separation between these two measurements may be more relevant, as a choice can be made to make use of an oracle to compute certain functions at times.

Space Complexity of the Spiking Neural Network The space complexity analysis has indicated that $\mathcal{O}(n^2)$ space is needed for the generation of the spiking networks of both the LP-relaxation (alg. 1) and the LP_{MDS}-approximation (alg. 2). The complexity of the SNN implementation depends on the creation of neurons (n) and synapses (n^2). The execution of the SNNs does not take up any space in addition to their creation, as the execution only changes voltage values within the network. Thus, execution has a space complexity of $\mathcal{O}(1)$. The complete program, including SNN generation, execution, and any necessary non-spiking function has a space complexity of $\mathcal{O}(n^2)$ as well.

Space Complexity of the Kuhn-Wattenhofer Algorithm Per vertex, the space complexity of both Kuhn-Wattenhofer algorithms is $\mathcal{O}(1)$, because each vertex stores the dynamic degree ($\tilde{\delta}(v_i)$), *color* and α-approximation (x_i) which requires 3 memory slots per vertex. resulting in a system-wide space complexity of $\mathcal{O}(n)$. The space complexity of the messages is defined by the message size of $\mathcal{O}(log(n))$ and the amount of messages of $\mathcal{O}(n^2)$.

4.2 Time Complexity

The time complexity of an algorithm is traditionally computed as the amount of atomic computational steps needed in relation to the size of the input. In our case, the input consists of a graph and the maximum degree of the graph. This means that we will express the time complexity as a function of vertices in the input graph.

Since we are building and running a discrete time spiking neural network, we have only calculated the time complexity for the building of the network in the way described above. For the execution of the discrete SNN, we assumed that every time step has a constant time complexity, which is reasonable, if the network is run on neuromorphic hardware. Therefore, the amount of steps the networks have to run determine their time complexity. The time to convergence, as suggested by Kwisthout and Donselaar [12], may be more appropriate for decision problems than for the goals of this research.

Time Complexity of the Spiking Neural Network The time complexity of generating the SNNs used in the LP-relaxation is $\mathcal{O}(n^2)$. The main contributor in this time complexity is the calculation of the δ_i and Δ, which both require synapses between neurons (in both directions) for every edge in graph G. The generation of the SNNs used in the LP_{MDS}-approximation has a time complexity of $\mathcal{O}(k^2 \cdot n^2)$. The bottleneck in this generation is formed by the two for-loops and the update function that they contain, in which the dynamic degree $\tilde{\delta}(v_i)$ is calculated, which requires bidirectional edges.

The execution of LP-relaxation has a time complexity of $\mathcal{O}(n)$, which is due to the calculation of $\delta^{(1)}$ and $\delta^{(2)}$, which require Δ time steps. The upper bound and worst case scenario of Δ, the maximum degree of all neurons, here is $\mathcal{O}(n)$.

Execution of the LP_{MDS} approximation has a time complexity of $\mathcal{O}(k^2)$, due to the two for-loops. The final time complexity of the execution is thus $\mathcal{O}(n + k^2)$.

The complete program, including SNN generation, execution, and any necessary non-spiking function, has a time complexity of $\mathcal{O}(k^2 \cdot n^2)$ due to the construction of the SNNs.

Time complexity of the Kuhn-Wattenhofer algorithm Per vertex, the time complexity of the LP-relaxation is $\mathcal{O}(n)$. Computation of $\delta^{(2)}$ and x_{DS} are the main contributors to this complexity. Per vertex, the time complexity of LP_{MDS} approximation is $\mathcal{O}(k^2 \cdot n)$, due to the two for-loops and the computation of the dynamic degree within them. The time complexity of the messages is defined by the number of communication rounds of $\mathcal{O}(k^2)$. Note that Kuhn and Wattenhofer only mention the message time complexity of $\mathcal{O}(k^2)$ in their paper, constituting to their claim of a constant-time algorithm. However, we argue that because the system-wide time complexity is not constant in time, this claim is invalid.

4.3 Energy Complexity

Energy complexity is an uncommon, widely debated and yet undefined complexity measure for traditional computation paradigms. It can be analysed as a weighted time complexity [15], but it can also be derived from the IO complexity [18].

The advantages of neuromorphic computing are primarily reflected by the relatively low energy consumption in comparison with von Neumann architectures. This has motivated the introduction of energy as a new complexity measure, next to time and space complexity [12]. Whereas the energy complexity of a traditional system is usually directly related to its time and space complexity, this is not per se the case for neuromorphic systems. Depending on the type of encoding (voltage, rate or temporal), the spiking behaviour of an SNN allows for sparser information representation. Assuming a binary encoding, the time between two spikes can be interpreted as a number, which only requires energy when the neurons fire. This means that the size of the number (voltage) does not impact the energy complexity of the representation in such an SNN.

The energy complexity in spiking neural networks is measured by the number of spikes, which assumes that they are discrete events of the same value, independent of actual spike amplitude [12]. Under the assumption that spikes are discrete singular events that can happen once per time step, $energy \leq time \cdot space$. Because we do not use a fully spike-based algorithm, but also inspect voltages at times to output and programmed neurons, the assumption that every spike has an energy complexity of $\mathcal{O}(1)$, does not hold. Therefore, we analyse energy both in terms of discrete spikes, and the synaptic currents to give a more exact picture of the energy usage. Both measures of energy are experimentally measured, while the synaptic current is also theoretically computed in terms of the size of the input.

Energy Complexity of the Spiking Neural Network The creation of the SNNs costs energy, given that the SNN consists of neurons with a non-zero initial voltage. This initialisation energy has a complexity of $\mathcal{O}(k^2 \cdot n^2)$. The biggest contributor here is the creation of the network of the update function, wherein n neurons are created that check the $\tilde{\delta}(v_i)$, with initial voltage bound by n. The dependency on k is achieved since the update function initialises a network and is called k^2 times.

The execution of the SNNs has an energy complexity of $\mathcal{O}(k^2 \cdot n^2)$. This is based on the notion that in a fully connected graph, spikes can travel from all neurons to all other neurons, resulting in an energy complexity of $\mathcal{O}(n^2)$. As the LP_{MDS}-approximation performs the update function inside two nested for-loops, the complexity of this algorithm is increased by a factor $\mathcal{O}(k^2)$.

The energy complexity in terms of spikes in the networks is dependent on $\mathcal{O}(k^2 \cdot n + n^2)$. This stems from the fact that we have n spiking neurons in the LP_{MDS}-approximation, but also n spiking neurons in the function in which *Delta* is calculated, with time complexity $\mathcal{O}(n)$.

For the non-spiking functions, we use their time complexity as an approximation method for their energy complexity, where we assume that at each time-step only one computation takes place and all computations cost equal amounts of energy. Under that assumption, the complexity of the complete program, including SNN generation, execution and any necessary non-spiking functions, remains $\mathcal{O}(k^2 \cdot n^2)$.

Energy Complexity of the Kuhn-Wattenhofer Algorithm The energy analysis for the energy used by the Kuhn-Wattenhofer algorithms, for which we again assume that time complexity is a bound for the energy consumption, yields $\mathcal{O}(n)$ and $\mathcal{O}(k^2 \cdot n)$ respectively for LP-relaxation and the LP_{MDS}-approximation. For the energy complexity of the messages, we have used the same assumption, yielding a complexity of $\mathcal{O}(k^2)$.

5 Discussion

We have shown that the distributed algorithm for finding an approximation of the minimum dominating set as presented by Kuhn and Wattenhofer [11] can be successfully implemented in a programmed spiking neural network. This work serves as an example for the porting of distributed algorithms to spiking neural networks and provides subnetworks that can be modularly used in other algorithms.

Complexity analysis shows that the SNN implementation fares worse in terms of time and energy complexity. However, regarding space complexity, the SNN implementation compares favourably to Kuhn and Wattenhofer. The time and energy costs of the initialisation of the spiking neural networks is largely responsible for these seemingly contradicting findings. It is to be noted that Kuhn and Wattenhofer do not take the initialisation of the message-passing system into account. Including the complexity induced by the initialisation in the time

complexity of Kuhn and Wattenhofer, results in a complexity of $\mathcal{O}(k^2 \cdot n + n^2)$. The theoretical time complexity of the algorithm is thus lower compared to the time complexity of the SNN implementation of $\mathcal{O}(k^2 \cdot n^2)$.

Making use of the inherent distributiveness of neural networks may contribute to the field of distributed computing, as the network can be seen as a population of computational units within one device. Neuromorphic architectures may in the future be used in distributed computing applications such as wireless (sensor) networks.

Future research may look into reducing the complexity further to render the time-, space- and energy complexities of the presented SNN implementation on par with the Kuhn-Wattenhofer algorithm. This may be achieved by making full use of the inherent properties of neuromorphic architectures. Another avenue is to efficiently integrate all subnetworks (functions) into one connected network. While the modularity of our implementation is advantageous in that its modules can easily be reused in various kinds of graphs algorithms, particular problems may benefit from one well-tailored network that is not divisible into modules. Lastly, the spiking neural network may be run on neuromorphic hardware

6 Conclusion

This work presents a novel neuromorphic implementation of the distributed minimum dominating set approximation algorithm by Kuhn and Wattenhofer. By programming the network and utilising voltage-based computation within neurons, the LP-relaxation and LP_{MDS}-approximation algorithms as presented by Kuhn and Wattenhofer have been successfully reproduced. The spiking neural networks are simulated in the SNN Simulator [19]. Several spiking neural networks that have been developed in the translation process can function as building blocks for spiking neural network implementations of other (graph) algorithms.

Measuring the time, space and energy complexity of the spiking implementation, we find that it is comparable to the original algorithm. However, the initialisation of the network takes up significant time and energy. As the complexity of the original Kuhn-Wattenhofer algorithm does not take the initialisation of the message-passing structure into account, we conclude that the spiking implementation does not fare significantly worse.

In conclusion, this work demonstrates that programming a spiking neural network is an avenue worth pursuing for scientific computing applications. Furthermore, it shows how leveraging neural properties in the domain of designing spiking implementations of graph problems, prospers on novel insights. Therefore, our work contributes to the scientific body of knowledge of neuromorphic implementations in the field of distributed computing.

References

1. Aimone, J.B., Parekh, O., Phillips, C.A., Pinar, A., Severa, W., Xu, H.: Dynamic programming with spiking neural computing. In: Proceedings of the International

Conference on Neuromorphic Systems. pp. 1–9 (2019)
2. Ali, A., Kwisthout, J.: A spiking neural algorithm for the network flow problem (2019)
3. Bosch, V., Diehl, A., Smits, D., Toeter, A.: SNN implementation of dominating set approximation. https://github.com/a-t-0/spiking-neural-network-of-dominating-set-approximation (2021). https://doi.org/10.5281/zenodo.5496091
4. Davies, M., Wild, A., Orchard, G., Sandamirskaya, Y., Guerra, G.A.F., Joshi, P., Plank, P., Risbud, S.R.: Advancing neuromorphic computing with loihi: A survey of results and outlook. Proceedings of the IEEE **109**(5), 911–934 (2021). https://doi.org/10.1109/JPROC.2021.3067593
5. Garey, M.R., Johnson, D.S.: Computers and intractability, vol. 174. freeman San Francisco (1979)
6. Intel: Lava: A software framework for neuromorphic computing. https://github.com/lava-nc, https://github.com/lava-nc
7. Kann, V.: On the approximability of NP-complete optimization problems. Ph.D. thesis, Citeseer (1992)
8. Karbasi, A.H., Atani, R.E.: Application of dominating sets in wireless sensor networks. Int. J. Secur. Its Appl **7**, 185–202 (2013)
9. Kay, B., Date, P., Schuman, C.: Neuromorphic graph algorithms: Extracting longest shortest paths and minimum spanning trees. In: Proceedings of the Neuro-Inspired Computational Elements Workshop. NICE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3381755.3381762, https://doi.org/10.1145/3381755.3381762
10. Kshemkalyani, A.D., Singhal, M.: Distributed computing: principles, algorithms, and systems. Cambridge University Press (2011)
11. Kuhn, F., Wattenhofer, R.: Constant-time distributed dominating set approximation. Distributed Computing **17**(4), 303–310 (2005)
12. Kwisthout, J., Donselaar, N.: On the computational power and complexity of spiking neural networks. In: Proceedings of the Neuro-Inspired Computational Elements Workshop. NICE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3381755.3381760, https://doi.org/10.1145/3381755.3381760
13. Li, J., Potru, R., Shahrokhi, F.: A performance study of some approximation algorithms for computing a small dominating set in a graph. Algorithms **13**(12), 339 (2020)
14. Martí, D., Rigotti, M., Seok, M., Fusi, S.: Energy-efficient neuromorphic classifiers. Neural Computation **28**(10), 2011–2044 (10 2016). https://doi.org/10.1162/neco_a_00882, https://doi.org/10.1162/NECO_a_00882, doi: 10.1162/NECO_a_00882
15. Roy, S., Rudra, A., Verma, A.: An energy complexity model for algorithms. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science. p. 283–304. ITCS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2422436.2422470, https://doi.org/10.1145/2422436.2422470
16. Severa, W., Parekh, O., Carlson, K.D., James, C.D., Aimone, J.B.: Spiking network algorithms for scientific computing. In: 2016 IEEE international conference on rebooting computing (ICRC). pp. 1–8. IEEE (2016)
17. Sipser, M.: Introduction to the Theory of Computation. Cengage Learning (01 2012), https://books.google.nl/books?id=1aMKAAAAQBAJ

18. Tran, V.N., Ha, P.H.: Ice: A general and validated energy complexity model for multithreaded algorithms. In: 2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS). pp. 1041–1048 (2016). https://doi.org/10.1109/icpads.2016.0138
19. University, R.: Radboud SNN simulator. https://gitlab.socsci.ru.nl/snnsimulator/simsnn, https://gitlab.socsci.ru.nl/snnsimulator/simsnn
20. Xu, Y.Z., Zhou, H.J.: Generalized minimum dominating set and application in automatic text summarization. In: Journal of Physics: Conference Series. vol. 699, p. 012014. IOP Publishing (2016)
21. Yakopcic, C., Rahman, N., Atahary, T., Taha, T.M., Douglass, S.: Solving constraint satisfaction problems using the loihi spiking neuromorphic processor. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1079–1084 (2020). https://doi.org/10.23919/DATE48585.2020.9116227