Dynamic Programming with Spiking Neural Computing

James B. Aimone
Sandia National Laboratories
Albuquerque, NM, USA
jbaimon@sandia.gov

Ojas Parekh
Sandia National Laboratories
Albuquerque, NM, USA
odparek@sandia.gov

Cynthia A. Phillips
Sandia National Laboratories
Albuquerque, NM, USA
caphill@sandia.gov

Ali Pinar
Sandia National Laboratories
Livermore, CA, USA
apinar@sandia.gov

William Severa
Sandia National Laboratories
Albuquerque, NM, USA
wmsever@sandia.gov

Helen Xu
Massachusetts Institute of Technology
Cambridge, MA, USA
hjxu@mit.edu

ABSTRACT

With the advent of large-scale neuromorphic platforms, we seek to better understand the applications of neuromorphic computing to more general-purpose computing domains. Graph analysis problems have grown increasingly relevant in the wake of readily available massive data. We demonstrate that a broad class of combinatorial and graph problems known as dynamic programs enjoy simple and efficient neuromorphic implementations, by developing a general technique to convert dynamic programs to spiking neuromorphic algorithms. Dynamic programs have been studied for over 50 years and have dozens of applications across many fields.

ACM Reference Format:
James B. Aimone, Ojas Parekh, Cynthia A. Phillips, Ali Pinar, William Severa, and Helen Xu. 2019. Dynamic Programming with Spiking Neural Computing. In ,. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3354265.3354285

1 INTRODUCTION

Graph algorithms continue to play a large role in high performance and data-center computing, but existing parallel designs may not be well-suited for the eccentricities of large-scale graph problems [16]. Furthermore, this difficulty is compounded by the increasing global power costs of computation [3, 10]. And while current efforts have brought high-performance graph analytics to graphics processing units (GPUs) [19, 31], these efforts are limited in scope, and GPUs still carry a substantial power budget. For large-scale systems and graphs, these methods alone may be not be sufficient in an increasingly energy-limited application. This motivates the need to explore graph algorithms on additional exotic computing platforms, in particular neural-inspired or neuromorphic computer architectures.

Recently, there has been a massive resurgence in interest in neuromorphic computing architectures [22]. These architectures take structural inspiration from biological neural systems, and their popularity has largely tracked alongside the growth of deep learning artificial neural network use [15]. The proimse is that neuromorphic platforms, such as IBM's TrueNorth [17], Intel's Loihi [5], and SpiNNaker [7], can act as a potential future-computing platform as we approach limits on traditional vonNeumann processing [13, 26, 28]. We now see several methods of achieving state-of-the-art in performance/Watt for deep learning tasks on neuromorphic platforms [21, 25], but we instead focus on a growing interest in using neuromorphic systems for numerical or otherwise direct computation. While unconventional, this method of treating neurons as computational elements in their own right has seen the development of a number of algorithms in applications such as arithmetic, image processing, diffusion equations, and, of course, graph algorithms [2, 12, 14, 18, 20, 23, 24, 27].

We remark, however, that little work has been done to characterize the families of algorithms suitable for efficient neuromorphic implementation. This is perhaps surprising given benefits of neuromorphic systems are readily available, including

(1) High levels of parallelism; Potentially asynchronous execution,
(2) High fan-in/fan-out of the node,
(3) Co-located processing and memory.

Furthermore, spiking neural architectures (the subclass of neuromorphic architectures with which we concern ourselves; details below) share many similarities with well-characterized threshold gate circuits [8]. These two facts combined suggest that a formal exploration of neuromorphic graph algorithms may be both possible and fruitful, and we begin that exploration by recognizing (rather than an individual algorithm) a large class of suitable programs.

The goal of this paper is to help fill this characterization by demonstrating that a broad class of dynamic programs enjoy efficient and relatively simple neuromorphic implementations. Dynamic programming is a general method for expressing and efficiently solving certain kinds of combinatorial and graph problems. Dynamic programming addresses a range of applications in linear algebra, economics, bioinformatics, databases, computer graphics, and machine learning [4, 29]. We give a general method for converting certain kinds of dynamic programs to neuromorphic algorithms. We also demonstrate that more sophisticated dynamic programs may be implemented neuromorphically by doing so for the longest increasing subsequence problem.

ICONS '19, Intl. Conf. Neuromorphic Systems, July 23–25, 2019
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/3354265.3354285

The remainder of the paper is organized as follows. In Section 2, we detail the spiking neuron model and connectivity model we use to develop and analyze the algorithms. In Section 3, we illustrate the concept of a spiking graph algorithm through the problem of computing shortest paths in an edge-weighted graph. In Section 4, we introduce dynamic programming and described a generic spiking framework to solve some classes of dynamic programs. Section 5 provides a concluding discussion and outlook for future work.

2 BASIC COMPONENTS OF NEURAL ALGORITHMS

2.1 Spiking Neural Architectures

The neural architectures we are concerned with in this paper are specifically *spiking neural architectures*. While there are different implementations of these platforms, we focus on a discrete *leaky-integrate and fire* or *LIF* description of a neuron that is common to spiking neuromorphic hardware and can be considered as entirely parallel in neural operations.

The basic LIF dynamics are as follows. A neuron starts with a voltage of $v = v_{\text{reset}}$. At each time step, the voltage, v of a neuron is updated with a decay from the previous time-step and a voltage change due synaptic inputs (Eq. 1). In continuous time, this would be an exponential decay, but in discrete time implementations, this amounts to a fractional step of the v towards v_{reset} of a magnitude, τ. (Note: if $\tau = 1$, this model becomes equivalent to a threshold gate circuit). After the voltage update, v is compared to a threshold voltage, $v_{\text{threshold}}$ to determine if the neuron spikes ($f(t+1) = 1$ in Eq. 2, and if so, the voltage resets according to Eq. 3). Finally, the synaptic inputs for the neuron are computed by summing all of the weights w_i of active ($f_i(t - d_i) = 1$) upstream neurons, offset by an appropriate delay, d_i, which represents the delay for that spike to arrive from neuron i (Eq. 4). Furthermore, if learning is available and used, the weights can be themselves considered functions of time $w(t)$.

$$\hat{v}(t+1) = v(t) - (v(t) - v_{\text{reset}}) \cdot \tau + v_{\text{syn}}(t) \qquad (1)$$

$$f(t+1) = \begin{cases} 1, & \text{if } \hat{v}(t+1) > v_{\text{threshold}} \\ 0, & \text{if } \hat{v}(t+1) \leq v_{\text{threshold}} \end{cases} \qquad (2)$$

$$\text{if } f(t+1) = 1 \text{ then } v(t+1) = v_{\text{reset}} \text{ else } v(t+1) = \hat{v}(t+1) \qquad (3)$$

$$v_{\text{syn}}(t+1) = \sum_{i=1}^{N} (f_i(t+1-d_i) \cdot w_i) \qquad (4)$$

This LIF model is a common abstraction of biological neuron dynamics, and it represents a common target for proposed neuromorphic hardware. While mathematically compact, it also has useful similarity to even simpler neural computing models such as threshold gates (*TGs*), which have been the focus of more extensive theoretical analysis, while also representing the key feature of energy-efficient communication—the limitation of communication in architectures to cases where $f(t) = 1$, which is commonly referred to as a *spike*. In this case we say that a neuron has spiked or fired at time t. Indeed, the promise of spiking neuromorphic hardware in large part can be attributed to this event-driven communication and the unbounded fan-in / fan-out of TG circuits.

While even these simple LIF neurons provide some configurability, spiking neural algorithms are primarily defined by the existence of synapses between different neurons. Each synapse defines a directed connection between a pair of neurons i and j, with independently programmable weight $w_{i,j}$ and delay $d_{i,j}$. The neurons and synaptic connections are collectively referred to as a spiking *circuit*. Computation is initiated in such a circuit by the simultaneous spiking of a designated set of *start* nuerons, and execution proceeds for a fixed amount of time or until a designated *terminal* neuron first spikes. The output of the computation is typically the state of the set of *output* neurons at the time of termination.

2.2 Relationship of neural algorithms and graphs

Taking the description above, spiking neural circuits can be thought of as directed, weighted graphs, and can also contain other attributes that pertain to delays, learning, or more complex synaptic dynamics. Neural algorithms are not necessarily acyclic; indeed, cycles if constructed appropriately can be useful for timing and coordination. In addition to small circuits that can be constructed for such control purposes; other neurons are often dedicated to receiving inputs or providing outputs.

The relationship of neural circuits and graphs has been recognized for a number of years, and there has long been work in using neural approaches to address graph problems. Much of this work has focused on artificial neural network learning algorithms to solve problems, such as the Hopfield-Tank model for problems such as Traveling Salesman [11], however this application's value has been debated [30]. More recent research, especially through the successes of deep learning, has focused on learning algorithms, and indeed many problems addressed by modern neural networks can be equated to classic graph problems. For example, a lot of the AlphaZero work is learned graph search.

There has been less, but an increasingly growing, body of work considering the direct implementation of graph problems onto spiking neuromorphic hardware, such as the work of Hamilton et al. [9], that considers spiking implementations of an algorithm by Aibara et al. from 1991 [1]. This recent work focuses on directly implementing a graph within a neural substrate and using delays to represent the original edge weights of the graph. Such a mapping affords a natural implementation of Dijkstra's celebrated algorithm for computing shortest paths in graphs on neuromorphic substrates.

The present work was initiated with an independent rediscovery of Aibara et al.'s implementation of Dijkstra's algorithm and culminates in the observation that this approach can be generalized to solve a plethora of discrete optimization and graph problem by way of dynamic programming. We observe that some of the best-known dynamic-programming approaches are not directly addressed by our general framework, yet in Section 4.3 we demonstrate that spiking algorithms may be attainable in such cases by developing a refined spiking implementation of the best-known dynamic programming approach for the longest increasing subsequence problem. In the next section we employ the aforementioned problem of computing shortest paths to illustrate elements of designing spiking graph algorithms.

3 SPIKING GRAPH ALGORITHMS

In this section we describe the single-source shortest path problem on graphs and describe a natural and elegant spiking graph algorithm to solve it. This algorithm will serve as the basis for our neuromorphic dynamic programming approach and illustrates the basic techniques we will use.

3.1 Shortest paths and Dijkstra's Algorithm

A fundamental problem in graphs is finding a shortest path between two nodes s and t (an s-t path) in a graph that has non-negative weights associated with its edges. Problems of this type are solved routinely by Google Maps in finding short navigation routes between two places on a map. In this case, the nodes of the graph represent locations of interest on a map, while edges indicate viable direct routes, with each weights indicating traversal times.

Dijkstra's algorithm is an elegant approach to finding shortest paths that is typically one of the first graph algorithms taught in algorithms courses [4, Ch. 24]. Some algorithms that are theoretically efficient are not so in practice, due to large constant factors in execution time. Yet, Dijsktra's algorithm affords efficient practical implementations. Such implementations of it can find an s-t path in a graph with n vertices and m edges in $O(m \log n)$ steps, with each step a primitive computational operation [4]. Implementations with an asymptotic running time of $O(m + n \log n)$ are known, but these do not tend to perform as well in practice, due to large constant factors hidden by the $O()$ notation. In fact, Dijkstra's algorithm solves a more general graph problem: that of finding shortest paths between a specific vertex s and every other vertex in the graph (i.e., a shortest s-v path for every node v). This is known as the single-source shortest paths (SSSP) problem.

A reason that an algorithm seeking to find a shortest path between s and t might indeed need to discover shortest paths between s and every other vertex, including t, is as follows. A shortest path, P between s and t must contain a shortest path between s and every vertex, v_i that is part of the path, P. If there were any shorter path, Q between s and some such v_i, then one would have a shorter path between s and t by going from s to v_i using Q and then continuing from v_i to t as P does. This is known as the optimal substructure property, and is a key ingredient in dynamic programming, as will be discussed in the next section. Thus it is difficult to imagine a shortest s-t path algorithm that does not also solve the SSSP problem in the process.

3.2 Spiking graph algorithm for shortest paths

Dijkstra's algorithm inspires a simple and natural spiking graph algorithm (SGA) to solve this problem, first published in 1991 [1]. As discussed in Section 2.1, our model of spiking neural architectures (SNA) includes programmable spike delay times on links. By this we mean that one may, independently for each link, specify how long a spike should take to traverse it. As communication in spiking architectures is typically asynchronous, delays are a powerful resource spiking algorithm design.

The SGA we describe assumes that the nodes and edges of the input graph can be mapped on the neurons and links of the SNA. One way to relax this assumption is by embedding an input graph into the neuron connectivity graph of an SNA; a detailed discussion of such techniques is beyond our scope. We program delay times so that the time required for a spike to traverse a link in the SNA is proportional to the weight of the corresponding graph edge. The SGA commences by sending spikes from a source neuron, corresponding to the vertex s, to all of its outgoing neighbors. Every other SNA neuron is programmed to simply propagate the first incoming spike it receives to all its outgoing neighbors. The SGA terminates when every neuron has received a spike, or it can be programmed to terminate early as soon as the neuron corresponding to a particular node of interest, t has received a spike. This is a complete high-level description of an SGA for the SSSP problem.

Computational complexity of the algorithm. One may view this SGA as employing spikes to explore a graph. Initial spikes are generated at node s and propagated to its neighbors, each of whom then propagates copies of incoming spikes to their neighbors, and so on. When a spike reaches some node v, this corresponds to a specific path that a sequence of spikes, starting at s, followed to reach v. Moreover, due to link delays, the time when the spike reaches v is proportional to the length of this path. Thus the time v receives its first spike is proportional to the length of the shortest path from s to v. This argument can be made precise to show that correctness of this SGA, in much the same vein as one may prove the correctness of Dijkstra's algorithm.

The running time of the algorithm is $O(L + n + m)$, where L is the length of the shortest path from s to t, and n and m are the number of nodes and edges in the input graph, G, respectively. We include the $O(n + m)$ term to account for the time required to load G into the SNA, which involves identifying the nodes and edges of G to neurons and links present in the SNA. This factor also accounts for the readout time of a solution. Thus far we have only discussed computing a number that is proportional to the length of a shortest path; in practice, we also seek to output a shortest path. We will discuss techniques for doing so in the next section.

How does this compare with Dijkstra's algorithm running on a conventional computer? The SGA may be viewed as a Dijkstra-like algorithm; however, it is far simpler than conventional pseudocode for Dijkstra's algorithm. More concretely, let τ_v be the time when the neuron v first spikes in the SGA. Dijkstra's algorithm also computes quantities proportional to τ_v; however it does this indirectly using a data structure called a priority queue. This allows Dijkstra's algorithm to avoid having to simulate sending spikes; however, building and maintaining the priority queue incurs extra overhead.

As stated above, a conventional practical implementation of Dijkstra's algorithm requires $O(n \log m)$ time. The SGA offers an advantage when one seeks to find relatively short paths in large graphs, which is the case for practical applications such as computer-assisted navigation. However, another advantage is that once a graph is loaded onto an SNA, one may modify some link delays and execute the SGA again without loading the graph again. This can prove advantageous over conventional Dijkstra implementations that, in the worst case, may have to perform $O(n \log m)$ steps to re-run the algorithm, even if only a few edge weights have changed. Another advantage is that the SGA is likely to incur less computational overhead during execution than a conventional algorithm, leading to practical advantages in execution time and energy expenditure, even when asymptotic analysis may suggest otherwise.

3.3 Constructing a shortest path

In order to compute actual shortest paths within the SGA, for each neuron v that spikes, we must keep track of the neuron u that caused v to first spike, at time τ_v (see the discussion of shortest path trees in [4, Ch. 24] for why this suffices). We sketch a scheme for accomplishing this. The first requires a total of $O(n \log n)$ neurons and $O(m \log n)$ links. The modified SGA is structured similarly to the original, except that each neuron u is represented with neurons $u_0, u_1, \ldots, u_k$ where $k = \lfloor \log n \rfloor + 1$. For each edge uv in G we include a link from each u_i to v_i, for $i = 0, \ldots, k$, whose delays are all set to the same value, proportional to the weight of uv. We assume that each node has an integer id between 1 and n and that s. The neuron v_0 takes the place of neuron v from the original SGA, and the population of neurons $V = \{v_1, \ldots, v_k\}$ acts a memory, recording a binary encoding of the id of the neuron u_0 that sent the spike causing v_0 to spike for the first time. To illustrate, suppose $n = 7$, node u has an id of 5, and nodes v and w are its neighbors. Whenever u_0 spikes, it sends a binary encoding of its id on the links from $\{u_1, \ldots, u_k\}$ to $\{v_1, \ldots, v_k\}$ and those from $\{u_1, \ldots, u_k\}$ to $\{w_1, \ldots, w_k\}$. Since the ids are fixed, this is easy to do, by including links with negligible delay from u_0 to u_1 and u_3 (this is because $(1, 0, 1)$ is the binary encoding of 5, represented by u_1 and u_3 firing and u_2 not firing). Finally, for each node u there is additional neuromorphic circuitry that "latches" the values of the population U when u_0 first spikes, so that U represents a binary encoding of the node id that caused it to first spike.

The above approach uses an additional $O(\log n)$ neurons for each of the n nodes of the graph as a memory. We note that other approaches for storing the shortest path information are possible, including using learning on synaptic connections to store this information. Next, we generalize the SGA described here to address solving dynamic programs. Although our treatment of this topic will be at a higher level, the details discussed here for shortest paths will implicitly apply.

4 SPIKING ALGORITHMS FOR DYNAMIC PROGRAMMING

In this section we demonstrate how to use neuromorphic architectures for solving a large class of classical dynamic programs. We begin by describing how dynamic programming uses the "optimal substructure property" and "overlapping subproblems" to recursively solve a problem without recomputing its subproblems. In section 4.1 we identify several main categories of problems that can be solved by dynamic programming and propose neuromorphic solutions. These involve problems where the the optimal solution can be found by taking simple maximums or minimums over input optimal subproblems, perhaps with some linear constraints. Finally, we introduce a neuromorphic circuit for solving the longest increasing subsequence (LIS) problem using dynamic programming. That circuit uses a more complex relationship between input optimal subproblems than those described in Section 4.1. We conclude by showing neuromorphic circuits to find a longest path in a directed acyclic graph, a problem closely related to shortest paths. This gives an alternative solution to LIS that requires classical pre-processing, but then fits the characteristics of the dynamic-programming problems described in Section 4.1.

4.1 An Overview of Dynamic Programming

Dynamic programming is a fundamental algorithmic paradigm for solving combinatorial algorithms in polynomial (efficient) time. Generally, dynamic programming is a useful approach when a problem can be solved optimally by breaking it into sub-problems and then combining optimal solutions to the sub-problems. It has been adopted in many different application domains. A Google scholar search on "dynamic programming" has 3.2 million hits, including general papers on the topic and specific applications.

Two fundamental concepts underlie dynamic programming: the *optimal substructure property* and *overlapping subproblems*. The optimal substructure property applies when an optimal solution to a problem can be expressed in terms of optimal solutions for smaller subproblems. For instance, the shortest path problem illustrates this property. Consider the problem of finding a shortest path from a source vertex s to a terminal vertex t in a graph. Assume we know that a third vertex v is part of an optimal solution. Then we can reduce the problem to two independent problems: finding a shortest path from s to v and finding a shortest path problem from v to t. We can prove that merging optimal solutions to these two problems gives an optimal solution to original problem of finding a shortest path from s to t. Observe that the optimal substructure property does not apply in many other problems. Consider the path version of the traveling salesperson problem, where we try to find a shortest path from s to t that visits each vertex once. This time, we cannot break the problem into two independent subproblems finding s to v and v to t, since it is not clear whether the remaining vertices are part of the first subproblem or the second subproblem.

The optimal substructure property allows us to compute the optimal solution recursively by breaking the problem down into smaller problems. Recursively investigating many solutions leads to many subproblems being investigated multiple times. The trick in dynamic programming is to define this space of overlapping subproblems and avoid solving them more than once. Subsequently, the efficiency of dynamic programming depends on the number of subproblems that need to be solved. In practice, we avoid recursion in implementations of dynamic programming algorithms by maintaining a table that stores values of optimal solutions to subproblems, and filling in the entries of this table in a bottom-up fashion. That is, we compute the solution to a subproblem when all the subproblems it depends on have been solved.

4.2 Neuromorphic Computing for Dynamic Programming

We propose a generalized approach to solving dynamic programming problems using neuromorphic computing. In our approach, each subproblem is represented by a neuron or a group of neurons and neurons are connected if the solution of one is affected by the other. Neurons communicate the optimal value of their respective subproblems by their firing times. As we discuss later, in some cases neurons fire multiple times for maximization problems, as better solutions become available.

With this neural-network structure, each neuron has enough information from optimal subproblems to compute its own solution value. The challenge is in how to compute this value given values of optimal solutions to the subproblems. We describe generic solutions

Name	Symbol
Optimal solution value for parameters x	S_x
Incoming neighbors of node i	$N(i)$
Cost of edge between nodes i and j	ec_{ij}
Cost of node i	nc_i
Reward for edge between nodes i and j	er_{ij}
Reward for node i	nr_i

Table 1: A reference table of the notation used in analyzing neuromorphic dynamic programs.

to a broad class of problems, based on the structure of the recursive formulation in the dynamic programming solution. Specifically, we describe generic solutions for constrained and unconstrained versions of maximization and minimization problems as long as the optimal solution value depends only on the minimum/maximum of subproblems.

At a high level, we define dynamic programming problems in terms of graph constructions and explain how to solve them neuromorphically. In our discussions, we use the notation described in Figure 1. When we describe neuromorphic circuits, we sometimes use the same notation S_x to refer to a circuit node that represents a subproblem.

Case 1a: Minimization problem with fixed costs.
First, we consider minimization problems where each node and edge has a fixed cost that fit the following minimization framework:

$$S_i = \min_{k \in N(i)} \{S_k + ec_{ik}\} + nc_i$$

Example: Shortest Path: The shortest path is a path between two vertices that minimizes the sum of the weights of edges on the path.

Definition 4.1 (Shortest Path). Given a graph $G = (V, E)$, a cost function on edges $w : E \to \mathbb{R}_{\geq 0}$, and two vertices $s, t \in V$. Find a sequence of vertices $\langle v_{\phi_1}, v_{\phi_2}, \ldots v_{\phi_n} \rangle$, such that $s = v_{\phi_1}$, $t = v_{\phi_n}$, and $(v_{\phi_i}, v_{\phi_{i+1}}) \in E$ for $i = 1, \ldots, n-1$ that minimizes

$$\sum_{i=1}^{n-1} w(v_{\phi_i}, v_{\phi_{i+1}})$$

Neuromorphic solution: The same recursive formulation can describe many problems. We describe a construction that only relies on the equations and is independent of the details of the underlying application. Let each subproblem S_i be represented by a neuron and connect each neuron to other neurons if the associated problems are dependent. In the formulation above, the neuron representing S_i receives spikes from all neurons in $N(i)$. Once a neuron receives a spike, it is activated. It waits for nc_i units of time, and then sends signals to each neighbor k with a delay of ec_{ik}. Each neuron is activated only once, and the time of activation defines the optimal solution value. For the shortest-path problem, we have a subproblem (shortest-path to a graph vertex) for each circuit vertex, circuit vertex cost zero, and circuit edge costs correspond to graph edge weights.

Case 1b: Maximization problem with fixed costs.
Next, we consider maximization problems with fixed costs using the following formulation:

$$S_i = \max_{k \in N(i)} \{S_k + er_{ik}\} + nr_i$$

Example: Longest Increasing Subsequence: The longest increasing subsequence problem aims to find a longest subsequence, not necessarily contiguous, within a given sequence in which the subsequence's elements are increasing.

Definition 4.2 (Longest Increasing Sequence). Given a sequence of real numbers $\langle x_1, \ldots x_n \rangle$, find a subsequence $\langle x_{\phi_1}, x_{\phi_2}, \ldots x_{\phi_k} \rangle$ that maximizes k such that for all i, $1 \leq i \leq k$, we have $1 \leq \phi_i < \phi_{i+1} \leq n$ and $x_{\phi_i} < x_{\phi_{i+1}}$.

In Section 4.3 we consider a simpler problem, the original problem definition, which is to just find k, the length of the longest increasing subsequence. In Section 4.4 we show a way to convert the LIS problem into one of the form above, though it requires preprocessing to create the circuit.

Neuromorphic solution: Again, we describe a construction that only relies on the equations and is independent of the details of the underlying application. Let each subproblem S_i be represented by a neuron and connect each neuron to other neurons if the associated problems are dependent. In the formulation above, a neuron representing S_i receives spikes from all neurons in $N(i)$. Once a neuron receives a spike, it is activated. It waits for nr_i units of time, and then sends signals to each neighbor k with a delay of er_{ik}. The construction so far is the same as the construction above. The difference is that the neuron keeps firing for each new spikes it receives, Each new spike corresponds to the identification of a new solution that is better than the previous solution, since this is a maximization problem. And discovery of a better solution should be communicated to the neighbors.

Case 2a: Constraint Satisfaction problem.
We can also solve constraint satisfaction problems with the following format:

$$S_{ij} = \max_{k \in N(i); j - ec_{ik}} \{S_{k, j-ec_{ik}} + ec_{ik}\} \text{ for } j \leq B,$$

where B is a given bound. **Example: Number-partitioning:** In the number-partioning problem, we are given set of positive integers. We wish to partion that set into two subsets to minimize the difference between the sum of the numbers in the two subsets.

Definition 4.3 (Number Partitioning). Given a set of positive integers, $A \subset \mathbb{Z}$, find A_1 and A_2 that form a bipartition of S that minimize

$$\left| \sum_{a_i \in A_1} a_i - \sum_{a_i \in A_2} a_i \right|$$

Although the partition problem is NP-complete, there is a pseudo-polynomial-time dynamic-programming solution. A variant of the number-partitioning problem is to find a subset of a given set of integers whose sum is as close as possible to, but no greater than, a given bound B.

To put the number-partition problem into the above form, let $G = \sum_{a_i \in A} a_i$. Then we wish to find a subset A_1 of maximum sum G_1 provided that sum obeys the constraint $G_1 \leq G/2$.

Neuromorphic solution: The circuit for this problem has a node S_{ij} for all $1 \leq i \leq n$. The j value (at most $G/2$) represents the sum of values from a subset of the first i numbers in set A. The circuit will add elements from A in an arbitrary but fixed order. The start node is S_{00}. Then there is an edge from S_{ij} to all $S_{k,j+a_k}$, for $i < k \leq n$ provided $j + a_k \leq G/2$. This edge represents adding the kth element of A next, skipping elements between the ith and kth. Node S_{ij} fires at time j if there is a subset of the first i elements whose sum is exactly j. The last node to fire gives the solution. The constraint is satisfied by the construction of the circuit. Then the circuit functions by nodes firing to all outgoing neighbors when activated.

Case 2b: Constrained minimization problem with fixed costs
The neuromorphic approach also extends to a specific class of constraint minimization problems with the following cost formulation:

$$S_{ij} = \min_{k \in N(i); j-nc_i \leq B} \{S_{k,j-nc_i} + ec_{ik}\}.$$

Example: Shortest path from node s to node t, whose node cost is at most B:
Neuromorphic solution: This is like the regular shortest-path circuit except that now each circuit node S_{ij} has a second subscript denoting the cost of the path from graph node s to graph node i. Edges from neighbors are constructed to correctly match that cost subscript. So node $S_{k,q}$ is an input to node $S_{i,j}$ if and only if node k is a neighbor of node j in the input graph and $q + nc_i = j$. The neural circuit starts at node S_{s,nc_s}, provided $nc_s \leq B$ and the first time a circuit node associated with graph node t fires represents the shortest constrained path.

Case 2c: Constrained maximization problem with fixed costs.
Problem Formulation:

$$S_{ij} = \max_{k \in N(i); j-ec_{ki}} \{S_{k,j-ec_{ki}} + nc_i\} \text{ for } j \leq B$$

Example: Knapsack: The knapsack problem is: given a set of objects $x_1, \ldots, x_n$. Object x_i has a weight w_i and a value v_i. The goal is to pick a subset of objects X such that $\sum_{x_i \in X} w_i \leq W$, for a knapsack capacity W and set X has maximum value $\sum_{x_i \in X} v_i$.
Neuromorphic solution: In our neural circuit, the last time node S_{ij} activates represents the highest-value subset of the first i items with weight exactly j. As with the partition problem, we consider adding or not adding elements to the knapsack in an arbitrary fixed order: $x_1, x_2, \ldots, x_n$. Neural circuit node S_{ij} represents a subset of the first i items whose total weight is exactly j. As before, we consider the knapsack capacity constraints when constructing the neural circuit. Node $S_{k,q}$ is an input to node $S_{i,j}$ if and only if $i > k$ and $q + w_i = j$. The delay on the edge is v_i. So a time that node $S_{i,j}$ activates is the value of a subset of the first i nodes with weight exactly j. The last activation indicates the optimal solution value.

Other problems that fit this basic form include longest increasing sequence, whose cost is $\leq B$ and finding a longest path (maximizing scenery) provided the arrival time is at most B.

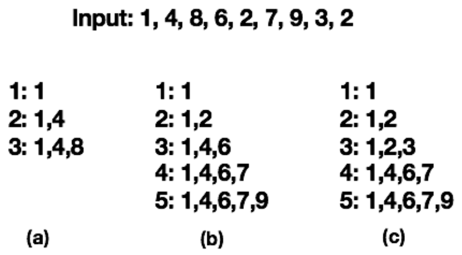
4.3 Longest increasing subsequence

Recall this definition from Section 4.2. The *longest increasing subsequence* problem (LIS) accepts a list of n positive integers $x_1, x_2, \ldots, x_n$ and returns the longest strictly increasing (not necessarily contiguous) subsequence. In Section 4.4, we give a method to transform the input sequence into a directed-acyclic graph (DAG) where the longest path gives the LIS. However, that method might require $\Theta(n^2)$ time to create a graph with $\Theta(n^2)$ edges during a required classical-computing preprocessing step. The circuit in this section uses the sequence input directly and is based on a more efficient classical algorithm.

In this section, we give a circuit for a simpler problem: just computing the length of the LIS. The circuit can mark information for recovering the full sequence, which we do not describe in this paper. Fredman [6] describes an $O(n \log n)$-time solution, which he attributes to Knuth, for a classical computer. This classical algorithm processes the input string left to right. It maintains an array L where the ith entry L_i holds the value of the last element in the best sequence of length i seen so far. The best sequence is the one with the smallest last element since it is the easiest to build upon. In the array, $L_i = \infty$ if there is no increasing subsequence of length i. At the time the algorithm is about to process element x_j, the information in the array L represents the best subsequences possible using only the first $j-1$ elements. After processing element x_j, the array L then represents the best subsequences possible using the first j elements. After the nth element has been processed, the answer is the largest value ℓ such that L_ℓ is finite.

To process element x_j, the algorithm finds the index m such that $L_m < x_j < L_{m+1}$. If $x_j = L_k$ for some k, then index m does not exist, and x_j is processed. Otherwise there is only one such index m. The algorithm sets $L_{m+1} = x_j$, indicating that x_j is the last element of a new best sequence of length $m + 1$. Element x_j can extend the subsequence of length m because it is larger than the last element. It is an improvement, since $x_j < L_{m+1}$. In the example in Figure 1, we show the full subsequence represented by each entry in array L at three points in the algorithm. This example contains additional information not present in L. Our array L contains only the last element of the best subsequence of each length, while Figure 1 depicts the entire best subsequence of each length. Figure 1(b) shows the list after processing the first seven elements, so the next element is 3. Since $2 < 3 < 6$, we create a new best sequence of length three by adding 3 to the best current sequence of length two.

We now describe a neuromorphic circuit to compute the length of an LIS using the above algorithm. We assume zero delay through a node for this discussion. We discuss node delays at the end of this section. The nodes do not leak ($\tau = 0$ in the model of Section 2). The circuit for LIS has a column for each sequence element x_j and a row for each possible sequence length i, as shown in Figure 2. There is a single node, labeled "start" that is active at the beginning of the circuit execution. When it fires, it sends a spike to a local node, "x_j," for each column j with a delay equal to the value of the jth sequence element, x_j and a weight sufficient to trigger node x_j immediately. Figure 3 shows the circuit for a column. In each column j, there are two nodes for each row L_i, called v_{it} (top) and v_{ib} (bottom). The spike from the "x_j" node in Figure 2, which we call the "alarm" for column j, is an input to each of the nodes in column j (these inputs are labeled "A" in Figure 3). The alarm applies a weight of 1 to the top nodes v_{it} and a weight of -1 to the bottom nodes v_{ib}. Alternatively, the local nodes "x_j" may be



Input: 1, 4, 8, 6, 2, 7, 9, 3, 2

(a)	(b)	(c)
1: 1	1: 1	1: 1
2: 1,4	2: 1,2	2: 1,2
3: 1,4,8	3: 1,4,6	3: 1,2,3
	4: 1,4,6,7	4: 1,4,6,7
	5: 1,4,6,7,9	5: 1,4,6,7,9

Figure 1: An example of solving the longest-increasing subsequence problem. Although our neural circuit only tracks the last item for the best sequence of each length, we show the sequences themself for clarity. A list preceeded by "1:" is the best sequence of length 1 so far; "2:" is the best sequence of length 2, etc. (a) shows the best sequences after processing the first 3 elements, going left to right. (b) is after the first 7 elements and (c) is the final set of lists. The longest increasing sequence has length 5.

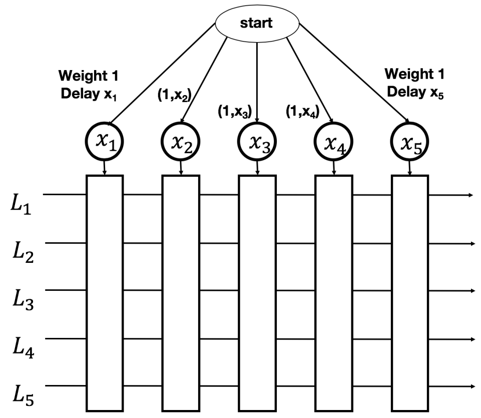


Figure 2: The overall circuit for LIS. Each column corresponds to a string element and row i corresponds to the last element in the best sequence so far of length i. The node labeled "start" is the only one active at the start. It sends a spike to each column j with a weight of 1 and a delay of x_j, which is the value of the jth sequence element. Because column j cannot participate in rows greater than j, we could remove unneeded nodes, giving a triangular circuit, saving about a factor of 2 in circuit size. Given an upper bound on LIS, we can reduce the number of rows.

replaced by direct connections from the "start" node to the nodes in each column, each with appropriate weight.

Consider the processing that occurs in some column j, corresponding to sequence element x_j. The spike coming from the left on row $L_i \in \{L_1, L_2, \ldots, L_n\}$ arrives at time L_i, the value of the last element of the best sequence of length i. Consistently, any $L_i = \infty$ never arrives. The L_i are always in sorted order: $L_1 < L_2 < L_3 < \ldots$. See the example in Figure 1 and observe that the sorted property comes from the way the L_i are updated. So the first signal from the

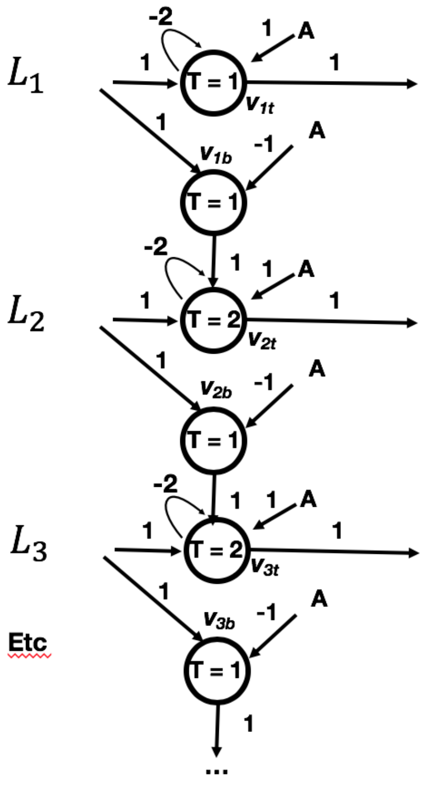


Figure 3: The LIS circuit for one column of the overall circuit associated with a with string element. For column j with string element x_j, "A" represents an alarm that arrives at time x_j.

left arrives on edge L_1. If it arrives before the alarm, then we must have $L_1 < x_j$. In this case, the spike from the left activates node v_{1t}, which fires the spike leaving the column on row L_1 to the right at time L_1. The self edge on node v_{1t} with weight -2 sets the node weight at v_{it} to -1. this prevents node v_{1t} from firing again at time x_j, when the alarm spike arrives. Also, the spike L_1 activates node v_{1b}. This fires an input into the third node v_{2t}, which is the first node for row 2. Effectively, this reduces the threshold of that node to 1 (since nodes have no leakage), so row 2 now acts the same way the first row did. Thus the processing when a spike arrives on line $L_2, L_3, \ldots$ operates in the same manner until the alarm at time x_j occurs before the incoming spike on some line L_i. This includes the case where there is no spike on edge L_i (i.e. edge $L_i = \infty$).

We now focus on the case when the alarm at time x_j occurs before a spike arrives on line L_i, and spikes have already arrived on lines $L_1, \ldots, L_{i-1}$, before time x_j. Then the effective threshold at node v_{it} is 1, since node $v_{i-1,b}$ has already fired. The alarm at x_j with weight 1 meets the threshold on node v_{it}, activating node $v_{i,t}$ and sending a spike rightward on row i at time x_j. The alarm also fires an input with weight -1 into node v_{ib}, setting the node weight to -1. This means that any later signal coming in on row i can not cause node v_{ib} to fire again. The alarm fires a spike with weight -1 into all the nodes v_{pb} for $1 \le p \le n$. So none of these "bottom" nodes will fire after the alarm. There is insufficient weight on the incoming L_i signals to hit the new threshold. The alarm spike also reduces the threshold of the nodes v_{qt} for all $q > i$ in column j, from 2 to 1. Thus when any spike arrives on line $q > i$, it meets the threshold for node v_{qt} and the spike passes through the column. That is, any incoming spike on L_{i+k} for $k = 1, \ldots, n - i$ leaves the circuit immediately, going to the right.

Correctness: This circuit does exactly the step for the jth element described above for the classical algorithm. The spikes arriving from the left continue to the right with no delay, that is, with the same value, for all rows except for possibly one. If $L_m < x_j < L_{m+1}$, then the spike on row L_{m+1} exits the column circuit at time x_j. Thus x_j replaces the value for L_{m+1}. All initial values $L_i = \infty$. Thus the correctness of the circuit follows from the correctness of the classical circuit.

The circuit, however, it does not look like a left-to-right scan since the activity is determined by the values of the sequence elements. The smallest element, say at column c, causes the first activity, going all the way to the right and exiting the circuit on line one at time x_c (assuming delays of 0 on all nodes and all links other than the links from the start node). Subsequent columns activate in order of their values, exit on the appropriate line, and travel to the right until they exit the circuit or are stopped by a blocking node at a column with an element that is smaller (arrived earlier) and gives a better final sequence value for that sequence size.

This algorithm runs in time L_{max}, the last element of the longest increasing subsequence. It uses $O(\ell_u n)$ nodes and edges, where ℓ_u is an upper bound on the length of the longest increasing subsequence (n suffices, but is likely conservative) and n is the length of the input sequence.

A real neuromorphic circuit has a non-zero delay through a node. We expect it to be small compared to edge delays in general. Our circuit works as long as the number of node delays for a signal does not change the order of the comparisons at the column nodes. If the node processing time is fixed, or has an upper bound D, we must scale the edge delays so that a one-unit edge delay is greater than nD, where n is the length of the input sequence.

4.4 Longest path in a directed-acyclic graph (DAG)

The neuromorphic circuit to find the longest path in a DAG has a circuit equal to the DAG. That is, there is a node for each DAG node and a directed edge for each DAG edge. If the DAG is unweighted, each edge has weight 1 and delay 1. A vertex is a source if it has no incoming directed edge. All source vertices of the DAG are active at the beginning of the algorithm. A sink, or terminal vertex has no outgoing edges. We read the output at these sink vertices.

The algorithm is similar to the shortest path algorithm, as we start with spikes on the source vertices. Whenever a node receives any spike, it sends a spike to all of its (outgoing) neighbors. Unlike the shortest-paths algorithm, circuit nodes keep firing each time they receive a signal. Delays in the spikes mean that arrival of a new spike correspond to a longer path before this vertex. Subsequently, this node must update its neighbors about this discovery. The algorithm can terminate after n steps, where n is the number of vertices.

Weighted version. If the DAG has edge weights, we add a delay to each edge equal to its weight.

An alternative LIS algorithm. We can also solve the LIS problem by creating a DAG and running the longest-path-in-a-DAG circuit. This requires classical pre-processing to create the DAG. Each sequence entry, x_i is represented by a vertex v_i. We add a directed edge from v_i to v_j if and only if $i < j$ and $x_i < x_j$. An edge (v_i, v_j) means sequence entries i and j can potentially occur (in that order) in a feasible solution to the LIS problem. We can reduce the number of edges by more carefully defining edges. We add a directed edge from v_i to v_j if and only if $i < j$ and $x_i < x_j$, and there is no k such that $i < k < j$ and $x_i < x_k < x_j$.

5 CONCLUSIONS AND FUTURE WORK

We present a general method for converting dynamic programs into efficient spiking neuromorphic algorithms. This enables new neuromorphic algorithms for a wide variety of problems of different flavors. The most pressing open question is whether our approach can be extended to address more general kinds of dynamic programs. Our spiking algorithm for finding longest increasing subsequences suggests that efficient neuromorphic implementations of more sophisticated dynamic programs are possible.

This paper omits some lower-level technical details that will be more extensively presented in a longer format. It is important to note that the high-level algorithms presented here hinge on precise spike timing and delays, which will inevitably result in modifications that are tailored to specific hardware platforms. These modifications notwithstanding, our results demonstrate that the structure of neuromorphic architectures may have advantages for graph analysis, even when moving beyond straightforward algorithms such as shortest paths to more complex dynamic programming applications.

ACKNOWLEDGEMENTS

Supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

REFERENCES

[1] R. Aibara, Y. Mitsui, and T. Ae. 1991. A CMOS chip design of binary neural network with delayed synapses. In *1991., IEEE International Sympoisum on Circuits and Systems*. 1307–1310 vol.3. https://doi.org/10.1109/ISCAS.1991.176611

[2] James B. Aimone, Ojas Parekh, and William Severa. 2017. Neural computing for scientific computing applications: more than just machine learning. In *NCS*.

[3] Anders Andrae and Tomas Edler. 2015. On global electricity usage of communication technology: trends to 2030. *Challenges* 6, 1 (2015), 117–157.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

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

[6] Michael L. Fredman. 1975. On computing the length of the longest increasing subsequences. *Discrete Mathematics* 11, 1 (1975), 29–35.

[7] Steve B Furber, David R Lester, Luis A Plana, Jim D Garside, Eustace Painkras, Steve Temple, and Andrew D Brown. 2013. Overview of the spinnaker system architecture. *IEEE Trans. Comput.* 62, 12 (2013), 2454–2467.

[8] András Hajnal, Wolfgang Maass, Pavel Pudlák, Mario Szegedy, and György Turán. 1993. Threshold circuits of bounded depth. *J. Comput. System Sci.* 46, 2 (1993), 129–154.

[9] Kathleen E Hamilton, Tiffany M Mintz, and Catherine D Schuman. 2019. Spike-based primitives for graph algorithms. *arXiv preprint arXiv:1903.10574* (2019).

[10] Scott Hemmert. 2010. Green hpc: From nice to necessity. *Computing in Science & Engineering* 12, 6 (2010), 8–10.

[11] John J Hopfield and David W Tank. 1985. "Neural" computation of decisions in optimization problems. *Biological cybernetics* 52, 3 (1985), 141–152.

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

[13] Jeffrey L. Krichmar, William Severa, Muhammad S. Khan, and James L. Olds. 2019. Making BREAD: Biomimetic Strategies for Artificial Intelligence Now and in the Future. *Frontiers in Neuroscience* 13 (2019), 666. https://doi.org/10.3389/fnins.2019.00666

[14] Xavier Lagorce and Ryad Benosman. 2015. Stick: spike time interval computational kernel, a framework for general purpose computation using neurons, precise timing, delays, and synchrony. *Neural computation* 27, 11 (2015), 2261–2317.

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[16] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[17] Paul A Merolla, John V Arthur, Rodrigo Alvarez-Icaza, Andrew S Cassidy, Jun Sawada, Filipp Akopyan, Bryan L Jackson, Nabil Imam, Chen Guo, Yutaka Nakamura, et al. 2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 6197 (2014), 668–673.

[18] John V Monaco and Manuel M Vindiola. 2017. Integer factorization with a neuromorphic sieve. In *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on.* IEEE, 1–4.

[19] NVIDIA. 2019. nvGraph. https://developer.nvidia.com/nvgraph

[20] Ojas Parekh, Cynthia A Phillips, Conrad D James, and James B Aimone. 2018. Constant-Depth and Subcubic-Size Threshold Circuits for Matrix Multiplication. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures.* ACM, 67–76.

[21] Daniel Rasmussen. 2018. NengoDL: Combining deep learning and neuromorphic modelling methods. *arXiv preprint arXiv:1805.11144* (2018).

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

[23] William Severa, Richard Lehoucq, Ojas Parekh, and James B. Aimone. 2018. Spiking Neural Algorithms for Markov Process Random Walk. In *International Joint Conference on Neural Networks 2018.* IEEE.

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

[25] William Severa, Craig M. Vineyard, Ryan Dellana, Stephen J. Verzi, and James B. Aimone. In Press. Training deep neural networks for binary communication with the Whetstone method. *Nature: Machine Intelligence* (In Press).

[26] John M Shalf and Robert Leland. 2015. Computing beyond moore's law. *Computer* 48, 12 (2015), 14–23.

[27] Stephen J Verzi, Fredrick Rothganger, Ojas D Parekh, Tu-Thach Quach, Nadine E Miner, Craig M Vineyard, Conrad D James, and James B Aimone. 2018. Computing with spikes: The advantage of fine-grained timing. *Neural computation* (2018), 1–31.

[28] M Mitchell Waldrop. 2016. The chips are down for Moore's law. *Nature News* 530, 7589 (2016), 144.

[29] Wikipedia. 2019. Dynamic programming. https://en.wikipedia.org/wiki/Dynamic_programming

[30] GV Wilson and GS Pawley. 1988. On the stability of the travelling salesman problem algorithm of Hopfield and Tank. *Biological Cybernetics* 58, 1 (1988), 63–70.

[31] Haoduo Yang, Huayou Su, Mei Wen, and Chunyuan Zhang. 2018. HPGA: A High-Performance Graph Analytics Framework on the GPU. In *2018 International Conference on Information Systems and Computer Aided Education (ICISCAE).* IEEE, 488–492.