2025 Design, Automation & Test in Europe Conference (DATE 2025) – Focus Session

Exploring Dendritic Computation in Bio-Inspired Architectures For Dynamic Programming

Anup Das
Electrical and Computer Engineering, Drexel University
Philadelphia, USA
Email: anup.das@drexel.edu

***Abstract*—Dynamic programming is a classical optimization technique that systematically decomposes a complex problem into simpler sub-problems to find an optimal solution. We explore the use of bio-inspired architectures to find the shortest path between two nodes in a graph using dynamic programming. We leverage dendritic computations, which are linear and non-linear mechanisms in neuronal dendrites that allow to implement different computational primitives. We exploit two key mechanisms: 1) a dendrite acts as a delay line to propagate an excitatory post-synaptic potential to the soma, and 2) a feedback mechanism from the soma into the dendrites to control this delay. Our key ideas are the following. First, we model each node on a graph as a leaky integrate-and-fire (LIF) neuron, supporting the two dendritic mechanisms. We use a countdown counter to implement forward propagation of a delayed synaptic potential and eligibility trace-based feedback to update the delay by incorporating the cost of edges in a graph. Next, we formulate dynamic programming in terms of the time to the first spike in neurons. We breakdown the shortest path problem into sub-problems of finding the earliest firing times of neurons, and iteratively building the final solution from these smaller sub-problems by tracing backward. We implement this approach for several real-world graphs and show its scalability. We also show early prototyping on a Virtex UltraScale FPGA.**

***Index Terms*—bio-inspired architecture, dendritic computation, dynamic programming, shortest path, graph**

I. Introduction

Dynamic programming is an optimization technique to find an optimal solution to a complex problem by breaking down the problem into simpler sub-problems [1]. It operates on the principle of optimal substructure, where the optimal solution to a larger problem can be constructed from the optimal solutions of its sub-problems [2]. It operates on the principle of optimal substructure, where the optimal solution to a larger problem can be constructed from the optimal solutions of its sub-problems. By storing solutions to overlapping sub-problems, dynamic programming eliminates redundant computations and achieves a polynomial-time complexity for problems that would otherwise require exponential time using naive recursive approaches. Dynamic programming was formalized by Richard Bellman in the 1950s [3], and has since become fundamental in algorithmic problem-solving in various domains including sequence alignment in bioinformatics [4], shortest path computation in network routing [5], and resource allocation [6]. Dynamic programming is typically executed on a CPU [7]. Recent approaches have tried to accelerate dynamic programming on GPUs by exploiting their massively parallel computing structures [8]–[10].

We explore the use of bio-inspired architectures to perform dynamic programming with the objective of finding the shortest path between two nodes in a graph. These are architectures that compute using the principles of the brain. Spiking Neural Networks (SNNs), which are often regarded as the third and the most recent generation of neural networks [11], are examples of bio-inspired architectures. SNNs translate a relatively simple computational property of a biological neuron, that of integrate-and-fire (IF). In this simple model, a neuron sums up the synaptic input and decides whether to generate an action potential (spike) by comparing the sum with a threshold. Connecting these IF units, an SNN can represent a complex structure – multilayer perceptron (MLP), convolutional neural network (CNN), and recurrent neural network (RNN). The connection between IF units can be trained to give the desired problem solving ability to an SNN [12]–[18]. A recent work has demonstrated dynamic programming using SNNs with simple neuron model [19].

Recent research works on the modeling of biological properties in neurons suggest that a neuron exhibits additional linear and nonlinear mechanisms in its dendritic tree that can serve as computation building blocks [20]–[23]. We explore two such mechanisms. First, a neuron acts as a delay line to propagate an excitatory postsynaptic potential originating in the dendrite to the soma. Second, a feedback mechanism from the soma into the dendrite to control this delay [20].

Contributions – To solve dynamic programming in the context of finding the shortest path between two nodes in a graph, we use SNN with neurons designed as IF units with a feedback structure to support the two dendritic computations (see Fig. 2). Our key ideas are the following.

1) We model a graph as an SNN using feedback-enabled IF units (FEIF). We use a countdown counter to implement the propagation delay in the forward direction (i.e., dendrite to soma). We use a trace-based soma-to-dendrite eligibility feedback to update the forward delay incorporating edge costs.
2) We implement the shortest path problem as a collection of sub-problems, that of finding the earliest firing time of FEIF units in a model by injecting a spike to the source. We propose a linear-time algorithm to construct the final

solution by tracing backward from destination to source using the solution to the sub-problems.

We implement the proposed approach for several real-world graphs with a wide distribution of in- and out-degrees. We show the scalability of the approach for these graph complexities. We also show an early hardware prototype of the design on a Virtex UltraScale FPGA.

II. Shortest Path Problem in Graphs

Figure 1 shows an example of finding the shortest path between nodes v_0 and v_4 in a directed graph with weights indicated on the edges. The shortest path consists of two nodes between the source and destination: $v_0(\text{src}) \to v_1 \to v_4 \to v_3(\text{dst})$. This is highlighted in Fig. 1c. The minimum cost on the shortest path is 13. We introduce the following definitions.

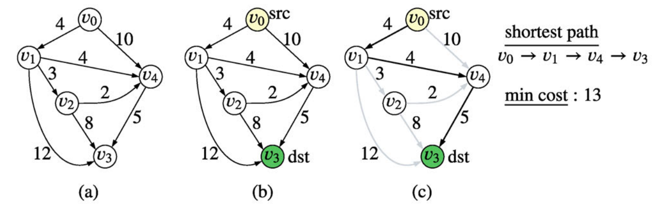


Fig. 1: Finding the shortest path in a graph.

Definition 1: (Nodes and Edges) *A node is a vertex in a graph. An edge is an arc connecting two nodes.*

Definition 2: (Graphs) *A graph $G = (V, E)$ consists of a set V of nodes and a set E of edges. An edge is represented using its source and destination pair, i.e., $e_{i,j} = (\mathbf{v}_i, \mathbf{v}_j)$. An edge $e_{i,j}$ has a weight $W(e_{i,j})$, which represents the capacity of the edge or the cost of traversing the edge.*

Definition 3: (Shortest Path) *The shortest path between the source node $\mathbf{v}_s$ and the destination node $\mathbf{v}_d$, denoted as $\delta(\mathbf{v}_s, \mathbf{v}_d)$, is the sequence of edges $(\mathbf{v}_0, \mathbf{v}_1)(\mathbf{v}_1, \mathbf{v}_2)\cdots(\mathbf{v}_{k-1}, \mathbf{v}_k)$, such that $\mathbf{v}_0 = \mathbf{v}_s$, $\mathbf{v}_k = \mathbf{v}_d$, and the total weight $\sum_{i=0}^{k-1} W(e_{i,i+1})$ is the least.*

Dynamic programming can be used to find the shortest path in a graph. We formulate this as follows.

- **Initialize:**
  - $d(\boldsymbol{v}_s) = 0$
  - $d(\boldsymbol{v}_i) = \infty \;\; \forall \;\; \boldsymbol{v}_i \in V \setminus \boldsymbol{v}_s$
- **Iterate:**
  - for $i = 0$ to $|V| - 1$
  $$d(\boldsymbol{v}_j) = \texttt{minimum}\{d(\boldsymbol{v}_j), d(\boldsymbol{v}_i) + W(e_{i,j})\} \;\; \forall \;\; e_{i,j} \in E$$

The above formulation consists of an **optimal substructure**, defined as follows. Let $P = \langle \boldsymbol{v}_0^p, \boldsymbol{v}_1^p, \cdots, \boldsymbol{v}_{k-1}^p \rangle$ with $\boldsymbol{v}_i^p \in V$ be the shortest path between $\boldsymbol{v}_0^p$ and $\boldsymbol{v}_{k-1}^p$. Then the sub-path $\langle \boldsymbol{v}_i^p, \cdots, \boldsymbol{v}_j^p \rangle$ is the shortest path between $\boldsymbol{v}_i^p$ and $\boldsymbol{v}_j^p$.

The baseline complexity of the dynamic programming formulation of the shortest path algorithm is $\mathcal{O}(|V| \cdot |E|)$.

III. Feedback Enabled IF Neurons

Figure 2 (left) shows a simple integrate-and-fire (IF) neuron [24]. In this model, a neuron is characterized by its membrane potential (mem), which accumulates spikes from synaptic inputs that are received from other neurons through their associated synapses. We consider these synapses to be current-based. Therefore, an input to the membrane is modeled as a current obtained by linear summation of its inputs, weighted by their respective synaptic strengths as

$$I(t) = \sum_i s_i \cdot x_i(t) \tag{1}$$

where s_i is the synaptic strength of i^{th} input and $x_i(t)$ is the input spike at time t. A neuron is modeled as an electrical RC circuit, with the membrane potential represented as voltage $U(t)$ across its capacitor. Therefore, the first-order differential equation for membrane potential is

$$\tau \frac{dU(t)}{dt} = -U(t) + R \cdot I(t) \tag{2}$$

where $\tau = R \cdot C$ is the membrane time constant.

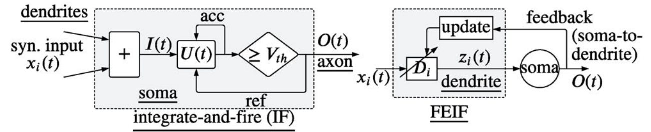


Fig. 2: Simple model of an integrate-and-fire neuron (left). A neuron with feedback enabled dendritic computing (right).

Output spikes are generated using a nonlinear function as

$$O(t) = \begin{cases} 1 & U(t) \geq V_{th} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

This is the only nonlinear neuron characteristic exploited in most SNNs to solve complex problems such as classification (image, text, and audio) and regression [20].

To model the impact of dendritic computations in a neuron, we simplify existing IF implementation (both hardware and software) by considering the following.

- We use the Forward Euler method to unroll membrane potential computations over time, i.e.,
  $$U(t) = U(t-1) + I \tag{4}$$
  Note that Eq. 4 can be obtained from Eq. 2 using a proper selection of R and C.
- We use unit synaptic strengths, i.e.,
  $$s_i = 1 \;\; \forall \;\; i \tag{5}$$

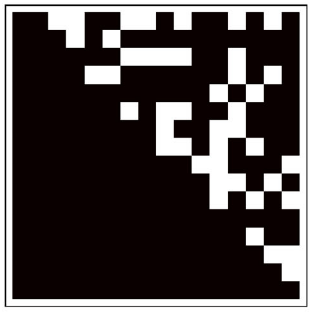
Figure 2 (right) shows a dendrite of the IF neuron (left). This dendrite is implemented as a feedback mechanism. In the forward direction, an input spike $x_i(t)$ is delayed by an amount D_i. The delayed spike is used to compute the membrane potential inside the soma using Eqs. 1-3. The soma-to-dendrite feedback is used to update the delay in the forward direction.

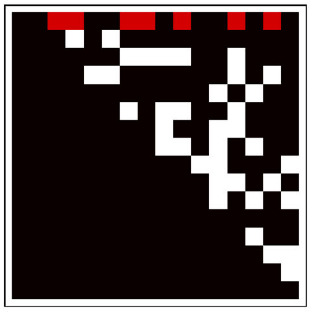
To generate a delayed synaptic input, we introduce a time-dependent intermediate variable $d_i(t)$, defined as

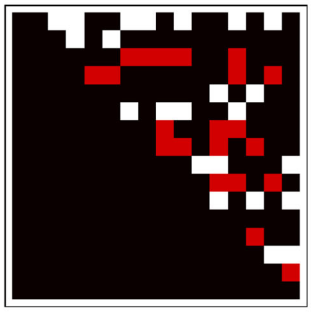
$$d_i(t) = \begin{cases} D_i & x_i(t) = 1 \\ \texttt{maximum}\Big(d_i(t-1) - 1, 0\Big) & \text{otherwise} \end{cases} \tag{6}$$

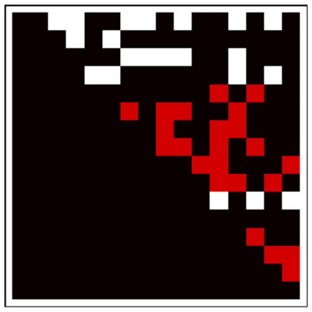
The dendrite-to-soma output in the forward direction is

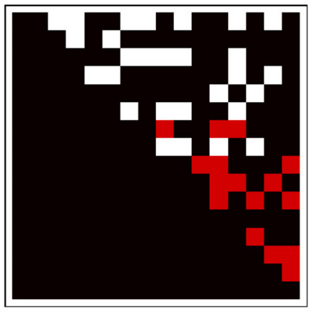
$$z_i(t) = \begin{cases} 1 & d_i(t) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

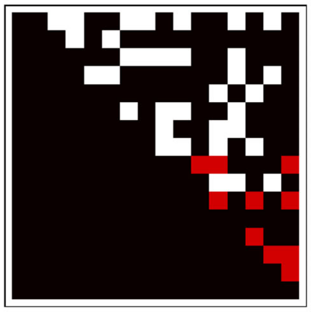

(a) $t = 0$.

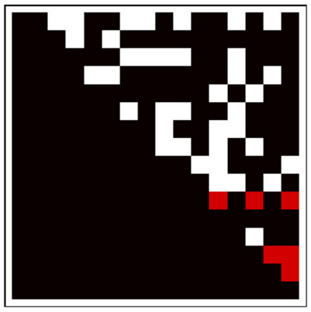

(b) $t = 1$.


(c) $t = 2$.


(d) $t = 3$.


(e) $t = 4$.


(f) $t = 5$.


(g) $t = 6$.

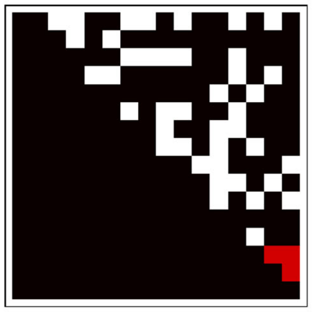

(h) $t = 7$.

Fig. 3: Synaptic activity propagation through a connected graph for 8 time steps.

Thereafter, we use a linear input summation in the soma as before (see Eq. 1), to generate a current as

$$I = \sum_i z_i \tag{8}$$

Finally, we use Eqs. 2 and 3 to generate an output spike.

The soma-to-dendrite feedback is used to update the delay D_i. This is problem-dependent. We describe this in Sec. IV-D.

In the following, we describe how an SNN with FEIF neurons can be used to solve another class of complex problems – that of solving dynamic programming with application to finding the shortest path between two nodes in a graph.

IV. Solving Shortest Path Problem using an SNN with FEIF Neurons

To compute the shortest path between two nodes in a graph, we model a graph as an SNN as follows.

Definition 4: (SNN GRAPHS) *An SNN $S_G = (A, C)$ of a graph $G = (V, E)$ consists of a set A of FEIF neurons and a set C of synaptic connections. The neuron $\mathbf{a}_i \in A$ corresponds to node $\mathbf{v}_i \in V$, while the connection $c_{i,j} \in C$ corresponds to edge $e_{i,j} \in E$. Each neuron (soma) is associated with a threshold voltage V_{th}. Each connection $c_{i,j} \in C$ is associated with a forward propagation delay $D_{i,j}$.*

Figure 3(a) shows the adjacency matrix of a random directed graph with 16 nodes generated using the NetworkX tool [25]. In this figure, a white block represents a connection between two nodes, while a dark block represents no connection. For this initial exploration, we consider a directed graph without any cycles in it. Therefore, the adjacency matrix is essentially an upper triangular matrix, as shown in the figure.

We use this adjacency matrix to define connections between FEIF neurons in the SNN representation of the graph. To illustrate the impact of forward propagation dendritic delays, we first consider unit delay, i.e., $D_{i,j} = 1, \ \forall \ c_{i,j} \in C$. Later, we relax this with actual delays for a given problem in Sec. IV-D.

A. Synaptic Activity Propagation: The Forward Pass

To illustrate synaptic activity propagation in an SNN, we inject a spike into neuron $\boldsymbol{a}_0$ at time $t = 0$. This synaptic activity propagates to its connected neurons. Due to unit dendritic delay, the soma of these neurons will receive a spike at time $t = 1$ (see Fig. 3b). If we set a very low value of V_{th}, a neuron will fire a spike upon receiving an input (Eqs 1-8). We show the synaptic activity at time $t = 1$ using red-colored blocks.

The SNN will continue to propagate spikes through the connected neurons. We illustrate the synaptic activity propagation for the remaining time steps in Figs. 3c-h. Synaptic activity propagation stops when it reaches the leaf nodes in the graph.

The maximum propagation time is defined as the time it takes for the activity to reach all leaf nodes in a graph. This is equal to the depth of a graph.

B. Shortest Path Extraction: The Backward Pass

To model the shortest path problem, we make two key modifications to the synaptic activity propagation.

- First, we inject a spike into the source node $\boldsymbol{a}_s \in V$.
- Second, we stop the synaptic activity propagation when it reaches the destination node $\boldsymbol{a}_d \in V$.

The time interval from the start to end of synaptic propagation is the length of the shortest path between $\boldsymbol{a}_s$ and $\boldsymbol{a}_d$. To find the sequence of nodes that are on this shortest path, we propose to decompose the problem into smaller sub-problems, that of finding the earliest firing time of FEIF neurons with respect to when the activity propagation reaches the destination node in the graph. We describe this in the following two steps.

- First, we record the time-to-first-spike (TTFS) in neurons as synaptic activity propagates through them. Let T_i be the TTFS of neuron $\boldsymbol{a}_i \in A$.
- Next, we **trace backward** from the destination neuron $\boldsymbol{a}_d$ to source $\boldsymbol{a}_s$ by considering the earliest firing times of connected neurons at each step.

Consider that $\boldsymbol{a}_j$ be the neuron currently selected to be on the shortest path at a specific step during the backward tracing. We introduce a variable τ_i^j to capture both connectivity and TTFS with respect to $\boldsymbol{a}_j$ as follows.

$$\tau_i^j = \begin{cases} T_j - T_i & c_{i,j} = 1 \\ \infty & \text{otherwise} \end{cases} \tag{9}$$

The neuron selected for the next step in the backward tracing is obtained as

$$\boldsymbol{a}_{\text{next}} = \texttt{argmin}\{\tau_i^j\} \tag{10}$$

Therefore, our iterative algorithm is as follows.

- **Initialize:**
  - shortest_path = $\{\boldsymbol{a}_d\}$
  - $\boldsymbol{a}_{\text{current}} = \boldsymbol{a}_d$
- **Iterate:**
  - while $\boldsymbol{a}_{\text{current}} \neq \boldsymbol{a}_s$
    * $\boldsymbol{a}_{\text{next}} = \texttt{argmin}\{\tau_i^{\text{current}} \mid \forall\ \boldsymbol{a}_i \in A\}$
    * shortest_path.$\texttt{push}(\boldsymbol{a}_{\text{next}})$
    * $\boldsymbol{a}_{\text{current}} = \boldsymbol{a}_{\text{next}}$

Proof – To prove that the path computed using this formulation is indeed the shortest path, we use induction. Consider that the above formulation generates a path $P_0^{k-1} = \langle \boldsymbol{a}_0, \boldsymbol{a}_1, \cdots, \boldsymbol{a}_{k-1}\rangle$ from source $\boldsymbol{a}_0$ to destination $\boldsymbol{a}_{k-1}$.

- *Assume that the sub-path $P_i^{k-1} = \langle \mathbf{a}_i, \mathbf{a}_{i+1}, \cdots, \mathbf{a}_{k-1}\rangle$ is the shortest path between $\mathbf{a}_i$ and $\mathbf{a}_{k-1}$.* The cost of this sub-path is defined in terms of propagation delay as

$$\mathcal{C}_i^{k-1} = \sum_{j=i}^{k-2}(T_{j+1} - T_j) = (T_{k-1} - T_i) \tag{11}$$

Since sub-path P_i^{k-1} is the shortest path, the cost $\mathcal{C}_i^{k-1}$ is the minimum. Now, we add neuron $\boldsymbol{a}_{i-1}$ to sub-path P_i^{k-1} to generate sub-path P_{i-1}^{k-1}. The cost of this sub-path is

$$\mathcal{C}_{i-1}^{k-1} = (T_{k-1} - T_{i-1}) = (T_{k-1} - T_i) + (T_i - T_{i-1}) \tag{12}$$

Using Eq. 11, we can rewrite Eq. 12 as

$$\mathcal{C}_{i-1}^{k-1} = \mathcal{C}_i^{k-1} + (T_i - T_{i-1}) \tag{13}$$

Using Eqs. 9-10, $(T_i - T_{i-1})$ is the minimum cost between $\boldsymbol{a}_i$ and $\boldsymbol{a}_{i-1}$. Therefore, $\mathcal{C}_{i-1}^{k-1}$ (Eq. 13) is also minimum, given that $\mathcal{C}_i^{k-1}$ is minimum. Therefore, by definition, *the sub-path P_{i-1}^{k-1} is the shortest path between $\mathbf{a}_{i-1}$ and $\mathbf{a}_{k-1}$.*
- Next, we use mathematical induction to prove that the path P_0^{k-1} is the shortest path through recursion by iterating the proof for $i = k-1, k-2, \cdots, 0$.

In the above proof, the sub-paths P_i^{k-1} are the optimal substructures, which make the proposed approach a dynamic programming formulation.

C. Computational Complexity

The time complexity of finding a shortest path in a graph is computed as follows. The synaptic activity propagation through a graph has a worst case complexity of $\mathcal{O}(|V|)$. This is the forward pass of the approach, which propagates the activity through each neuron and record the TTFS. Next, the iterative algorithm traces backward through the neurons. For each iteration, it computes the value of variables τ_i^j. Each computation can be performed in unit time. Therefore, the time complexity of the backward pass is $\mathcal{O}(|V|^2)$. The overall time complexity of finding a shortest path in a graph is

$$\text{time complexity} = \mathcal{O}(|V|) + \mathcal{O}(|V|^2) \approx \mathcal{O}(|V|^2) \tag{14}$$

D. Updating Dendritic Delays By Incorporating Edge Costs

In a weighted graph, each edge has an associated cost. Therefore, the shortest path needs to be computed incorporating edge costs. To generalize the approach, we propose a mechanism to update the unit delays $D_{i,j}$ using these edge costs by constructing an eligibility trace.

Let E_j be a variable that captures the eligibility trace of neuron $\boldsymbol{a}_j$. This variable is defined as follows.

$$E_j(t) = \begin{cases} 1 & O_j(t) = 1 \\ E_j(t-1) - E_j(t-1)/\eta & \text{otherwise} \end{cases} \tag{15}$$

where η is the decay rate of the trace variable $E_j(t)$. Essentially, the trace variable is set to 1 when neuron $\boldsymbol{a}_j$ fires a spike. Otherwise, the variable decays over time.

We use this variable to update the dendritic delay $D_{i,j}$ as

$$D_{i,j}^{(n)} = D_{i,j}^{(n-1)} + \alpha \cdot L_{i,j}^{(n-1)} \cdot E_j(t) \tag{16}$$

where $D_{i,j}^{(n)}$ is the delay at the n^{th} iteration of the algorithm, α is the learning rate, and $L_{i,j}^{(n-1)}$ is the loss computed as

$$L_{i,j}^{(n-1)} = W(e_{i,j}) - D_{i,j}^{(n-1)} \tag{17}$$

The above equation incorporates the edge cost $W(e_{i,j})$ in computing the delay updates.

E. Iterative Shortest Path Computation on Weighted Graphs

Algorithm 1 shows the pseudo-code of the proposed iterative approach to shortest path computation.

Algorithm 1: Iterative Shortest Path Computation.

```
Input: SNN Graph $S_G$, Number of iterations $N_{iter}$
Output: Shortest Path $P$.
1 $n = 0, P = \emptyset, D = 1$                      /* Initialize. */
2 for $0 \leq n < N_{iter}$ do                 /* Run for the iterations. */
3     $T$ = ForwardPass($D$)                    /* Activity propagation. */
4     $P$ = BackwardPass($T$)                   /* Path Extraction. */
5     $D$ = ETrace($D$)                         /* Delay update. */
6 return $P$
```

V. Results and Discussions

We evaluate our shortest path computation on 4 real-world graphs as summarized in Table I. These workloads are selected to represent social networking, peer-to-peer (P2P) internet, autonomous systems, and email communication [26].

	Nodes	Edges	Max. Out Degree	Types
WikiVote	7,115	201,524	50	Social
Gnutella	10,876	79,988	20	P2P
Skitter	192,244	1,218,132	25	Autonomous
Euall	265,214	730,051	14	Email

TABLE I. Evaluated real-world graphs.

Figure 4 shows the cumulative distribution function (CDF) of the out degrees in these graphs.

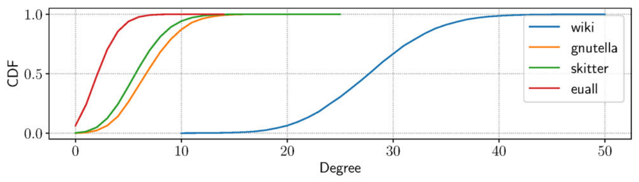


Fig. 4: CDF of out degrees.

A. Time Complexity

Table II reports the time complexity of the proposed solution compared to using dynamic programming for finding the shortest path between two nodes in a graph. We report results for the shortest path between two separate pairs of source and destination nodes in each evaluated graphs.

	Path Length (# edges)	DP	Proposed
WikiVote	55	1.43E+09	3.91E+05
	214	1.43E+09	1.52E+06
Gnutella	96	8.70E+08	1.04E+06
	433	8.70E+08	4.71E+06
Skitter	719	2.34E+11	1.38E+08
	2613	2.34E+11	5.02E+08
Euall	1190	1.94E+11	3.16E+08
	8083	1.94E+11	2.14E+09

TABLE II. Improvement compared to dynamic programming (DP).

For dynamic programming, the complexity in solving the shortest path problem depends on the number of nodes and edges. Essentially, the algorithm iterates through all nodes in the outer loop (see Sect. II). Within each loop, it analyzes all edges, giving a time complexity equal to the number of nodes times the number of edges. However, the proposed approach iterates through only the edges that are part of the shortest path, i.e., the path length. Within each iteration, it analyzes all nodes to use their TTFS. Therefore, the time complexity is $\approx$ path length times the number of nodes.

B. Hardware Implementation

We prototype the proposed approach on a Virtex UltraScale FPGA board, whose resource availability is the following.

- **Lookup Table (LUT):** 537,600
- **Registers (FF):** 1,075,200
- **Digital Signal Processing Blocks (DSP):** 768
- **Block RAMs (BRAM):** 1728

Figure 5 shows the schematic of the proposed FEIF neuron.

Table III reports the resources needed to implement a FEIF neuron. For reference we have included the resources needed for three state-of-the-art leaky-integrate-and-fire neuron designs that use the forward Euler method to unroll membrane potential computations in time. We observe that compared to [27], the proposed design reduces the number of LUT by 11%, FF by 50%, while using 2 extra DSPs. The improvement in LUT and FF is due to the use of simple IF neurons in FEIF with optimized dendritic delays. The two DSPs are needed to implement the quantized multiplication to implement the delay update using the eligibility trace. Compared to [28], the proposed design uses 10% more LUT and 2$\times$ more FF. Finally, compared to [29], the proposed design introduces only 5% more LUT and 1.7$\times$ more FF.

	Euler [27]	Euler [28]	Euler [29]	FEIF (Proposed)
LUT	95	76	80	84
FF	85	20	23	40
BRAM	0	0	0	0
DSP	0	0	0	2

TABLE III. Resource utilization compared to state-of-the-art.

Recently, several solutions are proposed to optimize the design of spiking neural networks on FPGA [29]–[31]. There are also techniques proposed to efficiently implement SNNs that represent graphs [32]–[34]. In the future, we will integrate dendritic computation in these existing solutions.

VI. Conclusions and Future Outlook

We introduce a novel approach to solving dynamic programming, that of using dendritic computations in spiking neurons. We model two computing mechanisms. In the forward direction, a dendrite acts as a delay to propagate spike from dendrite-to-soma. In the backward direction, a reverse potential from soma-to-dendrite controls the delay. We model dynamic programming for the problem of computing the shortest path between two nodes in a graph. To do so, we introduce the following key ideas. First, we represent a graph as a spiking neural network (SNN), where each neuron is implemented as an integrate-and-fire unit with the feedback loop modeling the dendritic computation. When a spike is injected into the source neuron, spikes propagate through the SNN between connected neurons. We stop this propagation when the destination neuron fires a spike. We record the time-to-first-spike of neurons as synaptic activity propagates through the SNN. Next, we trace backward from destination to source, identifying the earliest firing neuron at each step. We prove using mathematical induction that the tracing back operation results in extracting the shortest path between the source and destination. Finally, we use an eligibility trace to update dendritic delays by incorporating edge weights. This allows to find the shortest path in a weighted graph.

We evaluate the proposed approach for several real-world graphs and show the improvement in the computational complexity. We also show an early FPGA prototype of the design.

In the future, we will consider and demonstrate other use-cases of dynamic programming, including Fibonacci sequence

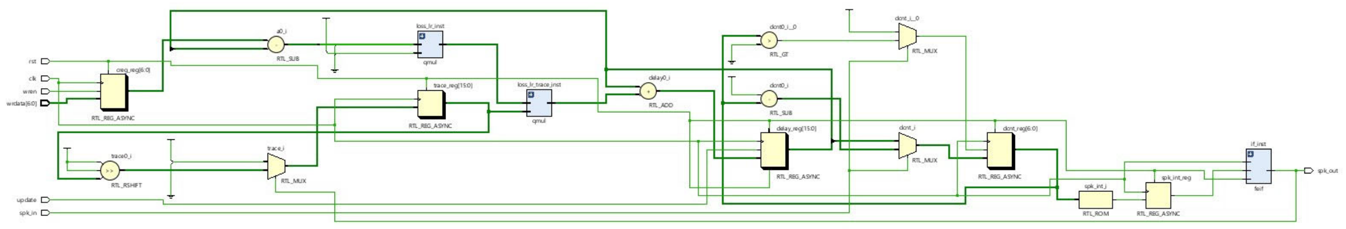


Fig. 5: Schematic of the proposed FEIF neuron.

generation, longest common subsequence identification, and Knapsack problem solutions.

ACKNOWLEDGMENT

This work is supported by Accenture LLP, the DOE DE-SC0022014, and NSF CCF-1942697.

REFERENCES

[1] R. Bellman, “Dynamic programming,” *science*, vol. 153, no. 3731, pp. 34–37, 1966.

[2] R. A. Howard, “Dynamic programming,” *Management Science*, vol. 12, no. 5, pp. 317–348, 1966.

[3] R. Bellman, “The theory of dynamic programming,” *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[4] S. R. Eddy, “What is dynamic programming?” *Nature Biotechnology*, vol. 22, no. 7, pp. 909–910, 2004.

[5] D. B. Johnson, “A note on dijkstra’s shortest path algorithm,” *Journal of the ACM (JACM)*, vol. 20, no. 3, pp. 385–388, 1973.

[6] S. E. Elmaghraby, “Resource allocation via dynamic programming in activity networks,” *European Journal of Operational Research*, vol. 64, no. 2, pp. 199–215, 1993.

[7] R. A. Chowdhury and V. Ramachandran, “Cache-efficient dynamic programming algorithms for multicores,” in *Annual symposium on Parallelism in Algorithms and Architectures*, 2008, pp. 207–216.

[8] K.-E. Berger and F. Galea, “An efficient parallelization strategy for dynamic programming on GPU,” in *IEEE International Symposium on Parallel & Distributed Processing (IPDPS), Workshops and Phd Forum*, 2013, pp. 1797–1806.

[9] S. Xiao, A. M. Aji, and W.-c. Feng, “On the robust mapping of dynamic programming onto a graphics processing unit,” in *International Conference on Parallel and Distributed Systems (ICPADS)*, 2009, pp. 26–33.

[10] V. Boyer, D. El Baz, and M. Elkihel, “Dense dynamic programming on multi GPU,” in *Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2011, pp. 545–551.

[11] W. Maass, “Networks of spiking neurons: the third generation of neural network models,” *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[12] K. Yamazaki, V.-K. Vo-Ho, D. Bulsara, and N. Le, “Spiking neural networks and their applications: A review,” *Brain Sciences*, vol. 12, no. 7, p. 863, 2022.

[13] A. Samadzadeh, F. S. T. Far, A. Javadi, A. Nickabadi, and M. H. Chehreghani, “Convolutional spiking neural networks for spatio-temporal feature extraction,” *Neural Processing Letters*, vol. 55, no. 6, pp. 6979–6995, 2023.

[14] A. Paul, N. Kandasamy, K. Dandekar, and A. Das, “Data driven learning of aperiodic nonlinear dynamic systems using spike based reservoirs-in-reservoir,” 2024.

[15] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida, “Deep learning in spiking neural networks,” *Neural Networks*, vol. 111, pp. 47–63, 2019.

[16] A. Paul and A. Das, “Learning in recurrent spiking neural networks with sparse full-FORCE training,” 2024.

[17] S. Kim, S. Park, B. Na, and S. Yoon, “Spiking-yolo: spiking neural network for energy-efficient object detection,” in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 07, 2020, pp. 11 270–11 277.

[18] Y. Cao, Y. Chen, and D. Khosla, “Spiking deep convolutional neural networks for energy-efficient object recognition,” *International Journal of Computer Vision*, vol. 113, pp. 54–66, 2015.

[19] J. B. Aimone, O. Parekh, C. A. Phillips, A. Pinar, W. Severa, and H. Xu, “Dynamic programming with spiking neural computing,” in *Proceedings of the International Conference on Neuromorphic Systems*, 2019, pp. 1–9.

[20] M. London and M. Häusser, “Dendritic computation,” *Annu. Rev. Neurosci.*, vol. 28, no. 1, pp. 503–532, 2005.

[21] J. Acharya, A. Basu, R. Legenstein, T. Limbacher, P. Poirazi, and X. Wu, “Dendritic computing: branching deeper into machine learning,” *Neuroscience*, vol. 489, pp. 275–289, 2022.

[22] S. Li, N. Liu, X. Zhang, D. W. McLaughlin, D. Zhou, and D. Cai, “Dendritic computations captured by an effective point neuron model,” *Proceedings of the National Academy of Sciences*, vol. 116, no. 30, pp. 15 244–15 252, 2019.

[23] L. Fischer, R. M. Soto-Albors, V. D. Tang, B. Bicknell, C. Grienberger, V. Francioni, R. Naud, L. M. Palmer, and N. Takahashi, “Dendritic mechanisms for in vivo neural computations and behavior,” *Journal of Neuroscience*, vol. 42, no. 45, pp. 8460–8467, 2022.

[24] A. N. Burkitt, “A review of the integrate-and-fire neuron model: I. homogeneous synaptic input,” *Biological Cybernetics*, vol. 95, pp. 1–19, 2006.

[25] A. Hagberg, P. J. Swart, and D. A. Schult, “Exploring network structure, dynamics, and function using networkx,” Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[26] K. Qiu, Y. Zhu, J. Yuan, J. Zhao, X. Wang, and T. Wolf, “Parapll: Fast parallel shortest-path distance query on large-scale weighted graphs,” in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[27] W. Guo, H. E. Yantır *et al.*, “Toward the optimal design and FPGA implementation of spiking neural networks,” *IEEE TNNLS*, 2021.

[28] W. Ye, Y. Chen, and Y. Liu, “The implementation and optimization of neuromorphic hardware for supporting spiking neural networks with MLP and CNN topologies,” *IEEE TCAD*, 2022.

[29] S. Matinizadeh, A. Mohammadhassani, N. Pacik-Nelson, I. Polykretisl, A. Mishra, J. Shackleford, N. Kandasamy, E. Gallo, and A. Das, “A fully-configurable digital spiking neuromorphic hardware design with variable quantization and mixed precision,” in *IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, 2024, pp. 937–941.

[30] J. Li, G. Shen, D. Zhao, Q. Zhang, and Y. Zeng, “Firefly: A high-throughput hardware accelerator for spiking neural networks with efficient dsp and memory optimization,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 8, pp. 1178–1191, 2023.

[31] ——, “Firefly v2: advancing hardware support for high-performance spiking neural network with a spatiotemporal fpga accelerator,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[32] A. Das, “Neuromorphic computing for graph analytics,” in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2024.

[33] T. Li, J. Li, G. Shen, D. Zhao, Q. Zhang, and Y. Zeng, “FireFly-S: Exploiting dual-side sparsity for spiking neural networks acceleration with reconfigurable spatial architecture,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2024.

[34] A. Mohammadhassani, S. Matinizadeh, L. M. Varshika, and A. Das, “A digital neuromorphic architecture for unsupervised shortest path computation on real-world graphs,” in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2025.