PATHFINDING NEURAL CELLULAR AUTOMATA

Anonymous authors
Paper under double-blind review

ABSTRACT

Pathfinding makes up an important sub-component of a broad range of complex tasks in AI, such as robot path planning, transport routing, and game playing. While classical algorithms can efficiently compute shortest paths, neural networks could be better suited to adapting these sub-routines to more complex and intractable tasks. As a step toward developing such networks, we hand-code and learn models for Breadth-First Search (BFS), i.e. shortest path finding, using the unified architectural framework of Neural Cellular Automata, which are iterative neural networks with equal-size inputs and outputs. Similarly, we present a neural implementation of Depth-First Search (DFS), and outline how it can be combined with neural BFS to produce an NCA for computing diameter of a graph. We experiment with architectural modifications inspired by these hand-coded NCAs, training networks from scratch to solve the diameter problem on grid mazes while exhibiting strong generalization ability. Finally, we introduce a scheme in which data points are mutated adversarially during training. We find that adversarially evolving mazes leads to increased generalization on out-of-distribution examples, while at the same time generating data-sets with significantly more complex solutions for reasoning tasks.[1]

1 INTRODUCTION

Pathfinding is a crucial sub-routine in many important applications. On a 2D grid, the shortest path problem is useful for robot path planning (Wang et al. [2011]) or in transportation routing (Fu et al. [2006]). Long paths (and quantities such as the diameter) are relevant for estimating photovoltaic properties of procedurally-generated microstructures for solar panels (Stenzel et al. [2016], Lee et al. [2021]), or the complexity of grid-based video game levels (Earle et al. [2021]).

Classical algorithms to solve pathfinding and related problems include the Bellman-Ford algorithm (Bellman [1958] Ford Jr [1956]) for finding the shortest path from a single source node to other nodes, Breadth-First Search (BFS) (Moore [1959], Merrill et al. [2012]), which models the connected nodes using a Dijkstra map (Dijkstra et al. [1959]), and Depth-First Search (DFS) (Tarjan [1972]), which explores the connections from each node sequentially.

Neural networks are increasingly being used for solving complex problems in the aforementioned applications involving pathfinding subroutines. Therefore, modeling classical pathfinding algorithms in "neurally plausible" ways could be advantageous for holistically solving these more complex problems. This approach has been explored before: for the shortest path problem, Kulvicius et al. [2021] construct hand-crafted neural networks to implement an efficient, distributed version of BFS.

On the other hand, we also know that the performance and generalization of neural networks depend heavily on their structure. Xu et al. [2019] posit the theory of *algorithmic alignment*, which is a measure of a network architecture's appropriateness for a reasoning task. If the network structure aligns with an algorithm for solving the target reasoning task, the network's sample complexity is lower. For example, the structure of the Bellman-Ford algorithm for shortest path finding aligns with Graph Neural Networks (GNNs) more than Multi-Layer Perceptrons (MLPs), and indeed GNNs are shown to generalize well on this task. More specifically, Definition 3.4 of Xu et al. [2019] asserts that networks are aligned when sub-modules of the network have a natural mapping onto sub-functions of the reasoning algorithm (e.g. when it is sufficient for each network submodule to learn the operation inside a for-loop in the target algorithm, instead of the for-loop itself).

[1]Code is available at https://anonymous.4open.science/r/pathfinding-nca-FEAD

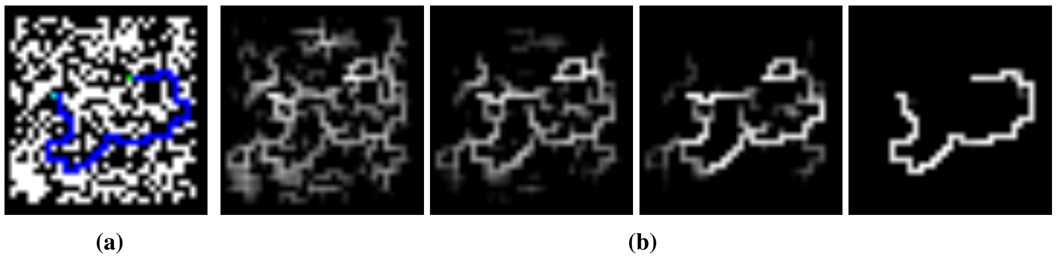

Figure 1: Learned pathfinding behavior. a) An example maze with the shortest path, where blue, green, dark blue, and black respectively represent source, target, path, and wall tiles. b) A learned model computes the shortest path on an out-of-distribution example. It first activates all traversable tiles, then strengthens activations between source and target, while gradually pruning away the rest.

Problem formulation. We focus on pathfinding in grid-based mazes with obstacles and empty tiles. While navigating the maze, one can move up, down, right, and left onto empty tiles. We consider two pathfinding problems: finding (i) the shortest path between fixed source and target (Fig. 1a), and (ii) the diameter, which is the longest shortest path between any pair of nodes.

An optimal method for the shortest path problem involves BFS, while an optimal method for the diameter problem includes BFS and DFS (Holzer and Wattenhofer [2012]). We follow the above line of work by implementing target algorithms as neural networks, and using them to propose architectural modifications to learning networks. We conceptualize *architectural alignment* as adjustments to network sub-modules that facilitate their ability to learn sub-functions of the target algorithm, when the mapping between sub-modules/functions is fixed.

Specifically, we showcase the architectural alignment of Neural Cellular Automata (NCAs) (Mordvintsev et al. [2020])—consisting of a single convolutional layer that is repeatedly applied to an input to iteratively produce an output of the same size—for pathfinding problems on grid-based mazes. NCAs are a natural choice for grid-based domains; like GNNs, they involve strictly local computation and are thus well-suited to similar problems. By the theoretical framework of Xu et al. [2019] (Definition 3.4), the lesser sample complexity of GNNs for pathfinding should apply equally to NCAs.

A summary of our contributions is as follows:

- We develop hand-coded NCAs for the shortest path finding problem (Sec. 3.1), which implement Dijkstra map generation (Sec. 2.1) and path extraction (Sec. 2.2). The latter is also necessary to extract the shortest path from the Dijkstra map generated by Kulvicius et al. [2021]. We thus demonstrate that NCAs can complete all the necessary sub-tasks for the shortest path problem.
- We provide an NCA implementation of DFS (Sec. 2.3), an essential component in the optimal, parallelized diameter-computing algorithm introduced by Holzer and Wattenhofer [2012], and outline how BFS- and DFS-NCAs can be combined to solve the diameter problem (Sec. 3.2).
- We suggest that NCA architectures can be further manipulated to align with the structure of hand-coded solutions for the problem in order to improve their performance, and we support our handcoding-inspired architectural modifications (Sec. 4) with experiments in Sec. 5.

Beyond motivating architectural alignment, these differentiable hand-coded models could act as fixed submodules or initialization schemes for larger hybrid architectures, potentially leading to more accurate and reliable learned solutions to more complex problems involving pathfinding.

It is worth noting that NCAs have been generalized to graphs by Grattarola et al. [2021] as Graph-NCAs, which can be seen to have at least the same representational capacity as convolutional NCAs on grids, while at the same time being applicable to arbitrary graphs. The GNNs trained on grid-maze domains in Tables 1, 10, and 11, are effectively Graph-NCAs. Using the message passing layer of, e.g., Graph Attention Networks (GATs, Veličković et al. [2018], studied in Table 11), Graph-NCAs can compute anisotropic filters, which are crucial for solving the diameter problem. Future work could investigate the performance of pathfinding-related problems on general graphs using Graph-NCAs.

2 HAND-CODED NETWORKS

We consider two problems: finding theshortest path and diameter. In Sec 2.1, we use NCAs to implement the generation of the Dijkstra activation map since it is the core component in finding the shortest path. In Sec 2.2 we implement path extraction using an NCA so that the shortest path can be extracted from the Dijkstra map using NCAs.

The optimal algorithm to find the diameter (Holzer and Wattenhofer [2012]) includes DFS, Dijkstra map generation, and path extraction. For the shortest path problem, we provide neural implementations of Dijkstra map generation and path-extraction, and in Sec 2.3 we present a hand-coded DFS NCA. We outline a neural implementation of the diameter algorithm using our hand-coded neural networks in Alg. 1, which we describe in Sec. 3.2.

We design our experiments in Sec 5 according to the modifications suggested by our hand-coded implementations, discussed in Sec 4. A detailed description of the hand-coded parameters and forward pass is left for Appendix A.2.

2.1 DIJKSTRA MAP GENERATION

We present our hand-coded NCAs to replicate the Dijkstra activation sub-process of BFS. Our approach is similar to Kulvicius et al. [2021], though we do not use max-pooling (relying only on ReLU activations), and we implement bi-directional BFS. We denote the model by $\mathrm{Dijkstra}(u)$ for the source node u, and present the full implementation details in A.2.1.

Our model maintains the following key channels: flood_s (flood_t) to denote if there is any possible path from the source (target) to the current node, and age to count the number of iterations for which any flood has been active at that tile. The idea is to flood any empty (i.e. non-wall) tile once an adjacent tile is flooded. Flood activations are binary-valued to represent if the flood has reached this tile from source/target; therefore, the activation is step function, which can be implemented using ReLUs. The age activation is integer-valued, starting at 0, and is incremented whenever the corresponding tile is flooded; therefore, a ReLU is the activation of this channel. Our model is bi-directional because the flood activations propagate from both source and target. We use the age channel to reconstruct the optimal path in our shortest path extraction NCA after flood_s and flood_t are connected.

2.2 SHORTEST PATH EXTRACTION

The Dijkstra activation map includes all tiles reached by either source or target floods, but one needs to identify the shortest path itself. The path can be reconstructed starting from the point(s) at which the two floods meet. (Note that there may be more than one possible shortest path and our model will simultaneously extract all possible shortest paths to the path channel, though ties could be broken using a similar directional queuing mechanism as will be described in our implementation of DFS.)

The flood variables flood_s and flood_t will meet in the middle of the shortest path, so when $\mathrm{flood}_s \times \mathrm{flood}_t$ becomes a positive number, the path channel will be active and path extraction will begin. There are four different channels, $\mathrm{path}_{i,j}$, with $(i,j) \in \{(1,2),(1,0),(0,1),(2,1)\}$, to detect the presence of path activation at adjacent tiles to the right, left, top, and bottom, respectively.

To determine if a tile u should be included in the path being extracted, we must check that one of its neighbors, v, is already included in the path, and that u was flooded directly prior to v during the Dijkstra activation. Therefore, assuming v is the $(i,j)^{\text{th}}$ neighbor of u, then $\mathrm{path}_{i,j}$ is active when the age of u, age_u, is equal to $\mathrm{age}_v + 1$ and $\mathrm{path}_v = 1$. We use a $\mathrm{sawtooth}$ to determine when these conditions are met at $\mathrm{path}_{i,j}$. Finally, we determine path channel of a given tile by summing over its $\mathrm{path}_{i,j}$ channels and passing through the step function. We denote this model by $\mathrm{PathExtract}(\cdot)$ with the input of Dijkstra map generated by $\mathrm{Dijkstra}(u)$ for the source node u.

2.3 DEPTH-FIRST SEARCH

While BFS operates in parallel, DFS must run sequentially. Therefore, it will use a stack, with last-in, first-out order, to store nodes that are on the unexplored "frontier," and return a node from the stack when the currently explored route reaches a dead end.

In order to flood the search tree one tile at a time, we prioritize neighbors based on their relative position. In our implementation, the priority decreases in the order of moving down, right, up, and left. We represent the sequential flood in the route channel, which is binary-valued. Similar to the path extraction model, we keep the directional routes in different channels as denoted in $\mathrm{route}_{i,j}$ with $(i,j) \in \{(1,2),(2,1),(3,2),(2,3)\}$, representing down, right, up and left moves and the main sequential route in route.

In this model, the first activation of the route is due to the source channel, and then any center tile's route is activated when it's the priority node of the neighbor with an active route. Therefore, the kernel size will change to 5×5, so the nodes can consider their neighbors' neighbors in order to determine whether they have priority in receiving an adjacent activation. The activation for $\mathrm{route}_{(i,j)}$ is given by a ReLU, and we calculate route by summing over $\mathrm{route}_{(i,j)}$ channels with the step activation, which is similar to directional paths, $\mathrm{path}(i,j)$.

We've shown that the route builds the DFS search tree. Here, we need to make sure that we stack our possible branches to continue from a different route when the current sequential route is stuck. Therefore, we define a binary-valued pebble channel to follow the current route, and we can use the existence of pebble to determine whether we need to pop the last-in tile from the stack. In order to represent the stack model, we declare three different channels: stack, which is binary-valued to denote if the tile is in the stack, $\mathrm{stack\ direction\ chan}$, which is $\in \{0.2, 0.4, 0.6, 0.8\}$ to keep the priority for the tiles that are added to the stack, and $\mathrm{stack\ rank\ chan}$, which is integer-valued to count the iteration time to keep them in the general order.

We note that the pebble channel is calculated by $\mathrm{route^t} - \mathrm{route^{t-1}}$ where t is the current timestep. We define a $since$ channel to represent the number of timesteps that have passed since pebble was active on the adjacent tile (which is added to the stack).

The activation of the stack happens when max-pooling the pebble channel over the entire maze returns 0, which means that route is stuck. This activates the min-pooling over the entire maze of $\mathrm{stack\ rank} + \mathrm{stack\ direction}$ (excluding tiles where this sums to 0). Min-pooling is used to determine the tile that was most recently added to the stack. This tile's stack activation is changed to 0, clearing the way for route activation. (If the same tile is added to the stack twice, we use a sawtooth to identify if any tile has $\mathrm{stack\ chan} = 2$, in which case we use skip connections to *overwrite* previous $\mathrm{stack\ rank}$ and $\mathrm{direction}$ activations.)

We emphasize that the DFS scheme outlined here can be implemented using convolutional weights, skip connections, max-pooling, and ReLU activations, and is thus differentiable. We denote the operation of this hand-coded neural network by $\mathrm{DFS}(u)$ for a starting node u.

3 Pathfinding by hand-coded neural networks

3.1 Computing the shortest path

Our goal is to compute the shortest path in the given maze when there is a defined source u. Thus, we first generate a Dijkstra map by the function call of $\mathrm{Dijkstra}(u)$, then, we extract the path from this Dijkstra map by calling $\mathrm{PathExtract}$ over this Dijkstra map. We can thus denote the shortest path finding routine conducted by our handcoded neural networks as $\mathrm{PathExtract}(\mathrm{Dijkstra}(\mathrm{u}))$. In practice, the networks' convolutional weights are concatenated, then the non-linearities and other computations specific to each routine are applied to the result in sequence.

3.2 Computing the diameter

In order to calculate the diameter, one could naively calculate the shortest path between all pairs of nodes and extract the largest path among them in $O(n^3)$ time. An asymptotically near-optimal method for calculating the diameter has been proposed by Holzer and Wattenhofer [2012], and includes BFS and DFS as subroutines and runs in $O(n)$. This algorithm relies on a parallelisable message-passing scheme.

In Alg. 1, we propose a neural implementation of the diameter algorithm. We assume that all nodes are connected in the graph. (Otherwise, the diameter routine would need to be called on an arbitrary node from each connected component.)

The algorithm starts with the DFS call on an initial randomly chosen node. While DFS is running, we make a call to the Dijkstra map generation routine, $\mathrm{Dijkstra}(u)$, whenever pebble is active at u.

To prevent the collision of the flooding frontiers from parallel Dijkstra routines, we must wait a certain number of timesteps before each new call. The pebble activation "moves" between timesteps in two ways: either directly between two adjacent tiles, or by jumping across tiles to the last on the stack because no adjacent tiles were visitable. In the first case, the waiting time is one, so that the new call to $\mathrm{Dijkstra}$ starts after the first step of the previous call. In the latter case, it needs to backtrack to the next node from the stack, so the waiting time should be equal to backtracking time $+1$. This ensures that each new flood is "inside" the previous one so there is no collision

Each cell u in the grid stores its age as $\mathrm{path\_max}$ when $\mathrm{Dijkstra}(u)$ returns, since in our $\mathrm{Dijkstra}$ implementation, the source has the largest age. When the DFS routine is completed, each $\mathrm{path\_max}$ corresponds to the longest shortest path in which the given tile is an endpoint. We calculate the diameter by taking the largest $\mathrm{path\_max}$, then extract the corresponding shortest path.

Algorithm 1: Diameter

```
$i = 0$, $\mathrm{prior} = \mathrm{null}$, $\mathrm{waiting\_time} = 0$, Choose a node $v$, start $\mathrm{DFS}(v)$
while DFS runs do
    if $u^{\mathrm{pebble}} == 1$ then
        if prior $! = $ null then $\mathrm{waiting\_time} = \left|\mathrm{prior}^{\mathrm{since}} - u^{\mathrm{since}}\right|$
        $\mathrm{prior} = u$, Wait $\mathrm{waiting\_time}$, $\mathrm{Dijkstra}(u)$
        if $\mathrm{Dijkstra}(v)$ ends then $v^{\mathrm{path\_max}} = v^{\mathrm{age}}$,
$\mathrm{PathExtract}(\mathrm{Dijkstra}(\arg\max(\mathrm{path\_max})))$
```

4 Architectural modifications

We focus on three sub-tasks via their hand-coded NCA implementations: Dijkstra map generation, path extraction, and DFS. For the shortest path problem, Dijkstra map generation and path extraction are necessary steps. This implementation illustrates the algorithmic alignment of NCAs and also suggests certain architectural modifications to increase their performance.

In our implementation of shortest path finding (Sec. 3.1), involving Dijkstra map generation and path extraction, we share weights between convolutional layers (each of which can be considered as a single iteration of the algorithm), where the spatial convolutional weights do not have any values in the corners. Additionally, we add a skip connection at each layer, feeding in the same one-hot encoding of the original maze, so that the model can reason about the placement of walls (and in particular avoid generating paths that move illegally through them) at each iteration of the algorithm. We also implement shortest path finding so as to be bi-directional, i.e., flowing out simultaneously from source and target nodes by increasing the number of channels. This halves the number of sequential steps necessary to return the optimal path while adding a small constant number of additional channels. Generally, we hypothesize that more channels could be leveraged—whether by human design or learned models—to make the algorithm return in a fewer number of iterations, offloading sequential operations (distinct convolutional layers) to parallelized ones (additional activations and weights to be processed in parallel).

			model	train	test		
			—	16x16	16x16		32x32
model	**n. layers**	**n. hid chan**	n. params	accuracies	accuracies	pct. complete	accuracies
GCN	**32**	**96**	9,600	37.89 ± 34.59	37.95 ± 34.65	18.24 ± 16.65	25.01 ± 23.75
		256	66,560	69.61 ± 38.92	69.36 ± 38.78	41.09 ± 22.97	43.92 ± 27.12
MLP	**64**	**96**	16,257,024	70.69 ± 3.29	13.91 ± 1.04	2.74 ± 0.20	0.00 ± 0.00
		256	42,799,104	52.92 ± 6.63	12.82 ± 1.78	1.94 ± 0.17	0.00 ± 0.00
NCA	**32**	**96**	86,400	**99.67**± 0.28	**96.79** ± 0.70	**96.26** ± 0.47	**84.75** ± 6.69
		256	599,040	79.68 ± 44.54	78.33 ± 43.79	78.27 ± 43.76	74.24 ± 41.55

Table 1: Shortest path problem – model architecture : For each architecture, we choose the minimal number of layers capable of achieving high generalization. NCAs with weight sharing generalize best, while GCNs generalize better than MLPs.

			model	train	test		
			—	16x16	16x16		32x32
model	**shared weights**	**n. hid chan**	n. params	accuracies	accuracies	pct. complete	accuracies
NCA	False	**32**	663,552	99.73 ± 0.31	94.96 ± 0.88	94.37 ± 0.56	58.95 ± 40.94
		48	1,437,696	59.90 ± 54.68	57.49 ± 52.49	57.26 ± 52.27	49.23 ± 44.96
		96	5,529,600	**99.74** ± 0.28	96.42 ± 1.09	96.90 ± 0.37	81.79 ± 4.36
		128	9,732,096	79.91 ± 44.67	77.65 ± 43.41	77.55 ± 43.35	67.83 ± 38.01
	True	**32**	10,368	58.09 ± 53.03	56.80 ± 51.86	56.54 ± 51.62	49.44 ± 45.16
		48	22,464	98.68 ± 0.44	**96.61** ± 0.33	**96.92** ± 0.09	**88.83** ± 1.78
		96	86,400	78.68 ± 43.99	77.64 ± 43.41	78.05 ± 43.64	72.21 ± 40.42
		128	152,064	39.57 ± 54.18	39.25 ± 53.75	39.38 ± 53.92	36.94 ± 50.58

Table 2: Shortest path problem – weight sharing: Sharing weights between layers can improve generalization while reducing the number of learnable parameters.

			model	train	test		
			—	16x16	16x16		32x32
model	**cut corners**	**n. hid chan**	n. params	accuracy	accuracy	pct. complete	accuracy
NCA	False	**48**	22,464	98.68 ± 0.44	96.61 ± 0.33	96.92 ± 0.09	88.83 ± 1.78
		96	86,400	78.68 ± 43.99	77.64 ± 43.41	78.05 ± 43.64	72.21 ± 40.42
		128	152,064	39.57 ± 54.18	39.25 ± 53.75	39.38 ± 53.92	36.94 ± 50.58
	True	**48**	12,480	97.43 ± 0.88	95.81 ± 0.91	95.41 ± 1.17	81.47 ± 10.00
		96	48,000	**98.97** ± 0.32	**97.61** ± 0.31	**97.83** ± 0.50	**90.25** ± 3.88
		128	84,480	79.12 ± 44.23	78.34 ± 43.80	78.57 ± 43.92	75.49 ± 42.22

Table 3: Shortest path problem – cutting corners: Ignoring diagonal relationships between grid cells allows for comparable performance with fewer parameters.

In Sec 5, we take inspiration from these observations and implementation tricks, investigating the effect of analogously constraining or augmenting a learning model. In particular, we consider an increased number of channels, weight sharing, and alternative convolutional kernel shape, and in several cases observe increased performance on the shortest path problem.

The diameter algorithm includes all three subroutines (Sec. 3.2). During DFS, we increase the kernel size from 3×3 to 5×5 to keep track of the edges' priorities. Also, DFS relies on max-pooling as a non-local subroutine, which suggests that it should be included in learned architectures to promote generalization (Xu et al. [2020]). Therefore, in Sec 5, we investigate the effects of increasing kernel size and adding max-pooling layers to the NCA network on the diameter problem.

5 Experiments

5.1 Experimental setup

For NCAs, we follow the network architecture of Mordvintsev et al. [2021], who train an NCA to imitate the style of texture images. [2] The NCA consists of one convolutional layer with a 3×3 kernel and padding of 1, followed by a ReLU activation.

For the GCN architecture, we follow the NCA architecture closely, but use a graph convolutional layer in place of a traditional convolutional one. To take full advantage of the graph neural architecture, we represent each maze as a sub-grid including nodes and edges between traversable (non-wall) tiles.

The MLP architecture comprises a series of dense layers followed by ReLU connections. First, the input maze (or intermediary activation) is flattened, then passed via a fully connected encoding layer to a smaller (256 node) activation, then back through another fully connected decoding layer to a large activation, which is finally reshaped back into the size of the input.

We compute the shortest path between source and target nodes using BFS and represent the target path as a binary array with the same width and height as the maze. Loss is computed as the mean squared error between a predicted path array and the target path array. The model's output is then clipped to be between 0 and 1.

For the sake of evaluation, accuracy (inverse loss) is normalized against all-zero output, which would achieve $\approx 97\%$ accuracy on the dataset. Accuracy can thus be negative when, e.g., a model predicts

[2]We exclude the RGB-specific pre-processing filters used in this work.

a path comprising a majority of non-overlapping tiles relative to the true path. Finally, we record whether, after rounding, the output perfectly matches the target path, i.e. the percentage of target paths perfectly completed, or *pct. complete* in tables.

Each model is comprised of a repeated sequence of identically-structured blocks (comprising a convolutional layer, a graph convolutional layer, or an encoder/decoder comprising two dense layers, in the case of NCAs, GCNs, and MLPs, respectively), with or without weight-sharing between them.

Maze is represented as a 2D one-hot array with 4 or 2 channels for the shortest path or diameter, respectively since there is no source and target in the diameter problem. We then concatenate a zero-vector with the same width and height as the maze and a given number of hidden channels. After the input passes through each layer of the model, it is returned as a continuous, multi-channel 2D array with a size equal to the input. We interpret an arbitrarily-chosen hidden channel as the model's predicted path output. After the last layer, we compute the mean-squared error loss of the 2D predicted path output with the ground-truth optimal path.

We use mini-batches of size 64. To feed batches to the GCN, we treat the 64 sub-grids containing the input mazes as disjoint components of a single graph. After each mini-batch, we use Adam (Kingma and Ba [2014]) to update the weights of the model. We train for $50,000$ updates for 5 trials.

The dataset comprises 10000 randomly-generated 16×16 mazes, which are generated by randomly placing empty, wall, source, and target tiles until the target is reachable by the source. The resulting paths are relatively simple, their path length is ≈ 9 tiles. We additionally test models on 32×32 mazes, which are generated as former and their mean path length ≈ 13 tiles.

We demonstrate the results to demonstrate the comparison of model architectures, the effects of the architectural modifications, and the importance of the data generation in the following subsections. We refer the reader to Sec. A.4 in Appendix.

5.2 Model architecture

In Table 1, we demonstrate that NCAs outperform GCNs and MLPs, and generalize better than them. This indicates that NCAs are well-aligned with pathfinding problems over grids. Also, GCNs generalize substantially better than MLPs, which fits with past work that has demonstrated the alignment of Graph Neural Networks with pathfinding tasks Xu et al. [2019].

The relatively poor performance of GCNs may seem at odds with past work by Xu et al. [2019]; Tang et al. [2020]. However, we train on a smaller dataset with more complex mazes for a shorter amount of time compared to earlier works. Also, the goal is to recover the optimal path itself as opposed to merely its length so it's a more complicated problem.

We note that our BFS implementation does not distinguish between a node's neighbors at different positions, and could thus be easily be adapted to use a GCN instead of an NCA. To learn this hand-coding, NCAs would need to learn more structure than GCNs, which come with this spatial symmetry built in. However, it is clear from Fig. 1 that our learned models are not directly performing our handcoded implementation. In particular, they appear to propagate slowly-diminishing activations out from the source and target nodes, progressively strengthening the value of nodes that connect source and target while weakening others. To produce this behavior, it may be important to know which neighbor provided the activation originally to prioritize neighbors on the receiving end. (Similarly, our hand-coded DFS-NCA uses spatial distinctions between neighbors to prioritize the distribution of activation among them.) But the GCN trained here is incapable of making these distinctions given that it applies the same weights to each neighbor and aggregates the results.

5.3 Shortest path

Following the modification suggested in Sec. 4, we investigate the performance after *weight sharing* between layers, and ignoring the corner weights, (*cutting corners*). In Table 2, we see that weight-sharing leads to the best performance while drastically decreasing the number of parameters. This agrees with our knowledge of known pathfinding algorithms, which repeatedly apply the same computations. Also, we demonstrate that increasing the number of hidden channels improves performance to a certain extent. This is reflective of a general trend in deep learning in which overparameterization leads to increased performance.

			model	train	test		
			—	16x16	16x16		32x32
model	**max-pool**	**n. hid chan**	n. params	accuracies	accuracies	pct. complete	accuracies
NCA	**False**	**96**	47,040	77.69 ± 43.43	66.48 ± 37.16	57.60 ± 32.21	-138.22 ± 204.08
		128	83,200	78.05 ± 43.63	66.89 ± 37.40	58.38 ± 32.65	-9.86 ± 12.18
	True	**96**	47,040	93.61 ± 0.41	85.74 ± 0.57	64.57 ± 2.07	**56.50** ± 35.32
		128	83,200	**95.72** ± 0.45	**86.54** ± 1.07	**71.82** ± 1.67	56.31 ± 26.80

Table 4: Diameter problem – max-pooling (with weight sharing and 3×3 kernels): Adding spatial and channel-wise max-pooling operations at each convolutional layer leads to increased generalization to large (32×32) out-of-distribution mazes when computing the diameter of a maze, which aligns with tasks involving the global aggregation of locally-computed information.

			model	train	test		
			—	16x16	16x16		32x32
model	**kernel size**	**cut corners**	n. params	accuracies	accuracies	pct. complete	accuracies
NCA	**3**	**False**	149,760	77.41 ± 43.27	67.20 ± 37.57	58.32 ± 32.60	27.47 ± 52.59
		True	83,200	95.72 ± 0.45	**86.54** ± 1.07	**71.82** ± 1.67	**56.31** ± 26.80
	5	**False**	416,000	**96.62** ± 0.96	76.89 ± 0.93	60.15 ± 1.78	21.85 ± 14.69
		True	216,320	96.30 ± 0.54	81.43 ± 0.45	68.72 ± 0.59	49.74 ± 2.58

Table 5: Diameter problem – kernel size/shape (with max-pooling and 128 channels): Smaller kernel shapes generalize best on the diameter task.

In Table 3, we examine the effect of modifying the kernel to ignore corners in each 3×3 patch, i.e. *cutting corners*. Across varying numbers of hidden channels, we observe comparable performance with and without this modification, despite having reduced the number of parameters by $4/10$. This supports the intuition from our hand-coded BFS implementation and suggests that diagonal neighbors provide little useful information when determining the next state of a given node when finding optimal paths.

5.4 Diameter

In the diameter problem, we again analyze *cutting corners* and *max-pooling*, as well as the effect of *kernel size*. In Table 4, we see that max-pooling has a significant effect on the models' performance and generalization on the diameter problem. Spatial max-pooling allows for the global aggregation of information computed locally at disparate points on the map. In our hand-coded DFS-NCA, max-pooling is used to *pop* frontier nodes from a stack so that we may traverse them in sequence. In the neural diameter algorithm, it also corresponds to the $\arg\max$ of the shortest paths that have been found in different connected components of the grid.

Table 5 suggests that simply increasing kernel size in convolutional layers tends to degrade the performance of NCAs on the diameter problem. This recalls the performance differences between MLP and GNN/NCA architectures observed in Table 1, in that MLPs, which observe spatially larger parts of the input maze at once are not robust to mazes outside of the training set. In one sense, this would seem to go against the intuition suggested by our hand-coded DFS-NCA, which uses 5×5 kernels. But very little of these larger patches are actually used in our hand-coded weights, and the potentially extraneous information they provide to a learning model may lead it to make spurious correlations.

Accordingly, in Table 5 we also examine the effect of *cutting corners* for cells that go unused in our handcoded DFS-NCA (refer to Table 2.3 for the weights). This increases the performance of 5×5 kernels despite resulting in fewer learnable parameters. Surprisingly, we also note that cutting the corner cells in 3×3 kernels similarly improves performance, suggesting that the diameter task may be feasible (or at least more learnable) when focusing on relationships between directly connected nodes, and leaving non-local computations to, e.g. a small number of max-pooling operations.

5.5 Adversarial data generation

Recall that we randomly generate a maze dataset in previous experiments. We now apply an evolutionary algorithm to incrementally alter the dataset during training. We evolve reasoning

			model – task	train		test		
			—	16x16		16x16		32x32
model	**env generation**	**n. hid chan**	n. params	sol. length	pct. complete	accuracies	pct. complete	accuracies
NCA – Shortest path finding	**False**	**96**	86,400	9.02 ± 0.00	78.53 ± 43.92	77.09 ± 43.10	76.84 ± 42.97	65.99 ± 37.70
		128	152,064	9.02 ± 0.00	**99.37** ± 0.24	97.44 ± 0.13	97.61 ± 0.20	90.06 ± 2.93
	True	**96**	86,400	21.39 ± 6.99	74.06 ± 41.44	79.18 ± 44.26	79.21 ± 44.28	78.12 ± 43.67
		128	152,064	**23.75** ± 1.36	92.92 ± 3.79	**98.88** ± 0.36	**99.34** ± 0.11	**97.17** ± 2.15
NCA – Diameter	**False**	**96**	84,672	24.09 ± 0.00	71.58 ± 40.13	66.15 ± 36.99	55.66 ± 31.15	-5.20 ± 41.28
		128	149,760	24.09 ± 0.00	**74.04** ± 41.39	67.20 ± 37.57	58.32 ± 32.60	27.47 ± 52.59
	True	**96**	84,672	26.06 ± 1.11	33.94 ± 19.12	70.47 ± 39.40	52.90 ± 29.61	**60.01** ± 33.55
		128	149,760	**27.43** ± 0.15	49.85 ± 4.07	**90.02** ± 0.44	**75.79** ± 1.57	30.02 ± 64.80

Table 6: Online adversarial data evolution (with 3×3 kernels, max-pooling, and weight-sharing): Adversarial evolution of mazes increases models' generalization ability, as well as the solution length of training mazes.

problems to maximize learned models' regret (here, their loss against the ground truth), in the same way, Parker-Holder et al. [2022] evolve game levels. When the model's loss falls below a certain threshold, we randomly select a batch of data points for mutation, apply noise to the data points (changing the state of some uniformly random set of tiles in the maze), and re-compute the ground-truth solution.

We then evaluate the model's performance on these new mazes (without collecting any gradient), ranking them by their fitness (the loss they induce). We then replace any of the least fit data points from the training set with new *offspring* mazes that are more fit. In Table 6, we see that adversarially evolving new data points in this way leads to increased generalization on both tasks. Additionally, this tends to increase the complexity of examples in the training set (by a factor of 2.5 on the pathfinding task).

5.6 Additional results and comparisons

NCAs tackle the shortest path problem with high accuracy on both training and test sets (Fig. 1). GCNs often produce outputs that are reasonable at a high level, though they tend to be blurrier (Fig. 2b). These models are slower to learn and do not appear to have converged in most of our experiments. MLPs, when confronted with the shortest path problem, will often reproduce many of the correct tiles, but leave clear gaps in the generated path (Fig. 2a). Instead of learning a localized, convolution-type operation at each layer, the MLP may be behaving more like an auto-encoder (owing to their repeated encoder/decoder block architecture), memorizing the label relative to the entire maze and reproducing it with relatively high accuracy but without preserving local coherence.

NCAs fail to perfectly generalize the diameter problem. They sometimes select the wrong branch toward the end of a largely correct path (Fig. 2e), which is a less crucial mistake. If there are two equivalent diameters, the model will sometimes activate both sub-paths.

6 Conclusion

In this paper, we introduce neural implementations of the shortest path finding and diameter problems. We posit that these hand-coded models can provide insight into learning more general models in more complex pathfinding-related tasks. We validate our claims by showing that architectural modifications inspired by hand-coded solutions lead to models that generalize better.

One limitation of our method is that it deals only with mazes defined on a grid. While our neural BFS model could be readily adapted to arbitrary graphs (i.e. translated from an NCA to a GCN architecture), our DFS implementation relies on the convolutional structure of NCAs. One could imagine re-implementing the sequential queuing logic of our DFS-NCA in a Graph-NCA, by using the message-passing layer of an anisotropic GNN such as a GAT. Such hand-coded solutions could be key to understanding how to scale the strong generalization ability exhibited by learned NCAs on complex grid mazes to similarly complex mazes on arbitrary graphs.

Future work may also benefit from these differential sub-modules directly, either using them to augment a learning model or using them as an adaptable starting point to further improve existing algorithms or fit them to a particular context.

REFERENCES

Huijuan Wang, Yuan Yu, and Quanbo Yuan. Application of dijkstra algorithm in robot path-planning. In *2011 Second International Conference on Mechanic Automation and Control Engineering*, pages 1067–1069, 2011. doi: 10.1109/MACE.2011.5987118.

L. Fu, D. Sun, and L.R. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers and Operations Research*, 33(11):3324–3343, 2006. ISSN 0305-0548. doi: https://doi.org/10.1016/j.cor.2005.03.027. URL https://www.sciencedirect.com/science/article/pii/S030505480500122X. Part Special Issue: Operations Research and Data Mining.

Ole Stenzel, Omar Pecho, Lorenz Holzer, Matthias Neumann, and Volker Schmidt. Predicting effective conductivities based on geometric microstructure characteristics. *AIChE Journal*, 62(5): 1834–1843, 2016.

Xian Yeow Lee, Joshua R Waite, Chih-Hsuan Yang, Balaji Sesha Sarath Pokuri, Ameya Joshi, Aditya Balu, Chinmay Hegde, Baskar Ganapathysubramanian, and Soumik Sarkar. Fast inverse design of microstructures via generative invariance networks. *Nature Computational Science*, 1(3):229–238, 2021.

Sam Earle, Justin Snider, Matthew C Fontaine, Stefanos Nikolaidis, and Julian Togelius. Illuminating diverse neural cellular automata for level generation. *arXiv preprint arXiv:2109.05489*, 2021.

Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. *ACM Sigplan Notices*, 47(8):117–128, 2012.

Edsger W Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2): 146–160, 1972.

Tomas Kulvicius, Sebastian Herzog, Minija Tamosiunaite, and Florentin Wörgötter. Finding optimal paths using networks without learning–unifying classical approaches. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–11, 2021. doi: 10.1109/TNNLS.2021.3089023.

Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2019.

Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364, 2012.

Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 5(2):e23, 2020.

Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Learning graph cellular automata. *Advances in Neural Information Processing Systems*, 34, 2021.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Keyulu Xu, Mozhi Zhang, Jingling Li, Simon Shaolei Du, Ken-Ichi Kawarabayashi, and Stefanie Jegelka. How neural networks extrapolate: From feedforward to graph neural networks. In *International Conference on Learning Representations*, 2020.

Alexander Mordvintsev, Eyvind Niklasson, and Ettore Randazzo. Texture generation with neural cellular automata. *arXiv preprint arXiv:2105.07299*, 2021.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Hao Tang, Zhiao Huang, Jiayuan Gu, Bao-Liang Lu, and Hao Su. Towards scale-invariant graph-related problem solving by iterative homogeneous graph neural networks. *arXiv preprint arXiv:2010.13547*, 2020.

Jack Parker-Holder, Minqi Jiang, Michael Dennis, Mikayel Samvelyan, Jakob Foerster, Edward Grefenstette, and Tim Rocktäschel. Evolving curricula with regret-based environment design. *arXiv preprint arXiv:2203.01302*, 2022.