Recursive Algorithmic Reasoning

Jonas Jürß[*]
University of Cambridge
jj570@cl.cam.ac.uk

Dulhan Jayalath[* †]
University of Oxford
dulhan@robots.ox.ac.uk

Petar Veličković
Google DeepMind
petarv@deepmind.com

Abstract

Learning models that execute algorithms can enable us to address a key problem in deep learning: generalizing to out-of-distribution data. However, neural networks are currently unable to execute recursive algorithms because they do not have arbitrarily large memory to store and recall state. To address this, we (1) propose a way to augment graph neural networks (GNNs) with a stack, and (2) develop an approach for sampling intermediate algorithm trajectories that improves alignment with recursive algorithms over previous methods. The stack allows the network to learn to store and recall a portion of the state of the network at a particular time, analogous to the action of a call stack in a recursive algorithm. This augmentation permits the network to reason recursively. We empirically demonstrate that our proposals significantly improve generalization to larger input graphs over prior work on depth-first search (DFS).

1 Introduction

Mimicking classical algorithms by executing them with neural networks can bring many of the benefits of reasoning to black-box supervised learning. One of these benefits is taking advantage of the guarantees that algorithms provide. Classical algorithms are guaranteed to be correct no matter the size of their input. For example, sorting algorithms are correct regardless of the length of the input array. However, neural networks provide no such confidence—they will usually not be correct for problem instances larger than those they were trained on (failing to generalize out of distribution). If neural networks could learn to reason like algorithms, they could gain the substantial generalization properties that algorithms posses [1, 2].

On the flip side, the use of neural networks for reasoning can bring speed and redundancy benefits that classical algorithms are unable to provide. As first demonstrated by Li et al. [3], neural networks that mimic algorithms can outperform hand-coded solutions, in terms of number of steps, in limited problem instance sizes. More recently, Numeroso et al. [4] showed that, by reasoning in continuous space, neural networks can even learn to execute algorithms when some input features are missing. Consequently, neural networks also have clear advantages over classical algorithms.

Given this dichotomy, recent work in *neural algorithmic reasoning (NAR)* [2] studies how to gain the benefits of both classical algorithms and neural networks by executing algorithms using neural nets. Many of these algorithms are naturally amenable to graph representations as they can be viewed as manipulations of sets of objects and the relations between them [2, 5]. Hence, contemporary approaches in NAR train GNNs with a recurrent state to execute algorithms by predicting the state of the algorithm across time steps [1, 5]. For example, given the insertion sort algorithm, an input array can be represented as a chain of connected nodes, where at each time step, a GNN is applied to evolve the node embeddings. These node embeddings can predict the algorithm's state at this step, such as the position of the insertion pointer. Thus, the GNN mimics the steps of the algorithm.

While this paradigm allows GNNs to execute a range of classical algorithms, they cannot execute *recursive* algorithms (e.g., DFS) in general. This is because reasoning recursively requires a neural

---

[*]Equal contribution. † Work done while author was at the University of Cambridge.
Our code is available at https://github.com/DJayalath/gnn-call-stack/.

network to have memory at least large enough to store as many states as the maximum recursion depth of the problem. As a typical neural network is fixed in size, it cannot store all the states required to reason about an arbitrarily large recursive problem. Delétang et al. [6] empirically show that only networks with structured memory (e.g., a stack) generalize on context-sensitive tasks such as DFS.

To address this fundamental issue, we propose a framework for augmenting GNNs with stack memory. Inspired by call stacks in computer programs, this augmentation enables the network to learn how to save and recall states. In addition, we identify several key improvements for sampling algorithm trajectories in the CLRS-30 algorithmic reasoning benchmark [5]. These improvements allow the network to more closely structurally resemble a recursive algorithm.

We test our framework by implementing two methods of augmenting GNNs with a stack. We evaluate these approaches on the benchmark, empirically observing that our stack-based methods outperform standard GNNs (including the work in CLRS-30) on out-of-distribution generalization. Moreover, through a set of ablation experiments, we find support for our suggested improvements and discover additional modifications which further improve out-of-distribution generalization performance.

Our insights may be practical beyond DFS. The execution path of a recursive function's call graph is precisely a depth-first search. Therefore, DFS can in principle be used to express all other recursive algorithms given that the execution path is known upfront. Consequently, we believe that our analysis will be beneficial for algorithmic reasoning across recursive problems where this is the case. Nevertheless, we also more generally demonstrate how some of our insights can be applied to recursive algorithms that are not DFS. Additionally, we note that by supporting recursive reasoning, we enable the execution of looping control flows which have previously been difficult for neural networks to learn [3]. By instead realizing them as tail recursion, they can be executed under our architecture.

Our main contributions are:

1. A novel neural network architecture that uses a stack to learn to save and recall state exactly; this architecture significantly outperforms previous work [5], while using less memory, on out-of-distribution generalization when learning DFS.

2. Improvements to the sampling of algorithm trajectories which lead to closer alignment of GNNs in NAR with recursive reasoning.
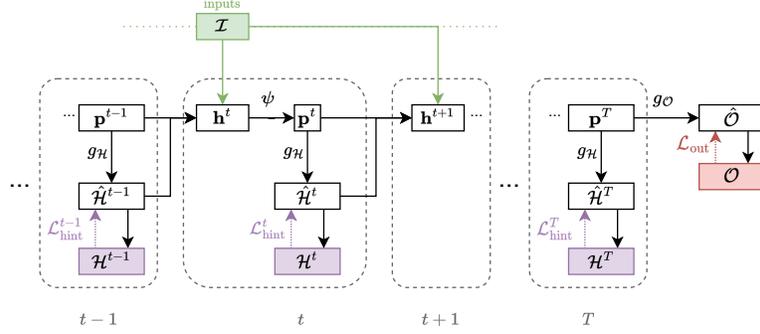
2 Background

2.1 The CLRS Algorithmic Reasoning Benchmark

Towards the goal of unified evaluation in NAR, Velickovic et al. [5] introduced CLRS-30: a benchmark for evaluating GNNs on algorithmic tasks *and* a standardized neural model for algorithmic reasoning. CLRS-30 measures NAR performance on a set of 30 curated algorithms, aligning closely to the definitions in Cormen et al. [7]'s foundational algorithms textbook: *Introduction to algorithms*.

The neural model represents the inputs and outputs of an algorithm (and the relations between them) as a graph $G = (V, E)$. At each step of training, CLRS-30 provides ground truth values for the state of variables in an algorithm—referred to as *hints* [5, 8]. The *left* and *right* pointers in the quicksort algorithm are examples of hints. The network learns to predict the state of these hints, which are encoded as vectors, at each step of computation. By learning to predict hints, the network's reasoning may align more closely with the computation of the algorithm.

Figure 1 provides a high-level unfolded view of the recurrent steps in the CLRS-30 neural model. The inputs to one step are node inputs $\mathbf{x}_i$, edge inputs $\mathbf{e}_{ij}$, and graph inputs $\mathbf{g}$. These inputs are defined by the algorithm and hints are included as part of them (where hints can belong to nodes, edges, or the graph). They are encoded with linear layers f into node, edge, and graph features $\mathbf{h}_i, \mathbf{h}_{ij}, \mathbf{h}_g \in \mathbb{R}^{d_\mathbf{h}}$ where

$$\mathbf{h}_i^t = f_n(\mathbf{x}_i^t) \qquad \mathbf{h}_{ij}^t = f_e(\mathbf{e}_{ij}^t) \qquad \mathbf{h}_g^t = f_g(\mathbf{g}^t), \tag{1}$$

Figure 1: Recurrent steps in CLRS-30. The processor network takes the features $\mathbf{h}^t$ and produces the processed features $\mathbf{p}^t$. These are used to predict the hints $\hat{\mathcal{H}}^t$ for that step. A loss is calculated between the ground truth hints and the predicted hints. The hints can optionally be teacher forced with some probability. The predicted hints are accumulated with the processed features and algorithm inputs to form the encoded features $\mathbf{h}^{t+1}$ for the next step. In the final step, the processed features are used to predict the output $\hat{\mathcal{O}}$. A loss is calculated between this and the ground truth output of the algorithm.

and $d_{\mathbf{h}}$ defines the dimensionality of these features[†]. The features are passed through a processor network ψ (a GNN) which outputs processed node and edge features $\mathbf{p}_i, \mathbf{p}_{ij} \in \mathbb{R}^{d_{\mathbf{h}}}$ such that

$$\mathbf{p}_i^t, \mathbf{p}_{ij}^t = \psi(\mathbf{h}_i^t, \mathbf{p}_i^{t-1}, \mathbf{h}_{ij}^t, \mathbf{h}_g^t) \tag{2}$$

where $\mathbf{p}_i^{t-1}$ is a recurrent state carried forward from the previous time step. These processed features are decoded to predict hints

$$\hat{\mathcal{H}}^t = g_{\mathcal{H}}(\{\mathbf{p}_i^t \mid i \in V\}, \{\mathbf{p}_{ij}^t \mid (i, j) \in E\}) \tag{3}$$

where $g_{\mathcal{H}}$ is the *hint decoder*. In the final step T, the neural model also predicts the output of the algorithm

$$\hat{\mathcal{O}} = g_{\mathcal{O}}(\{\mathbf{p}_i^T \mid i \in V\}, \{\mathbf{p}_{ij}^T \mid (i, j) \in E\}) \tag{4}$$

where $g_{\mathcal{O}}$ is the *output decoder*. If this is not the final step, hint predictions are aggregated with the algorithm inputs in the following step to form the next inputs $\mathbf{x}_i^{t+1}$, $\mathbf{e}_{ij}^{t+1}$, and $\mathbf{g}^{t+1}$. In training, hints can optionally be teacher forced (i.e., predicted hints can be substituted for ground truth hints during training) with some probability. Finally, the processed node embeddings $\mathbf{p}_i^t$ will be passed to the next step as the new recurrent state.

2.2 Depth-First Search in CLRS-30

Every algorithm is associated with a set of hints that should be predicted at each step. In CLRS-30, the DFS algorithm (Appendix E) has the hints described in Table 1. In this algorithm, each hint is encoded for each node. For example, there is a pointer to the predecessor of every node in the graph. We call these types of hints *per-node* hints as they have a value for every node. In other algorithms, a hint can be associated with the whole graph (a *graph* hint). As an example, the *min* pointer in binary search is a graph hint. It does not belong to any particular element of the input representation, but is instead shared between all of them—it is a property of the whole graph.
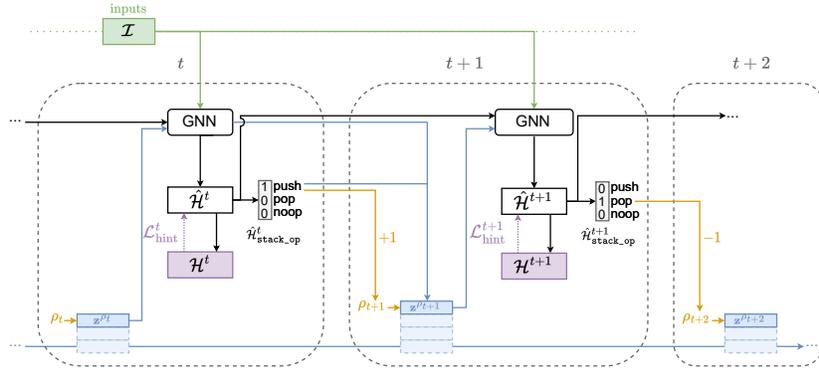
3 Augmenting A GNN With A Stack

Recursive algorithms typically require storing state in a call stack, executing the recursive call, and finally restoring this state to complete the recursion step. To support similar reasoning, our method adds stack memory to the processor network described in Section 2.1, enabling the network to push

---

[†]This is the typical case. Node, edge, and graph features can also be given different dimensions.

Table 1: Hints provided by CLRS-30 for the depth-first search algorithm. All hints (except *time*) are per-node hints. Hints with * correspond to a variable in the DFS algorithm given in Appendix E.

Hint	Explanation	Graph or Per-Node?
π_h*	Pointer to predecessor for each node	Per-Node
*color**	Color of each node	Per-Node
d*	Time of discovery for each node	Per-Node
f*	Time of finalization for each node	Per-Node
s_{prev}	Pointer to previous node for each node	Per-Node
s	Current node	Per-Node
u*	Node being explored	Per-Node
v*	Node to be explored	Per-Node
s_{last}	Last node explored for each node	Per-Node
*time**	Time step	Graph

Figure 2: Pushing and popping states with a stack-augmented GNN. The input to the network is the state at the top of the stack, the hints, and the algorithm inputs. Note that we do not pass the recurrent state to the GNN (Section 4.2). The network predicts the next processed features (which form the stack element), stack operation, and next hints. If the operation is a push, the next stack element is formed and placed on the stack; if it is a pop, the current top stack element is discarded.

and pop latent states. In addition to arbitrary-sized read-write memory, this provides an inductive bias towards storing and recalling state like a call stack.

To allow the network to use the stack, we first add a one-hot encoded graph hint

$$\mathcal{H}_{\text{stack\_op}}^t \in \{\texttt{push}, \texttt{pop}, \texttt{noop}^{\ddagger}\}$$

denoting the stack operation that the target algorithm performs at step t. The ground truth for this hint is push when entering a recursive call in the target algorithm, pop when returning from one, and noop otherwise. At inference time, when the network predicts this hint, latent states are pushed/popped from a stack, simulating a call stack during the execution of a recursive algorithm.

To utilize the stack operation predictions, we introduce a stack at step t as $\mathcal{S}^t$, which is composed of a sequence of stack elements $\mathbf{z}^0, \ldots, \mathbf{z}^{\rho_t}$ where ρ_t indicates the number of elements on $\mathcal{S}^t$. We start with $\rho_0 := 0$ and define $\mathbf{z}^0 := \mathbf{0}$.

The elements that are pushed to the stack are defined by the type of stack. We introduce two types: a stack for every node whose elements are processed node features (a *node-wise* stack), and a single stack for the graph for which elements are some pooled encoding of the node features (a *graph* stack). Figure 2 provides a high-level demonstration of how the stack is used and how stack usage is learned. Stack operations are supervised such that the network learns to push and pop at precisely the same times as the recursive algorithm (i.e., when state needs to be saved, and when it needs to be recalled).

---

‡Indicates no stack operation. The state of the stack is unmodified.

3.1 Node-Wise Stack

To store one element per node $i \in V$ we define $\mathbf{z}_i^{\rho_t} \in \mathbb{R}^{d_{\text{stack}}}$ to be the top stack element corresponding to node i at step t. Depending on the predicted operation $\hat{\mathcal{H}}_{\text{stack\_op}}^t$ we can then update the stack for step $t+1$ as follows:

$$\rho_{t+1} = \begin{cases} \rho_t + 1, \text{ if } \texttt{push} \\ \max\{\rho_t - 1, 0\}), \text{ if } \texttt{pop} \\ \rho_t, \text{ if } \texttt{noop} \end{cases}$$

$$\mathbf{z}_i^{\rho_{t+1}} = \begin{cases} \phi_{\text{value}}(\mathbf{p}_i^t), \text{ if } \texttt{push} \\ \mathbf{z}_i^{\max\{\rho_{t-1}, 0\}}, \text{ if } \texttt{pop} \\ \mathbf{z}_i^{\rho_t}, \text{ if } \texttt{noop} \end{cases}$$

Here, $\phi_{\text{value}} : \mathbb{R}^{d_{\mathbf{h}}} \to \mathbb{R}^{d_{\text{stack}}}$ denotes a (potentially learnable) function to decide which information to put on the stack. In each step t, we concatenate $\mathbf{z}_i^{\rho_t}$ to the initial node embeddings that serve as input to the GNN ψ given by $\mathbf{h}_i^t$ and optionally $\mathbf{p}_i^{t-1}$ (see Section 4.2).

Notably, using the top of the stack as an input to the network is effectively the same as providing a dynamic skip connection across time. As a result, we mitigate vanishing gradient issues because we do not need to backpropagate through intermediate time steps between when the state was first pushed and the current time.

3.2 Graph-Level Stack

We also consider a graph-level stack. In this case, the stack element $\mathbf{z}^{\rho_t} \in \mathbb{R}^{d_{\text{stack}}}$ is a vector of fixed size. The pop and noop operations are similar to the node-wise stack. In the case of a push operation, we update the stack with

$$\mathbf{z}^{\rho_{t+1}} := \bigoplus_{i \in V} \phi_{\text{value}}(\mathbf{p}_i^t) \tag{5}$$

where $\oplus$ is some permutation-invariant aggregation. In step t, the top stack element $\mathbf{z}^{\rho_t}$ is concatenated to the graph features $\mathbf{h}_g^t$. We use a 2-layer MLP for our value network ϕ_{value}. As an alternative, we also explore taking the first d_{stack} entries of the node embedding such that

$$\phi_{\text{value}}(\mathbf{p}_i^t) := (\mathbf{p}_i^t)_{0:d_{\text{stack}}} \tag{6}$$
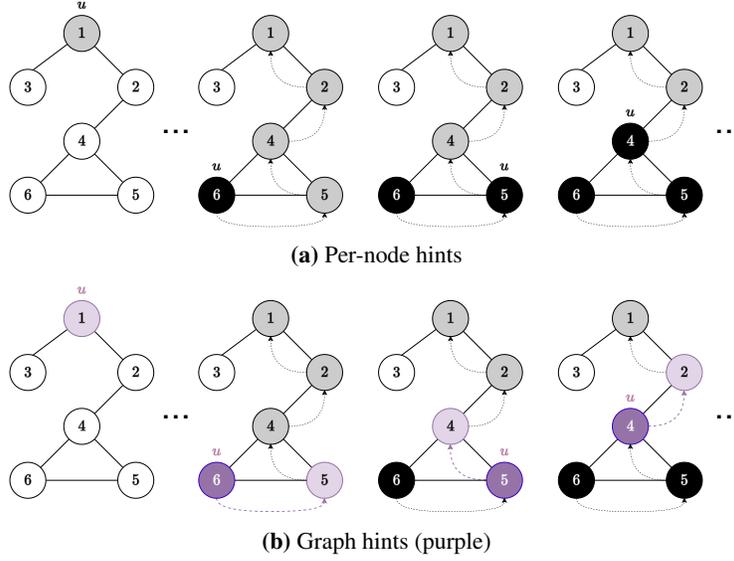
where $d_h \geq d_{\text{stack}}$.

4 Stacks Are Not All You Need

4.1 Recursive Problems Require Additional Memory

The implementations of many recursive algorithms in CLRS-30 are not truly recursive. For instance, in DFS, hints, such as the predecessor π, are present for *each* node (per-node hints). Together, these per-node hints provide *global* information about the entire computation state. This information is already sufficient to deduce what to do in the next algorithm step *without additional memory*. As demonstrated by Figure 3, when per-node hints are used for critical variables in the algorithm, a stack is not required. Hence, the algorithm can be executed without recursive steps.

In contrast, the recursive implementation of DFS described by Cormen et al. [7] (Appendix E) uses variables that would be graph hints rather than per-node hints. For example, the algorithm has access to only the predecessor of the *current* node rather than the predecessors of all nodes. The use of per-node hints in the CLRS-30 implementation is therefore problematic as it implies that the network will not be closely aligned with a recursive algorithm.

To remedy this, we propose using graph hints with the aim of achieving closer algorithmic alignment with recursion. As a demonstrative example, we modify the DFS implementation in CLRS-30 to use hints based directly on the variables in the algorithm (Table 2), where all except *color* are graph hints rather than per-node hints. These new hints are relative only to the current node being explored. The *color* hint remains a per-node hint as this information is required when looping over the neighbors of a node in the algorithm (see line 5 of DFS-Visit() in Appendix E). This configuration of hints is

(a) Per-node hints

(b) Graph hints (purple)

Figure 3: Per-node hints are sufficient to determine the next step of DFS. Some steps of the DFS algorithm are shown on a graph in (a). The nodes are colored white (unvisited), gray (being explored), or black (visited) according to the *color* variable in DFS (see Appendix E). The back-pointers indicate the predecessor of a node (π). The current focus of the algorithm is u. After reaching the state in the second step, the algorithm backtracks to the predecessor, Node 5. Finalizing Node 5 as it has no unexplored neighbors, the algorithm backtracks to Node 4 and so on. It knows which node to go to next, because it has the current node u, the predecessor pointers π for all nodes, and the colors of all nodes. Therefore, per-node hints are enough to determine the next step of DFS without any additional state. In (b), we show how this would look with our proposed hints (Table 2) which provide only information for u, the predecessor pointer for u, and the colors of all nodes. Only nodes and pointers colored in purple are known at each step. The predecessor pointer for u is insufficient to backtrack more than one node as the algorithm can only backtrack from Node 6 to 5 before running out of purple-shaded back-pointers. The algorithm needs to recall the state relevant to Node 5 to backtrack further (as shown in the third step). Therefore, state recall is required to solve DFS with our hints.

more similar to the state pushed to the call stack in a truly recursive DFS algorithm as we only save the state related to the node we are currently exploring. With this change, the network will not have enough information to execute DFS without storing and restoring state as the DFS algorithm does. A similar procedure can be applied to recursive algorithms in general by ensuring that variables which would be pushed onto the call stack in a recursive call are always graph hints.

Table 2: Modified hints for the depth-first search algorithm. All hints except *color* are graph hints. All except *stack_op* correspond directly to the algorithm in Appendix E.

Hint	Explanation	Graph or Per-Node?
u	Node being explored	Graph
u_π	Predecessor of node	Graph
u_d	Time when node discovered	Graph
u_f	Time when node finalized	Graph
u_v	Neighbor to be explored	Graph
color	Color of all nodes	Per-Node
time	Time step	Graph
stack_op	Stack operation (push/pop/noop)	Graph

4.2 Recurrent States Can Encode Global Information

Removing per-node hints is not enough as they can also be learned implicitly in the node-wise recurrent state $\mathbf{p}_i^{t-1}$. Whatever information can be carried from a hint, can also be learned as part of this hidden state. Therefore, the network could learn a representation similar to the previous unmodified hints through this state. In such a case, the network would not need to save or restore state once again. Consequently, we do not pass information in the recurrent state to the GNN and enforce that it relies only on the information on our stack. Hence, we modify Equation 2 to

$$\mathbf{p}_i^t, \mathbf{p}_{ij}^t = \psi(\mathbf{h}_i^t, \mathbf{h}_{ij}^t, \mathbf{h}_g^t) \tag{7}$$
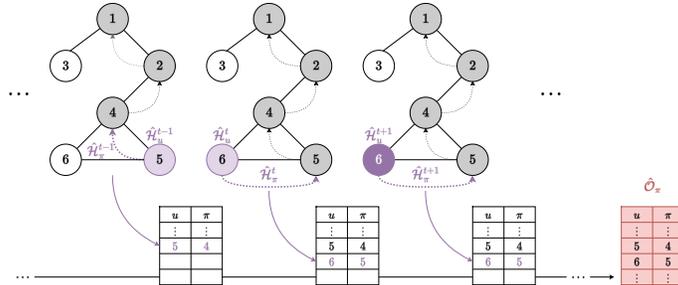
Similarly, a node-wise stack could act like a node-wise recurrent state if the stack were pushed to every step. In this case, the top of the stack would provide a recurrent state to the GNN which it could use to learn per-node hints implicitly. We avoid this problem as our stack is explicitly supervised through ground truth stack operations sampled from the target algorithm which would never require pushing state to the call stack in every step.

4.3 Recursive Algorithms Generate Results Sequentially

In CLRS-30, the complete output of the algorithm (e.g., the sequence of nodes in a search) is predicted at once from the processed features. This is not how recursive algorithms typically work. A DFS algorithm, for example, will generate the result by outputting each new node found as the search explores the graph. Moreover, since we use the top stack element and node embeddings to predict the output of the algorithm, to predict the output in the way that CLRS-30 does it, we need to memorize all the nodes found by the search as part of the stack element, introducing a memory bottleneck that will degrade performance as the number of nodes in the graph grows. Therefore, we modify CLRS-30 to tabulate predicted outputs for each node as the network executes the algorithm and, at the end, use this table to collect these results together into a single final output. This is more like how a typical recursive algorithm generates its result. We provide an example of this for DFS in Figure 4. This method can also be applied to other recursive algorithms. For example, in quicksort, the final position of the pivot element can be tabulated after each recursive call.

5 Results & Discussion

In Table 3, we summarize our results on different network configurations. We refer to test accuracy on larger graphs (96 nodes) as out-of-distribution (OOD) performance. While the setup proposed by Ibarz et al. [9] (Experiment 1) achieves near-perfect test accuracy on our dataset, it fails to generalize to larger graphs. In contrast, our method achieves similar in-distribution performance, and at 73% accuracy for a graph-level stack and 100% for a node-wise stack, *drastically better OOD generalization performance* (Experiment 2 and 10) while using *less memory* (Appendix D).

Figure 4: Collecting the output from predecessor hint predictions. In each step, the network only predicts graph hints like the current node $\hat{\mathcal{H}}_u^t$ and its predecessor $\hat{\mathcal{H}}_\pi^t$ marked in purple (dark for black nodes, light for gray nodes). To collect the final output, we maintain a table during execution. In each step, we take the entry of the predicted current node $\hat{\mathcal{H}}_u^t$ and overwrite its predecessor with the predicted predecessor $\hat{\mathcal{H}}_\pi^t$. Predecessors of nodes that are not purple are only shown for context. Our network does not have access to the corresponding hints.

Table 3: Call stacks improve generalization performance. Test accuracy is measured for graphs with 32 (in-distribution) and 96 (out-of-distribution) nodes respectively. The train and validation set contain a mix of graphs of up to 32 nodes as described in Appendix A. *Ours* denotes the setting where hints are mostly graph hints (Section 4.1), with various ablations. We report the mean and standard deviation of test accuracy over three runs at the point of best validation accuracy (early stopping). For complete implementation details, see Appendix C. Our node-wise call stack configuration achieves the best out-of-distribution generalization performance.

		Graph-Level stack (Sec. 3.2)	Node-Wise stack (Sec. 3.1)	Hidden state p_i^{t-1} (Sec. 4.2)	Output collection (Sec. 4.3)	Teacher forcing (50%)	ϕ_{value} learned (Eq. 6)	Attention (App. B)	Test Acc. (32 nodes)	Test Acc. (96 nodes)
Ibarz et al.	(1)	N/A	N/A	✓	✗	✓	N/A	N/A	$99.79^{\pm 0.20}$	$53.92^{\pm 14.06}$
	(2)	✓	✗	✗	✓	✓	✓	✗	$98.00^{\pm 0.60}$	$73.00^{\pm 6.31}$
	(3)	✗	✗	✗	✓	✓	N/A	N/A	$65.33^{\pm 4.69}$	$72.88^{\pm 7.47}$
	(4)	✓	✗	✓	✓	✓	✓	✗	$\mathbf{100.00}^{\pm \mathbf{0.00}}$	$82.19^{\pm 1.07}$
	(5)	✗	✗	✓	✓	✓	N/A	N/A	$\mathbf{100.00}^{\pm \mathbf{0.00}}$	$78.65^{\pm 5.44}$
Ours	(6)	✓	✗	✗	✗	✓	✓	✗	$50.54^{\pm 3.90}$	$25.08^{\pm 0.97}$
	(7)	✓	✗	✗	✓	✗	✓	✗	$71.73^{\pm 2.14}$	$43.88^{\pm 15.73}$
	(8)	✓	✗	✗	✓	✓	✗	✗	$97.54^{\pm 0.83}$	$67.42^{\pm 3.67}$
	(9)	✓	✗	✗	✓	✓	✓	✓	$92.27^{\pm 5.04}$	$49.40^{\pm 2.28}$
	(10)	✗	✓	✗	✓	✓	✓	N/A	$\mathbf{100.00}^{\pm \mathbf{0.00}}$	$\mathbf{100.00}^{\pm \mathbf{0.00}}$
	(11)	✗	✓	✓	✓	✓	✓	N/A	$\mathbf{100.00}^{\pm \mathbf{0.00}}$	$99.79^{\pm 0.29}$

Experiment 7 shows that teacher forcing is required to achieve the generalization performance we see. In line with a similar observation by Ibarz et al. [9], DFS is one of the algorithms in CLRS-30 that benefits from teacher forcing.

In Experiment 3, we note that removing the stack only has a minor effect on the the OOD performance but significantly reduces in-distribution accuracy. We hypothesize that this is the performance that can be achieved when only the hints are provided. There is no additional information propagation in terms of stack or hidden state. The main generalization benefit stems from turning our per-node hints into graph hints and collecting outputs as described in Section 4. This is a result of a significant improvement in alignment with the original DFS algorithm.

Notably, not learning the value network ϕ_{value} (Equation 6), does not noticeably impact performance (Experiment 8). This could indicate that learning an encoding of the processed features is unnecessary as the GNN ψ, which processes features, is able to learn an effective encoding by itself. In addition, as demonstrated by Experiment 6, collecting the outputs is a crucial component of aligning with the DFS algorithm as it has a significant impact on accuracy.

While we adjusted the DFS hints and output collection in a way that theoretically makes a graph-level stack sufficient, learning to store the relevant information for each node appears to be easier than also learning which node to focus on. This reflects in the fact that adding a (per-node) recurrent state to the graph-level stack (similar to Ibarz et al. [9] in Experiment 1) yields another significant boost in performance (Experiment 4). Removing the graph-level stack only has a minor effect (Experiment 5).

Based on this insight, we evaluate a network augmented with a node-wise stack as described in Section 3.1. We use the same graph hints and outputs as described in Section 4. This allows us to propagate per-node information while maintaining the inductive bias of a call stack. **We achieve perfect test accuracy in-distribution as well as on larger out-of-distribution graphs** with this modification (Experiment 10). When reintroducing the recurrent state, the network learns to use the node-wise call stack and achieves only slightly worse generalization accuracy (Experiment 11).

As our network is essentially a graph RNN, it faces the same forgetting issues as other recurrent networks [10, 11]. The stack resolves these forgetting issues, which are caused by using a node-wise

hidden state instead of a stack as in Experiment 5. Our method enables the network to perfectly recall state from more than one step in the past—a particularly practical feature when dealing with problems of high recursion depth.

Our modifications make the implementation in CLRS-30 significantly more closely aligned with the recursive DFS algorithm defined by Cormen et al. [7]. How has previous work also achieved strong empirical *in-distribution* results on DFS without this close alignment? Noting that memoization does not change expressiveness, DFS can be expressed as a dynamic programming (DP) problem. Given the structural alignment between GNNs and DP (shown by Xu et al. [1]), in prior work, the hints were sufficient to allow the algorithm to be solved as DP. It is perhaps because other approaches do not conform as closely to DP as our approach structurally aligns with recursion that these implementations do not result in networks which generalize as well as ours *out-of-distribution*.

6 Related Work

Motivated by patterns which are difficult to learn in deep neural networks, Joulin and Mikolov [12] developed stack-augmented recurrent networks. They use the stack to learn control over the memory of the network, enabling it to learn with infinite structured memory. They showed that these networks are able to learn some basic algorithms (such as binary addition) which require memorization. However, they did not study recursive algorithms or attempt to align the learned method with the algorithm structure. Cai et al. [13] incorporated recursion through the Neural Programmer-Interpreter (NPI) framework [14]. This was achieved by incorporating recursive elements into the NPI traces (somewhat similar to incorporating per-node hints in CLRS-30). They demonstrated strong generalization performance when learning sorting algorithms. In contrast to our approach, they do not explicitly introduce a call stack to learn the relevant state. Petersen et al. [15] proposed a method of relaxing conditions on control structures in algorithms such that they were smoothly differentiable. This approach allows neural networks to directly learn the relaxed algorithms. Similar to the method proposed by Cai et al. [13], this technique also cannot permit reasoning like recursive algorithms as there is no saving or restoring of state.

7 Conclusion & Future Work

To enable neural networks to inherit some of the generalization properties of recursive algorithms, we introduced a new framework for augmenting GNNs with a call stack. This framework permits GNNs to execute recursive algorithms. We also proposed changes for sampling intermediate algorithm trajectories and predicting outputs in the CLRS-30 algorithmic reasoning benchmark that improved structural alignment with recursion. With these improvements, our framework allowed a GNN with a call stack to significantly outperform previous work when generalizing out-of-distribution on DFS. Moreover, our stack-augmented graph neural network has the ability to perfectly recall state from history, avoiding the memory bottleneck of hidden states in recurrent networks.

One goal of future work would be to formalize the structural alignment of our modifications in CLRS-30 with our stack-augmented GNN architecture, similar to the alignment between DP and GNNs shown by Xu et al. [1]. This could pave the way for further insights about reasoning recursively with neural networks. We also note that our method relies on ground-truth hints to supervise stack usage. This is usually not available outside an algorithmic reasoning setting. Usage of the stack could instead be learned through reinforcement learning, supervising the stack with the policy loss. This would enable call stacks to be used with neural networks beyond only known recursive algorithms. Similar to the work by Li et al. [3], it could allow the network to discover new methods of using the stack which can outperform known solutions. For example, the policy network of a navigation agent could employ our architecture. This could enable the agent to better learn to map and navigate its environment as this task requires recursively planning paths.

Nevertheless, in its current state, our work is the first to demonstrate the use of a stack-augmented neural network in NAR. We have used this architecture to improve algorithmic alignment in recursive problems and enlarged the class of algorithms we can precisely reason about with GNNs. As a result, this work is a step towards transferable algorithmic knowledge [16] and generalist algorithmic learners [9]. We hope that our work sheds light on future directions in the journey towards learning to reason with neural networks.

Acknowledgements

References

[1] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. URL https://openreview.net/forum?id=rJxbJeHFPS. 1, 9

[2] Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7): 100273, July 2021. ISSN 2666-3899. doi: 10.1016/j.patter.2021.100273. URL https://www.sciencedirect.com/science/article/pii/S2666389921000994. 1

[3] Yujia Li, Felix Gimeno, Pushmeet Kohli, and Oriol Vinyals. Strong Generalization and Efficiency in Neural Programs, July 2020. URL http://arxiv.org/abs/2007.03629. arXiv:2007.03629 [cs, stat]. 1, 2, 9

[4] Danilo Numeroso, Davide Bacciu, and Petar Veličković. Dual algorithmic reasoning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=hhvkdRdWt1F. 1

[5] Petar Velickovic, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The CLRS algorithmic reasoning benchmark. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 22084–22102. PMLR, 2022. URL https://proceedings.mlr.press/v162/velickovic22a.html. 1, 2

[6] Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. Neural networks and the chomsky hierarchy. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=WbxHAzkeQcn. 2

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, Massachusetts, 3rd edition, 2009. ISBN 978-0-262-03384-8 978-0-262-53305-8. 2, 5, 9, 14

[8] Beatrice Bevilacqua, Kyriacos Nikiforou, Borja Ibarz, Ioana Bica, Michela Paganini, Charles Blundell, Jovana Mitrovic, and Petar Velickovic. Neural algorithmic reasoning with causal regularisation. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 2272–2288. PMLR, 2023. URL https://proceedings.mlr.press/v202/bevilacqua23a.html. 2

[9] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Joseph Dudzik, Matko Bosnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Velickovic. A generalist neural algorithmic learner. In Bastian Rieck and Razvan Pascanu, editors, *Learning on Graphs Conference, LoG 2022, 9-12 December 2022, Virtual Event*, volume 198 of *Proceedings of Machine Learning Research*, page 2. PMLR, 2022. URL https://proceedings.mlr.press/v198/ibarz22a.html. 7, 8, 9, 13

[10] Jan Koutník, Klaus Greff, Faustino J. Gomez, and Jürgen Schmidhuber. A clockwork RNN. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, volume 32 of *JMLR Workshop and Conference Proceedings*, pages 1863–1871. JMLR.org, 2014. URL http://proceedings.mlr.press/v32/koutnik14.html. 8

[11] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*, 1:1–150, 1991. 8

[12] Armand Joulin and Tomas Mikolov. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. 9

[13] Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=BkbY4psgg. 9

[14] Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.06279. 9

[15] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Learning with algorithmic supervision via continuous relaxations. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 16520–16531, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/89ae0fe22c47d374bc9350ef99e01685-Abstract.html. 9

[16] Louis-Pascal Xhonneux, Andreea-Ioana Deac, Petar Veličković, and Jian Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? In *Advances in Neural Information Processing Systems*, volume 34, pages 19500–19512. Curran Associates, Inc., 2021. 9

[17] Rick Durrett. *Random Graph Dynamics*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 2006. doi: 10.1017/CBO9780511546594. 12

[18] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017. 13

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980. 13

A   Altering The Dataset With Balanced Recursion Branches

The graph distribution in CLRS-30 for training and testing is generated from randomly sampling Erdős–Rényi (E-R) graphs with different edge connection probabilities. The expected distance between two nodes in these graphs is logarithmic in the number of nodes [17]. To also study balanced recursion branches, we modify the graph distribution by replacing 15% with randomly generated binary trees. These graphs have branching chains that are in expectation the same length, making them different to E-R graphs.
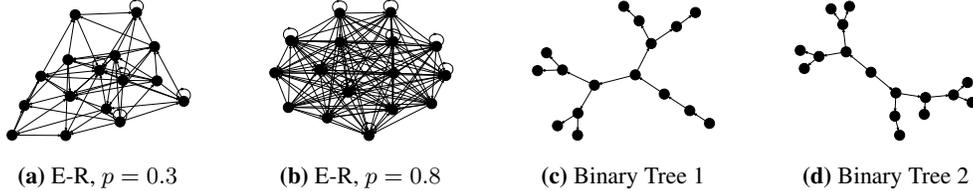

(a) E-R, $p = 0.3$ (b) E-R, $p = 0.8$ (c) Binary Tree 1 (d) Binary Tree 2

Figure 5: Example graphs from our dataset.

B   Pooling With Attention

In addition to the setup described in Section 3.2, we also evaluate a simple form of attention where we weight the embeddings produced per node based on a learned function $\phi_{\text{att}} : \mathbb{R}^{d_{\mathbf{h}}} \times \mathbb{R}^{d_{\text{graphfts}}} \to \mathbb{R}$ depending on both, the node embeddings as well as the encoded graph features.

$$\mathbf{z} := \bigoplus_{i \in V} \phi_{\text{att}}(\mathbf{p}_i^t, \mathbf{h}_g^t)\phi_{\text{value}}(\mathbf{p}_i^t) \tag{8}$$

The underlying idea here is that many algorithms would likely focus on the information of one or few nodes for each step embedding. Apart from that, note that an extensive variety of attention-based mechanisms would be possible. On the one hand, one could change ϕ_{att} from a simple MLP to a more complex function, e.g. by generating key and value vectors from the node embeddings and query vectors from the graph level features. On the other hand, one could change the representation of graph-level data from the graph features $\mathbf{h}_g^t$ to e.g. some pooled version of the node embeddings $\bigoplus_{i \in V} \mathbf{p}_i^t$ or a combination of these. However, thoroughly evaluating all of these techniques was not the main goal of this work.

In Experiment 9, we evaluate this attention-based pooling approach. It fails to generalize to larger graphs. Since we use sum pooling, this may be because the magnitude of the encoded graph features are out-of-distribution, making the generated weights perform worse. A mean-pooling approach could lead to better performance.

C   Implementation Details

Table 4 details the values of hyperparameters in our method. Note that complete implementation details are also provided in our repository. The codebase contains a README file with commands to reproduce all of our experiments.

D   Runtime And Memory Consumption

In Table 5, we show the runtime and memory consumption of our architecture for all of the ablations we ran in Table 3. These statistics reveal that using a stack does not increase memory consumption significantly. In fact, our modifications lead to significantly decreased memory consumption when compared to the baseline. This is likely a result of using smaller hints since we converted many node-level hints into graph-level ones. This is also the reason that output collection saves a significant chunk of memory since we do not need to predict the output hints for all nodes.

E   Depth-First Search Algorithm

Table 4: **Parameter values in our network configurations.**

Parameter	Value
GNN Architecture	MPNN [18]
Epochs	20000
$\bigoplus$	$\sum$
d_{h}	128
d_{stack}	64
ϕ_{value}	{layers: 2, hidden dim: 128}
ϕ_{att}	{layers: 2, hidden dim: 128}
Activation	ReLU
Optimizer	Adam [19]

Table 5: **Peak memory consumption (GB) and overall runtime (in minutes) for all training runs.**

		Graph-Level stack (Sec. 3.2)	Node-Wise stack (Sec. 3.1)	Hidden state $\mathbf{p}_i^{t-1}$ (Sec. 4.2)	Output collection (Sec. 4.3)	Teacher forcing (50%)	ϕ_{value} learned (Eq. 6)	Attention (App. B)	Peak GPU Memory	Total Runtime
Ibarz et al.	(1)	N/A	N/A	✓	✗	✓	N/A	N/A	22.61	$87.31^{\pm 0.12}$
	(2)	✓	✗	✗	✓	✓	✓	✗	7.14	$62.43^{\pm 0.10}$
	(3)	✗	✗	✗	✓	✓	N/A	N/A	6.87	$68.41^{\pm 0.09}$
	(4)	✓	✗	✓	✓	✓	✓	✗	7.56	$67.36^{\pm 0.59}$
	(5)	✗	✗	✓	✓	✓	N/A	N/A	8.82	$67.36^{\pm 0.59}$
	(6)	✓	✗	✗	✗	✓	✓	✗	13.33	$88.47^{\pm 0.10}$
Ours	(7)	✓	✗	✗	✓	✗	✓	✗	8.57	$71.32^{\pm 0.16}$
	(8)	✓	✗	✗	✓	✓	✗	✗	8.45	$67.82^{\pm 0.27}$
	(9)	✓	✗	✗	✓	✓	✓	✓	8.75	$69.68^{\pm 0.21}$
	(10)	✗	✓	✗	✓	✓	✓	N/A	7.14	$79.30^{\pm 0.75}$
	(11)	✗	✓	✓	✓	✓	✓	N/A	8.43	$80.93^{\pm 0.18}$

Algorithm 1: Depth-First Search. Reproduced from *Introduction to algorithms* [7, Chapter 22].

DFS(G)

1   **for** each vertex $u \in G.V$
2        $u.color = $ WHITE
3        $u.\pi = $ NIL
4    $time = 0$
5   **for** each vertex $u \in G.V$
6   **if** $u.color == $ WHITE
7   DFS-Visit(G, u)

DFS-Visit(G, u)

1    $time = time + 1$
2    $u.d = time$
3    $u.color = $ GRAY
4   **for** each $v \in G.Adj[u]$
5   **if** $v.color == $ WHITE
6            $v.\pi = u$
7   DFS-Visit(G, v)
8    $u.color = $ BLACK
9    $time = time + 1$
10   $u.f = time$

Listing 1: A high-level overview of our method as Python-like pseudocode.

```python
result_hint: str
current_node_hint: str
graph_hint_names: List[str]
node_hint_names: List[str]
V: List[Node]
E: List[Tuple[Node, Node]]

graph_hints = {n: None for n in graph_hint_names}
node_hints = {v: {n: None for n in node_hint_names} for v in V}
results = {v: None for v in V}
stack = []
for _ in range(GNN_iter):
    stack_top = stack[-1]
    h_g = sum(f_g[hint_name](hint_value)
              for hint_name, hint_value in graph_hints.items())
    inputs = {}
    for v in V:
        h_i = sum(f_n[hint_name](hint_value)
                  for hint_name, hint_value in node_hints[v].items())
        inputs[v] = concatenate(h_g, h_i, stack_top)

    p_i = psi(inputs)
    graph_hints = {k: g(sum(p_i[v] for v in V)) for k, g in g_g.items()}
    node_hints = {k: {v: g(p_i[v]) for v in  V} for k, g in g_n.items()}

    if argmax(graph_hints["stack_op"]) == 2: # push
        stack.push(sum(phi_value(p_i)))
    elif argmax(graph_hints["stack_op"]) == 0 and len(stack) > 1: # pop
        stack.pop()

    results[argmax(graph_hints[current_node_hint])] = graph_hints[result_hint]
```