How to transfer algorithmic reasoning knowledge to learn new algorithms?

Anonymous Author(s)
Affiliation
Address
email

Abstract

Learning to execute algorithms is a fundamental problem that has been widely studied. Prior work [1] has shown that to enable systematic generalisation on graph algorithms it is critical to have access to the intermediate steps of the program/algorithm. In many reasoning tasks, where algorithmic-style reasoning is important, we only have access to the input and output examples. Thus, inspired by the success of pre-training on similar tasks or data in Natural Language Processing (NLP) and Computer Vision, we set out to study how we can transfer algorithmic reasoning knowledge. Specifically, we investigate how we can use algorithms for which we have access to the execution trace to learn to solve similar tasks for which we do not. We investigate two major classes of graph algorithms, parallel algorithms such as breadth-first search and Bellman-Ford and sequential greedy algorithms such as Prim and Dijkstra. Due to the fundamental differences between algorithmic reasoning knowledge and feature extractors such as used in Computer Vision or NLP, we hypothesise that standard transfer techniques will not be sufficient to achieve systematic generalisation. To investigate this empirically we create a dataset including 9 algorithms and 3 different graph types. We validate this empirically and show how instead multi-task learning can be used to achieve the transfer of algorithmic reasoning knowledge.

1 Introduction

Transfer learning [2] has been responsible for significant successes in multiple areas of machine learning, including Natural Language Processing (NLP) [3, 4] and Computer Vision (CV) [5]. Pre-training and reusing the learned weights, by freezing them as feature extractors, or fine-tuning from them as an initialisation, are common approaches to transfer in these domains. This has enabled successful learning on problems where data is limited in some form.

Algorithmic reasoning on graphs [1, 6] is a fundamental problem that has been studied under the assumption that we have access to the execution traces of the algorithms we want to learn. In practice, such as many real world reasoning tasks, this may not be true. Due to the limited data, we look to transfer learning to enable us to solve algorithmic tasks without intermediate steps. Specifically, we investigate how to transfer knowledge between similar *graph algorithms* in the setting where we have access to the execution traces for some (e.g. PRIM [7, 8]), but not others (e.g. DIJKSTRA [9]). This has not been explored before, but is important as many reasoning related fields (e.g. question answering, link prediction in knowledge graphs) care deeply about systematic generalisation, while only having input-output pairs to learn from.

Veličković et al. [1] successfully trained graph neural networks (GNNs) [10, 11, 12, 13] to execute graph algorithms. Two key ingredients were access to the intermediate steps of the algorithm and algorithmic alignment [14]. Extending this work, we enable solving tasks without access to the

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

intermediate steps using other similar algorithms as an inductive bias. To do this we add several new algorithms. Further, since the algorithms studied in [1] were all expressible with only linear components in the GNN, we extend their architecture with a more expressive encoder to enable algorithmic alignment with more tasks.

One assumption we make is that the algorithms are similar (e.g. both PRIM and DIJKSTRA greedily select nodes from a priority queue). This shared algorithmic knowledge should serve as an inductive bias to enable a neural network to systematically generalise, even when learning only on input-output pairs. We hypothesise that due to differences between feature extraction and algorithmic reasoning, successful approaches to transfer as used in CV and NLP will provide only minimal improvements when transferring from one base algorithm to a target algorithm. While features across images may be similar, the features of algorithms differs (e.g. shortest distance in DIJKSTRA and lightest edge to the tree in PRIM), but are processed in a conceptually similar manner. Our intuition is that this conceptual relationship may not yield weights that are near each other in the weight space, thus making transfer difficult. We instead propose to use the base algorithms as inductive biases by training them with the intermediate steps along side training the target algorithm, for which we do not provide intermediate supervision. Our second hypothesis is that this will help systematic generalisation. We validate both hypotheses—transfer does not help, while multi-task learning helps—empirically.

The contributions of this paper are:

1. presenting a new benchmark for transferring algorithmic reasoning knowledge on graphs;
2. sampling the best trajectory to stabilise training when no execution traces are available;
3. show that standard transfer techniques fail to significantly improve systematic generalisation;
4. demonstrate how systematic generalisation can instead be improved on algorithmic tasks with multi-task learning.

2 Related Work

Neural Execution: Many papers have studied how to execute algorithms before [15, 16, 17, 18, 19]. With the rise of graph representation learning [20, 21, 22] and graph neural networks [10, 11, 12, 13], recent work has looked at executing graph algorithms [1, 6]. This line of work assumes access to the intermediate steps of the algorithm. Further, people have looked at teaching transformers to learn explicitly denoted subroutines and then learn how to combine them [23]. The key difference between our work and prior work is that we look at the setting of learning graph algorithms where we do not have access to the execution trace and we do not denote subroutines explicitly. Instead we propose to use similar algorithms as an inductive bias and look at how to enable this transfer of algorithmic knowledge.

Transfer learning: Transfer learning was originally proposed in the 1970s [24, 25]. Today, it is a common tool in NLP and Computer Vision to use pre-trained models and fine-tune them on the target task [2, 3, 5]. This is helpful when the target task is similar and the available data is limited in some way. Our work differs in two ways; firstly, we focus on algorithmic reasoning knowledge, which we hypothesis will require different techniques than reusing weights. Secondly, the data on the target task is partially missing specifically the execution trace of the algorithm is missing.

3 Background

This section gives background on the algorithms and graphs used for the experiments. To be precise we consider a graph to be a tuple $G = (V, E)$ with a set of vertices V and a set of edges $E \subseteq V \times V$[1].

This section explains the algorithms studied in this paper in more detail. After that, we give a brief introduction into graph neural networks, the standard architecture paradigm for graph inputs.

3.1 Algorithms

We broadly study two classes of algorithmic reasoning: parallel and sequential reasoning. Specifically, the parallel algorithms (shown in Algorithm 1) studied in this paper exchange messages with their

[1] Throughout the paper we use graphs that are simple with weights.

neighbours until an equilibrium is reached. Sequential algorithms, presented in Algorithm 2, greedily remove elements from a priority queue and updating the neighbouring nodes' keys.

Algorithm 1 Parallel

Input: graph G, weights w, source index i
initialise_nodes(G.vertices, i)
repeat
  for $(u, v) \in G$.edges() **do**
    relax_edge(u, v, w)
  end for
until none of the nodes change

Algorithm 2 Sequential

Input: graph G, edge weights w, source node index i
initialise_nodes(G.vertices, i)
$Q \leftarrow$ PriorityQueue(G.vertices)
repeat
  $u \leftarrow Q$.pop_min()
  for $v \in G$.neighbours(u) **do**
    relax_edge(u, v, w)
  end for
until Q is empty

Different algorithms will implement their own `relax_edge` and `initialise_nodes` functions, however, the overall framework stays the same and can be learned by our GNN architecture (§ 4.1). For the sequential algorithms each node has a state feature indicating whether it has been removed from the priority queue, a key feature for the priority queue, and a pointer to the predecessor node.

We study the following algorithms:

PARALLEL

1. BREADTH-FIRST SEARCH (BFS)
2. BELLMAN-FORD
3. WIDEST PATH (PARALLEL)
4. MOST RELIABLE-PATH (PARALLEL)

SEQUENTIAL

1. PRIMS
2. DIJKSTRA
3. DEPTH-FIRST SEARCH (DFS)
4. WIDEST PATH (PARALLEL)
5. MOST RELIABLE-PATH (PARALLEL)

We give the pseudo-code for all algorithms in the supplementary. Note that the `relax_edge` function is increasingly difficult to learn for the neural network as you go down the list: PRIMS uses the edge weight as a key, thus the `relax_edge` function is the identity, for DIJKSTRA and DEPTH-FIRST SEARCH addition is necessary to compute the edge update, for WIDEST-PATH we need to take a maximum, which with a ReLU activation can still be done exactly, and for MOST RELIABLE-PATH the neural network needs to learn to approximate multiplication.[2]

3.2 Graph Neural Networks

A general framework to describe several graph neural network architecture is the message passing framework introduced by Gilmer et al. [12]. Such graph neural networks (GNNs) consist of three parts: a message function M, an update function U, and an aggregator $\bigoplus$. M and U are arbitrary neural networks. The high-level idea is

1. each node computes a message for its neighbours using M;
2. then each node aggregates the received messages using $\bigoplus$;
3. finally, each node updates its embedding using U.

The message function in this paper takes as input the sending and receiving nodes' representation as well as an encoding of the edge feature along which the message is sent. The aggregator chosen is the $\max$ aggregator applied element-wise as it was determined to work best by Veličković et al. [1] and provides algorithmic alignment [14] between the architecture and the tasks. The update function takes as input the node's previous embedding and aggregated messages.

[2] We note that the tasks are all linear time in the size of the graph. More challenging problems such as NP-hard problems are primarily more difficult because of their increased run-time and hence longer sequences allowing for more accumulation of errors. As long as we have algorithmic alignment [14] between the architecture and the algorithm in question there is no additional challenge to NP-hard problems except the length of their sequence and hence more opportunities to introduce errors and propagate them.

3.3 Problem definition

We study learning graph algorithms that take in a graph G, a weight function $w : E \to \mathbb{R}$ that assigns a weight to each edge in the graph, and node features $X : V \to \mathbb{R}^k$. Algorithms compute an output $Y : V \to F^k$ and a predecessor $P : V \to V$ at each time step. This encompasses a large class of graph algorithms solving tasks such as reachability or shortest-path.

For instance, for a sequential algorithm the ith node's features $X_t[i]$ would be the `key` value and the `state` variable. Given X_t, the graph G, and the edge weights, the graph algorithm at each iteration returns the node features Y_t and the predecessor for each P_t (`pred` variable in pseudocode § 3.1). Y_T and P_T at the final step are considered the output of the algorithm. The intermediate steps refers to $X_t, Y_t, P_t \; \forall t \in \{1, \ldots, T-1\}$ after each iteration.

4 Methods

In this section, we first briefly present two architectures we will use for training and then discuss how existing algorithmic knowledge can be used to solve tasks when no execution trace is given.

4.1 Architecture

As our starting point we choose the encoder-processor-decoder architecture proposed in Veličković et al. [1]. The high-level idea is that the network encodes the current node state into a hidden embedding, on which a graph neural network unit, called the processor, is applied using the graph structure. The results are then decoded by the decoder, which does the prediction of the node features at the next time step (§ 3.1). The point of this architecture is to allow several algorithms to use the same processor architecture. A more detailed summary of the architecture is given below:

4.1.1 NeuralExecutor

The NeuralExecutor (NE) [1] uses an encoder-processor-decoder architecture. Let $X \in \mathbb{R}^{n \times k}$ be node states, where n, k are the number of nodes and features, respectively. Each edge (u, v) has a weight $w(u, v) \in \mathbb{R}$. The architecture keeps a hidden state for each node $H \in \mathbb{R}^{n \times l}$ with l features, which is initialised to all zeros. The encoder $\mathcal{E}$ consists of a linear layer and computes a hidden embedding $\mathcal{E}(X_i, H_i) = Z_i$, where i indicates the ith step in the computation. The processor $\mathcal{P}$ is message passing neural network (MPNN) with a $\max$ aggregator with linear message and update functions. The processor computes the new hidden state for each node $\mathcal{P}(H_i, A, w) = H_{i+1}$. Then, we have the decoder $\mathcal{D}(Z_i, H_{i+1}) = Y_{i+1}$ and predecessor predictor $\mathcal{S}(Z_i, H_{i+1}) = S_i$. Finally, a termination network $\sigma(\mathcal{T}(H_{i+1}))$ decides whether we should terminate or not.

We make minor changes to allow for better algorithmic alignment: we remove the `ReLU` activations from the processor and replace the termination layer with a processor and linear module[3].

4.1.2 NeuralExecutor++

Veličković et al. [1] showed positive transfer when learning algorithms in a multi-task setup with intermediate steps. They did so by concatenating together the node features and encoding them together into the same h embedding (see Fig. 1). This has the advantage of giving strong guidance to the secondary algorithm learned in this multi-task set-up. However, ideally we are able to only use the base algorithm during training without having to use it at inference time introducing another failure mode. Thus, we propose to change the architecture as follows:

Since the node encoder is unable to learn the individual subroutines operating on edges, we replace it with a latent encoder for each task that operates on edges and for an edge (i, j) takes in $[h^{(i)}, x^{(i)}, e^{(ij)}, h^{(j)}, x^{(j)}]$ (Fig. 1). The important difference is that each has task has its own encoder rather than all tasks sharing one encoder. The goal is to force the model to learn the shared subroutines in the processor only and the specialised subroutines in the latent encoder only (see Fig. 1).

[3] We remove the `ReLU` because it limits the ability of the max aggregator to minimise values by using negative inputs. The termination condition depends on whether the remaining nodes are still reachable hence an MPNN is a more appropriate than a linear layer.

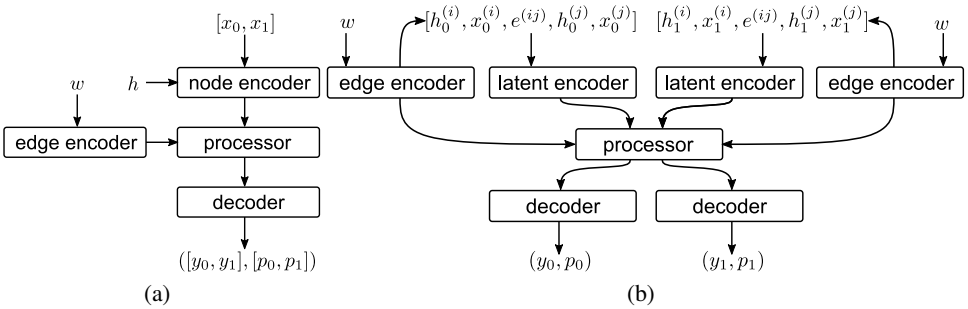

Figure 1: x_0, x_1 are the node states of two algorithms respectively. w are the edge weights. p_0, p_1 are the predecessor predictions for the two algorithms respectively and y_0, y_1 are the next node state predictions. h_is are the previous hidden state kept by the network and $e^{(ij)}$ is the computed edge weight embedding. Superscript indices indicate a particular node. (a) Shows the original Neural Executor architecture when doing multi-task learning. (b) Shows the more expressive Neural Executor++ when doing multi-task learning. The key difference is that (b) forces a common way to operate on a hidden embedding space, while (a) focuses on achieving both at the same time.

Further, we change the latent encoder to consist of a linear and non-linear encoder in parallel that are added together. This should allow for algorithmic alignment with a much larger array of tasks (specifically WIDEST PATH), but comes at the cost that overfitting is more likely, which may hurt generalisation to larger graphs. This last problem has been avoided in prior work [1, 14] by having the neural network only learn linear components, which will not be able to overfit easily.

4.2 Training and Loss functions

Sequential algorithms (Seq): We used `softmax` for the prediction of the next node to be removed from the queue and `softmax` for the predecessor prediction, where we masked out all nodes except the neighbours and the node itself. We used a smooth l_1 loss with $\beta = 0.001$ for the prediction of the key of the selected node. Finally, we used `binary cross entropy` for termination prediction. We only update the node state of the chosen node, this helps with drift of the node state at test time.

Parallel algorithms (Par): We used a smooth l_1 loss with $\beta = 0.001$ for the prediction of the key except for BFS where we used binary cross-entropy as the node state is either 0 or 1. In this setting, during teacher-forcing, we masked out nodes that are unreachable at a given time step.

Teacher forcing (TF): We train the network to predict the next step given the ground truth inputs. At test time the networks prediction are used instead of the ground truth.

No algorithm (NA): We train the network using only the loss on the final outputs, this means we change the `softmax` for predicting the next node to a `binary cross entropy` for sequential algorithms. The next inputs are those predicted by the network using `gumbel softmax` to predict the next node at each stage. The number of steps are given to the network in this scenario. Inspired by [26], we sample 10 trajectories using the best one for back-propagation, when using a `gumbel softmax`. The idea is that only the best loss is of interest at evaluation time and not the average loss. WE show in the supplementary that this helps stabilise and improve training compared to taking the mean of the trajectories.

4.3 Standard transfer methods

We try three main approaches:

Freeze weights: This is a standard transfer technique, where we use the learned weights of the processor unit on a base algorithm B for learning a target algorithm T. In this set-up we freeze the weights of the processor and only the encoder-decoder part of the architecture can be learned. We assume that the algorithm B was learned with teacher forcing, while T is learned with no-algorithm. We note that encoder-decoder weights cannot be reused because in general the input/output dimensions may differ from algorithm to algorithm.

Fine-tune weights: This is the same as the previous technique except that we do not freeze the processor weights and instead let them be changed by the gradient descent algorithm.

2-Processors: Again we assume we have access to the processor weights learned on a base algorithm B. This setting uses two processors in parallel whose outputs we sum together. One of the processor uses the learned processor weights and is frozen, while the other processor is free to learn the necessary changes and is randomly initialised. We suspect this will be the best transfer method as it does not forget the information given by the domain algorithm, but retains the flexibility to adapt the processor.

4.4 Transfer via multi-task learning

A stronger inductive bias is to train the base algorithm B, for which we have access to intermediate steps, together with the target algorithm T, for which we do not. This multi-task set-up highlights the difference between the original Neural Executor architecture proposed in Veličković et al. [1] and Neural Executor++ (see Fig. 1). The former simply concatenates the node features of B and T and forces them to share the hidden embedding. The latter allows different encoder-decoders and only shares the weights of the processor, i.e. each algorithm has its own hidden embedding. This second approach encourages the network to execute the shared subroutine in the processor and the individual subroutines in the encoder-decoder. The overall idea is that the base algorithm B serves as an inductive bias as to how to structure the latent space and teach the processor how to evolve T.

4.5 Graphs

We study three kinds of graphs:

$$\text{i) Erdos-Renyi } p = \min\left(\frac{\log_2 |V|}{|V|}, 0.5\right), \text{ ii) Barabasi-Albert, iii) 2d-grid graphs.}$$

Erdos-Renyi graphs are random graphs where each possible edge has probability p of being added to the graph. Barabasi-Albert graphs are power-law graphs with a few highly connected nodes and many dangling nodes. 2d-grid graphs are very regular graphs in an arbitrary 2d-grid shape. We chose these 3 classes of graphs because they represent some of the major possible differences between graphs. 2d-grid graphs are very regular and Veličković et al. [1] notes that very regular graphs tend to transfer poorly from or to random graphs. Erdos-Renyi graphs tend to be sparse, but highly likely to be connected. Barabasi-Albert graphs tend to be quite dense graphs with shorter average path-length than random graphs. As such these graph classes differ significantly from each other.

For all graphs, we generate edge weights that are uniformly between $[0.2, 1.0]$, this range prevents key values such as shortest path from becoming too extreme[4].

5 Experiments

For all experiments we use 5,000 graphs of each type (Erdos-Renyi (ER), Barabasi-Albert (BA), 2d-Grids (2d-G)) with 20 nodes each. We train using ADAM [27] with a learning rate of 0.0005, a batch size of 64, and use early stopping with a patience of 10 to prevent overfitting. We test on graphs size 20, 50, and 100 nodes. The hidden embedding size is set to 32 except for NE++ for multi-task experiments, where it is 16 to account for the additional expressivity of having several encoders. Each experiment was executed on a V100 GPU in less than 5 hours for the longest experiment.

We measure the average performance over all 3 graph types at evaluation separately and present the average with standard deviation in the main paper. Large standard deviation may arise due to the extreme difference between random graphs of type ER or BA versus 2d-G graphs.

5.1 Metrics

Sequential algorithms: *Predecessor (Pred.)* accuracy is the most important measure as to whether a task has been successfully completed as it gives us the path predicted by the network. *Next node (Next)* accuracy measures whether the next node is the correct one to pick. *Key* accuracy measures whether

[4] All code to generate data and train models will be released upon acceptance with an MIT license.

Table 1: Teacher forcing (seq.).

Model	#Nodes	DIJKSTRA Next node	DIJKSTRA Key	DIJKSTRA Predecessor	MOST RELIABLE Next node	MOST RELIABLE Key	MOST RELIABLE Predecessor
NE	20	0.982 ± 0.004	0.0367 ± 0.01	0.995 ± 0.002	0.762 ± 0.05	0.0358 ± 0.02	0.947 ± 0.01
	50	0.911 ± 0.009	0.569 ± 0.7	0.98 ± 0.02	0.443 ± 0.08	0.0697 ± 0.05	0.901 ± 0.01
	100	0.659 ± 0.02	4.79 ± 6	0.936 ± 0.04	0.237 ± 0.07	0.0924 ± 0.06	0.833 ± 0.007
NE ++	20	0.992 ± 0.003	0.0108 ± 0.004	0.997 ± 0.0008	0.826 ± 0.07	0.0264 ± 0.02	0.953 ± 0.03
	50	0.65 ± 0.03	445 ± 600	0.789 ± 0.02	0.508 ± 0.2	0.0676 ± 0.05	0.888 ± 0.05
	100	0.271 ± 0.01	$1.62e10 \pm 2e10$	0.478 ± 0.07	0.301 ± 0.2	0.0906 ± 0.06	0.829 ± 0.06

Table 2: No algorithm (seq.).

Model	#Nodes	DIJKSTRA Key	DIJKSTRA Predecessor	MOST RELIABLE Key	MOST RELIABLE Predecessor
NE	20	$0.000104 \pm 7e\text{-}05$	0.977 ± 0.01	0.00829 ± 0.001	0.813 ± 0.06
	50	$9.71e5 \pm 1e06$	0.879 ± 0.1	20 ± 7	0.103 ± 0.08
	100	$3.34e16 \pm 5e16$	0.806 ± 0.2	$1.32e6 \pm 4e5$	0.232 ± 0.05
NE ++	20	$6.4e\text{-}06 \pm 3e\text{-}06$	0.995 ± 0.003	$0.000465 \pm 6e\text{-}05$	0.874 ± 0.06
	50	1.81 ± 2	0.851 ± 0.06	125 ± 70	0.434 ± 0.06
	100	$8.54e3 \pm 1e4$	0.612 ± 0.05	$2.8e13 \pm 4e13$	0.240 ± 0.05

the key of the picked node is correct, measured in mean squared error. *Next node* are indicative of whether the correct algorithm is being executed, while *Pred.* and *Key* primarily serves to indicate the correctness of the solutions found.

Parallel algorithms: *Key* accuracy measures the node features mean squared error for all algorithms except BFS, where it is measured in accuracy as the node feature is a binary choice between 0 and 1. *Predecessor (Pred.)* accuracy measures the accuracy of predicting the predecessor node.

6 Results and Discussion

For the sequential algorithms we study transfer from PRIM to DIJKSTRA and from WIDEST PATH to MOST RELIABLE PATH, for parallel Algorithms we study transfer from BFS to BELLMAN-FORD and from WIDEST PATH to MOST RELIABLE PATH.

6.1 Sequential

The first experiments establish baselines in terms of achievable performance given the intermediate steps and trained with teacher-forcing (Tab. 1). We run each algorithm separately. Next we establish the performance in the no-algorithm setting (Tab. 2), i.e. what is achievable without intermediate supervision.

Expressivity can harm systematic generalisation: Firstly, we note that the additional expressivity of the NE++ (§ 4.1.2) seems to hurt systematic generalisation even with the large amount of data available (Tab. 1) as we can see on DIJKSTRA. On MOST RELIABLE both do equally well on *Key* and *Pred.*, but looking at *Next node* we can see that NE++ does better in simulating the algorithm. MOST RELIABLE is a non-linear task so a non-linear encoder is expected to help. We also note that as the graphs grow in size, the number of reachable nodes in the priority queue increases, making it more likely we pick the wrong node without affecting the correctness of prediction.

Secondly, we note that in the NA setting (Tab. 2) the NE is able to solve the *Pred.* prediction quite well up to 100 nodes, but clearly found an alternative way of reasoning as the key prediction is hugely wrong for larger graphs. Note that the largest shortest distances will be found in 2-grid graphs, where it will be upper bounded by 51. Also for MOST RELIABLE the performance on *Pred.* drop at 50 nodes is significantly more severe with less representation power.

Transfer yields little improvement: The two key experiments are the transfer setting (§ 4.3) and the multi-task setting (§ 4.4). We hypothesised that the standard transfer experiments (fine-tune and freeze) would not help systematic generalisation. None of the transfer methods (Tab. 3) help generalise either task significantly. In fact, they harm systematic generalisation in terms of *Pred.* prediction in all cases. The only benefit that can be observed is better generalisation on *Key* accuracy indicating that the network outputs are less extreme. The best transfer method is 2-Processor as we hypothesised in § 4.3, which improves *Key* prediction at the cost of harming *Pred.* accuracy.

Table 3: Transfer to no algorithm (seq.). Pre-trained on PRIM and WIDEST, respectively.

Model	#Nodes	DIJKSTRA		MOST RELIABLE	
		Key	Predecessor	Key	Predecessor
NE Freeze	20	0.014 ± 0.005	0.919 ± 0.03	0.0235 ± 0.003	0.752 ± 0.06
	50	122 ± 200	0.403 ± 0.09	$3.91e5 \pm 3e4$	0.136 ± 0.009
	100	$3.18e6 \pm 4e6$	0.393 ± 0.1	$5.06e15 \pm 7e14$	0.239 ± 0.04
NE Fine-tune	20	0.0021 ± 0.0009	0.95 ± 0.02	0.036 ± 0.005	0.773 ± 0.07
	50	$1.11e3 \pm 1e3$	0.759 ± 0.09	$2e3 \pm 200$	0.364 ± 0.1
	100	$5.93e7 \pm 8e7$	0.612 ± 0.1	$1.06e9 \pm 1e8$	0.291 ± 0.02
NE 2-Processor	20	0.00136 ± 0.0005	0.94 ± 0.03	0.0163 ± 0.003	0.769 ± 0.08
	50	14.2 ± 20	0.838 ± 0.06	205 ± 30	0.251 ± 0.04
	100	$1.04e4 \pm 1e4$	0.695 ± 0.07	$1.22e10 \pm 4e9$	0.185 ± 0.03
NE++ Freeze	20	0.00136 ± 0.001	0.937 ± 0.02	0.00687 ± 0.0008	0.801 ± 0.09
	50	42.6 ± 50	0.159 ± 0.04	465 ± 100	0.42 ± 0.1
	100	262 ± 400	0.105 ± 0.07	$8.06e7 \pm 3e7$	0.328 ± 0.08
NE++ Fine-tune	20	0.000414 ± 0.0003	0.966 ± 0.02	0.00669 ± 0.002	0.809 ± 0.07
	50	13.5 ± 20	0.0377 ± 0.03	$1.79e5 \pm 2e5$	0.243 ± 0.05
	100	$2.51e4 \pm 4e4$	0.0379 ± 0.04	$8.17e14 \pm 5e14$	0.226 ± 0.05
NE++ 2-Processor	20	0.00443 ± 0.002	0.94 ± 0.04	0.0022 ± 0.0003	0.846 ± 0.05
	50	16.6 ± 20	0.644 ± 0.1	306 ± 70	0.356 ± 0.03
	100	$4.66e3 \pm 6e3$	0.221 ± 0.02	$3.58e6 \pm 9e5$	0.209 ± 0.02

Table 4: Multi-task (seq.): Using PRIM and WIDEST as inductive bias, respectively.

Model	#Nodes	DIJKSTRA		MOST RELIABLE	
		Key	Predecessor	Key	Predecessor
NE	20	0.00362 ± 0.0005	0.958 ± 0.01	0.207 ± 0.03	0.548 ± 0.09
	50	11.5 ± 2	0.866 ± 0.1	1.85 ± 0.5	0.499 ± 0.06
	100	126 ± 30	0.697 ± 0.3	6.47 ± 3	0.403 ± 0.01
NE ++	20	$0.000178 \pm 8e\text{-}05$	0.981 ± 0.009	0.00279 ± 0.0004	0.834 ± 0.07
	50	0.413 ± 0.4	0.839 ± 0.1	0.199 ± 0.3	0.815 ± 0.009
	100	2.91 ± 3	0.718 ± 0.2	0.843 ± 1	0.733 ± 0.1

Multi-task helps systematic generalisation: In the multi-task set-up (Tab. 4), several things occur: the *Key* prediction generalises even better and is predicting in a reasonable range given the longest shortest path in graphs of size 100. Further, NE in this setting has similar *Pred.* accuracy on DIJKSTRA compared to NA, NE++ benefitted from the inductive bias in terms of its *Pred.* accuracy on graphs of size 100 for DIJKSTRA. The results on MOST RELIABLE are significantly improved and NE++ achieves good levels of systematic generalisation in solving the task. NE interestingly worsens in its performance on 20 nodes, but maintains a stable *Pred.* accuracy on larger graphs. Demonstrating that the inductive bias from WIDEST prevents overfitting in distribution and improves the performance on larger graphs. Overall, the results validate our initial hypothesis that multi-task learning is the correct approach to transfer knowledge.

Trying to extract shared subroutines does not help transfer: Finally, we study to what extent the models are able to separate the common shared subroutines and the subroutines individual to each algorithm by training multiple algorithms with TF in a multi-task set-up together (Tab. 5). If the processor successfully captures only the shared subroutines, then we might expect the transfer results to be improve. We can see in Tab. 5 that multi-task pre-training does not significantly improve results and that multi-task learning with the target algorithm is still the best approach. However, one alternative explanation is that given a good processor, the encoder struggles to learn the expected encoding by the processor and thus performs poorly.

6.2 Parallel

Parallel algorithms are significantly easier than sequential ones due to their much shorter length and the lack of a central data-structure that needs to be learned to execute. This can be observed in the much higher performance in the NA setting (Tab. 6). Interestingly, it seems that in this setting expressivity was helpful for systematic generalisation, even in the BELLMAN-FORD setting.

Transfer harms performance: In Tab. 6 we show only the best transfer result, but as we can see this actually harms *Pred.* accuracy for NE++ for both algorithms, while producing roughly the same result for NE. In both cases, the results suggest that random initialisations are better than transfer ones. We think this may because the shared algorithmic knowledge of parallel algorithms is already inherently captured by GNNs as they apply message functions in parallel to each edge, which then only need to learn the `relax_edge` function. Pre-training on several algorithms did not help (Tab. 5).

Table 5: 2-Proc. transfer pre-trained on PRIM, DIJKSTRA, & DFS for sequential and BFS & BELLMAN-FORD for parallel.

Model	#Nodes	MOST RELIABLE (SEQ)		MOST RELIABLE (PAR)	
		Key	Predecessor	Key	Predecessor
NE ++ 2-Proc.	20	0.00237 ± 0.0003	0.837 ± 0.06	0.0408 ± 0.009	0.773 ± 0.07
	50	62.9 ± 20	0.394 ± 0.05	0.161 ± 0.1	0.637 ± 0.04
	100	$1.76e6 \pm 4e5$	0.242 ± 0.07	2.68 ± 4	0.552 ± 0.08

Table 6: No algorithm (par.): For transfer we report the results of the best method (§ 4.3). Pre-trained on BFS and WIDEST respectively.

Model	#Nodes	BELLMAN-FORD		MOST RELIABLE PATH	
		Key	Predecessor	Key	Predecessor
NE (NA)	20	0.0182 ± 0.02	0.943 ± 0.02	0.018 ± 0.002	0.774 ± 0.06
	50	59 ± 80	0.836 ± 0.1	0.147 ± 0.2	0.673 ± 0.02
	100	$1.98e6 \pm 3e6$	0.739 ± 0.2	10.4 ± 10	0.565 ± 0.05
NE++ (NA)	20	0.00253 ± 0.002	0.972 ± 0.01	0.00957 ± 0.004	0.855 ± 0.06
	50	0.226 ± 0.3	0.943 ± 0.01	0.0367 ± 0.04	0.829 ± 0.03
	100	196 ± 300	0.905 ± 0.04	120 ± 200	0.783 ± 0.02
NE (Transfer Fine-tune)	20	0.0386 ± 0.02	0.928 ± 0.03	0.0221 ± 0.006	0.763 ± 0.06
	50	25 ± 40	0.838 ± 0.05	0.332 ± 0.4	0.669 ± 0.02
	100	$1.72e5 \pm 2e5$	0.758 ± 0.1	230 ± 300	0.598 ± 0.03
NE++ (Transfer 2-Proc.)	20	0.0223 ± 0.02	0.938 ± 0.03	0.0131 ± 0.003	0.804 ± 0.07
	50	0.666 ± 0.7	0.895 ± 0.005	3.04 ± 4	0.687 ± 0.05
	100	10.8 ± 10	0.832 ± 0.05	579 ± 800	0.589 ± 0.1
NE (Multi-task)	20	0.0154 ± 0.02	0.966 ± 0.01	0.173 ± 0.1	0.654 ± 0.02
	50	6.22 ± 9	0.949 ± 0.004	0.407 ± 0.4	0.638 ± 0.03
	100	$1.53e3 \pm 1e3$	0.904 ± 0.02	0.615 ± 0.6	0.624 ± 0.05
NE++ (Multi-task)	20	0.00353 ± 0.004	0.977 ± 0.01	0.00672 ± 0.0005	0.847 ± 0.06
	50	0.0141 ± 0.02	0.970 ± 0.006	0.00805 ± 0.002	0.818 ± 0.01
	100	8.84 ± 10	0.870 ± 0.1	0.00971 ± 0.002	0.788 ± 0.02

Multi-task only helps Key accuracy: Similarly to sequential reasoning multi-task vastly outperforms transfer techniques and significantly improves *Key* prediction compared to NA, while keeping *Pred.* similar. Contrary to sequential reasoning the *Pred.* prediction is comparable between NA and multi-task. We think this is due to the shorter execution length providing less of an inductive bias for the target algorithm and the strong inductive bias of GNN architectures towards parallel algorithms. However, the access to more stable gradients due to the multi-task learning approach seems to help learning to some extent due to the improved *Key* predictions. Further, we observe that when the model has less capacity (NE on MOST RELIABLE PATH) multi-task is still able to improve systematic generalisation on *Pred.* at the cost of slightly worse in-distribution (20 nodes) performance.

7 Conclusion

We set out to investigate how systematic generalisation could be improved on algorithmic tasks when the intermediate steps of the algorithm are not available. Inspired by the success of transfer learning in domains such as CV and NLP, we investigated it's applicability to learning graph algorithms in this setting. We showed that standard transfer learning is inadequate to leverage algorithmic knowledge learned from intermediate steps to new algorithmic tasks. Further, we showed how multi-task learning can enable the successful transfer of inductive biases learned from other algorithms when intermediate steps are available, significantly improving systematic generalisation. The results are especially strong in the more difficult sequential reasoning domain. Moreover, we conclude that expressivity can hurt systematic generalisation if the task is too easy and intermediate supervision is available. This should be taken into account when choosing the model. These disadvantages disappeared when trying to learn algorithmic reasoning without intermediate steps in our multi-task set-up, in this setting NE++ always outperforms the simpler architecture. Both architectures can achieve systematic generalisation. *Limitations* of our work are that the results are specific to algorithms on static graphs. Furthermore, as the number of execution steps increases faster than linear in the number of nodes results are likely to worsen significantly.

This paper's contributions are fundamental in nature and thus, the societal impact of this paper is low and there are no associated ethical risks. Any benefits or risks stem from further advances in reasoning systems that may be in some form be based on this work.

References

[1] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

[2] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 201–208. JMLR Workshop and Conference Proceedings, 2010.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[4] TB Brown, B Mann, N Ryder, M Subbiah, J Kaplan, P Dhariwal, A Neelakantan, P Shyam, G Sastry, A Askell, et al. Language models are few-shot learners. arxiv 2020. *arXiv preprint arXiv:2005.14165*, 4, 2020.

[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale, 2020.

[6] Petar Veličković, Lars Buesing, Matthew C Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

[7] Vojtěch Jarník. On a certain problem of minimization. *Práce moravskè přírodovědecké společnosti 6, fasc. 4*, pages 57–63, 1930. URL `http://hdl.handle.net/10338.dmlcz/500726`.

[8] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, pages 1389–1401, 1957. URL `https://archive.org/details/bstj36-6-1389`.

[9] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959. URL `https://doi.org/10.1007/BF01386390`.

[10] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[11] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[12] 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] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[14] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about?, 2020.

[15] Wojciech Zaremba and Ilya Sutskever. Learning to execute, 2015.

[16] Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms, 2016.

[17] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines, 2016.

[18] Scott Reed and Nando de Freitas. Neural programmer-interpreters, 2016.

[19] Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018. URL `https://proceedings.neurips.cc/paper/2018/file/e2eabaf96372e20a9e3d4b5f83723a61-Paper.pdf`.

[20] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, Jul 2017. ISSN 1558-0792. doi: 10.1109/msp.2017.2693418. URL `http://dx.doi.org/10.1109/MSP.2017.2693418`.

[21] William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications, 2018.

[22] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[23] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines, 2020.

[24] S Bozinovski and A Fulgosi. The influence of pattern similarity and transfer learning upon training of a base perceptron b2. In *Proceedings of Symposium Informatica*, pages 3–121, 1976.

[25] S Bozinovski. Teaching space: A representation concept for adaptive pattern classification. Technical report, COINS Technical Report, University of Massachusetts at Amherst, 1981.

[26] Brenden K Petersen, Mikel Landajuela Larma, Terrell N. Mundhenk, Claudio Prata Santiago, Soo Kyung Kim, and Joanne Taery Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=m5Qsh0kBQG`.

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

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes]
- Did you include the license to the code and datasets? [No]
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See § 7
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See § 7
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] See § 5 for instructions. The code is included in supplementary.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 5.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] We report standard deviation across graphtypes.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See § 5
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes] This work wrote the code to generate a new benchmark (§ 3,4.5)
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] The data consists of graphs and algorithms, no personal information was used.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] The data consists of graphs and algorithms, no personal information was used.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]