Sequential Reptile for Multitask Graph Execution: Group 14

Yihang Guo
yihangguo@g.ucla.edu

Alexander Taylor
ataylor2@cs.ucla.edu

Ted Zadouri
tedzadouri@g.ucla.edu

Jingdong Gao
mxuan@g.ucla.edu

Armaan Abraham
armaanabraham@g.ucla.edu

Abstract

Graph Neural Networks (GNNs) have been shown to be powerful tools for representing the structure of the execution of classic computer science algorithms, such as Breadth First Search and Bellman-Ford. However, prior approaches have not attempted to instill any form of order into the training schedule of their processing network. In this work, we make use of a scheduling approach that has been shown to yield improved results in a multi-task setting by aligning dissimilar task gradients. We demonstrate that a novel approach SEGA: Sequential neural Execution of Graph Algorithms that incorporates a scheduling approach in the training process in addition to the modulating the availability of training samples substantively improves results over previous works. The code is available at: https://github.com/mxuan0/MultitaskGraphExecution:

1 Introduction of the background

Deep learning models have become embedded deeper and deeper into modern society, and have repeatedly been shown to achieve high-performance on many tasks and even exceed human expert performance with increasing frequency[1, 19] across multiple domains. However, despite consistently achieving high performances across many domains [11], deep learning models offer few guarantees in terms of generalization. Generalizing to unseen data distributions has remained a major challenge in machine learning and has been addressed by works in the domains of transfer, meta, and out-of-distribution learning. In a similar vein, recent works in the area of neural algorithmic reasoning [14, 15, 17] have attempted to instill guarantees of generalization into deep learning frameworks by using classic computer science algorithms, such as the Bellman-Ford shortest path algorithm (BF) and Breadth-First Search (BFS), which have been proven to have guaranteed time complexity and theoretical invariances for their executions. Deep learning networks have shown the capacity to learn representations of the inputs to these classic algorithms, but naively predicting the output representation provides no guarantee of correctness or efficiency. For instance, consider m lists to be sorted. With a sorting algorithm such as merge sort, we are guaranteed $o(n \log n$ time complexity and a sorted list as output for each of the m lists. However, if we consider the same m lists and train a neural network to correctly sort 80%, we are provided no complexity or correctness guarantees for any given remaining 20%.

As mentioned above, prior to the work conducted in [15], neural algorithmic reasoning tended to rely on using only the representation of an algorithms input as input to the deep learning model. [15] incorporated the steps of a given algorithm's execution into the representation of the state of a graph algorithm by adding node features specific to each algorithm. The reasoning behind this approach is that classical computer science algorithms such as BF and BFS share common execution flows or subroutines, such as each node retrieving information about neighboring nodes by enumerating all of

its edges. Intuitively, allowing these subroutines to become 'visible' in a multi-task or transfer setting should allow the model to form better representations of the execution flow of a given algorithm task and improve performance across all tasks.

Our work seeks to build on this intuition by incorporating the idea that the order of task training impacts performance. We want to show that the schedule used in training both impacts performance, and that a schedule using simpler training examples first will allow the model improve its performance across multiple tasks.

2 Problem definition and formalization

The problem studied here is the same as Xhonneux et al. in [17]. We study the learning of graph algorithms that take in a graph G, a weight function $w : E \leftarrow \mathbb{R}$ that assigns a weight to each edge in the graph, and node features $X : V \leftarrow \mathbb{R}^k$. Algorithms compute an input $Y : V \leftarrow F^k$ and a predecessor $P : V \leftarrow V$ at each time step. This encompasses a large class of graph algorithms solving tasks such as reachability and shortest-path.

The graph neural network receives a sequence of $T \in \mathbb{N}$ graph-structured inputs. G will be fixed over each element in the sequence, while internal states associated with the nodes of G vary as they reflect the intermediate node-states kept by the classical graph algorithm that is being learned. To provide a graph $G = (V, E)$ as input to the graph neural network, each node $i \in V$ has associated node features $\mathbf{X}_i^{(\tau)} \in \mathbb{R}^{N_x}$ where $\tau \in 1, ..., \mathcal{T}$ denotes the index in the input sequence and N_x is the dimensionality of the node features. At each step τ, the algorithm produces node-level outputs $\mathbf{Y}_i^{(\tau)} \in \mathbb{R}^{N_y}$. Some of these outputs may then be reused as inputs on the next step.

The graph neural network follows the encode-process-decode architecture. The specific architecture of this graph neural network is the same as that described in [17].

We are interested in learning a graph neural network that can execute one more of several potential algorithms. By training the graph neural network simultaneously on multiple tasks, we hope that the shared subroutines across graph algorithms will lead to positive knowledge transfer between the tasks. Xhonneux et al. use the aggregate loss of all tasks to optimize the parameters ϕ at each training iteration [17]:

$$\min_{\phi} \sum_{t=1}^{T} \mathcal{L}(\phi, \mathcal{D}_t) + \lambda\Omega(\phi)$$

where $\mathcal{D}_t, t \in \{1, ..., T\}$ are the datasets corresponding to each task and $\lambda\Omega(\phi)$ is the regularization term. We replace this optimization method with Sequential Reptile [9], which consists of an outer learning trajectory and a nested inner learning trajectory. For each iteration in the outer learning trajectory, we compute the meta gradient, $\mathrm{MG}_t(\phi)$, and use it to update the model parameters:

$$\phi \leftarrow \phi - \eta \cdot \mathrm{MG}(\phi)$$

To compute $\mathrm{MG}(\phi)$ at each outer iteration, we update a copy of the model parameters over K iterations of the inner learning trajectory. For each inner iteration, k, we randomly sample a task index $t_k \in \{1, ..., T\}$ and a corresponding mini-batch $\mathcal{B}_{t_k}^{(k)}$, and then sequentially update $\theta^{(k)}$ as follows:

$$\theta^{(0)} = \phi, \;\; \theta^{(k)} = \theta^{(k-1)} - \alpha \frac{\partial \mathcal{L}(\theta^{(k-1)}; \mathcal{B}_{t_k}^{(k)})}{\partial \theta^{(k-1)}}, \;\; \text{where } t_k \sim \mathrm{Cat}(p_1, ..., p_T)$$

$\mathrm{Cat}(p_1, ...p_T)$ is a categorical distribution parameterized by $p_1, ...p_T$, the probability of selecting each task. The meta gradient is calculated using θ after the final inner iteration: $\mathrm{MG}(\phi) = \phi - \theta^{(K)}$.

Lee et al. Taylor expand $\mathrm{MG}(\phi)$ to show that

$$\mathbb{E}[\mathrm{MG}(\phi)] \approx \frac{\partial}{\partial \phi} \mathbb{E} \left[\sum_{k=1}^{K} \mathcal{L}(\phi; \mathcal{B}_{t_k}^{(k)}) - \frac{\alpha}{2} \sum_{k=1}^{K} \sum_{j=1}^{k-1} \left\langle \frac{\partial \mathcal{L}(\phi; \mathcal{B}_t^{(k)})}{\partial \phi}, \frac{\partial \mathcal{L}(\phi; \mathcal{B}_t^{(j)})}{\partial \phi} \right\rangle \right]$$

where $\langle \cdot, \cdot \rangle$ is the inner product [9]. We can see that $\mathbb{E}[\mathrm{MG}(\phi)]$ minimizes the aggregate loss over all tasks (first term) and maximizes the inner products between gradients computed in different batches (second term). The parameter α determines the trade-off between these two terms.

3 Related Work

3.1 Neural Execution of Graph Algorithms

Many papers have done comprehensive research on representing the execution of algorithms using deep learning frameworks [8, 12, 13, 18]. With the rise of the graph neural networks and the graph representation learning, recent works further investigate executing algorithms using graph neural networks [2, 3, 6, 15]. As previously mentioned, [15] focuses on graph algorithms and found that message passing neural networks (MPNN) with maximization aggregation functions are best for simulating the process of making decisions within neighborhoods and capturing algorithmic subroutines.
The work carried out in [15] utilized an encode-process-decode framework where the encoder f_A and decoder g_A are task A specific, and the processor P shares its parameters across all tasks $A_1 ... A_n$. The termination of a given algorithm is handled by a task-specific termination network T_A which provides the likelihood of termination. f_A, g_A, and T_A are all linear projections, giving most of the power to the processor P, which follows the aforementioned MPNN with maximization aggregation function framework.

3.2 Sequential Reptile

Meta-learning has been shown to be a viable approach to challenges currently facing machine learning research, such as data bottlenecks and generalization to unseen data distributions [7]. In that vein, Reptile [10] is an efficient first-order meta-learning method suitable for large-scale learning scenarios and it is similar to model-agnostic meta-learning (MAML). [5]. Subsequent work led to the formulation of Sequential Reptile, which improves upon its predecessor by aligning dissimilar task gradients by composing the inner-learning optimization with all tasks. Sequential Reptile has been shown to improve well in multi-task learning settings such as multilingual Question Answering (QA) and Named Entity Recognition (NER) [9]. In this work, we incorporate Sequential Reptile into our multi-task training step to align the gradients of the algorithm tasks we consider.

4 Methodology

Veličković et al. [15] has observed positive knowledge transfer when training multiple graph algorithms that share similar subroutines with vanilla multitasking. In this work, we seek to maximize knowledge sharing in the learning space among different algorithms to enable more efficient training and accurate predictions for each task. To this end, we propose to apply sequential reptile[9], a multitask training algorithm designed for multilingual learning, to the domain of algorithmic learning. In addition to vanilla sequential reptile, where for each step in the inner loop, the next task is randomly sampled, we experimented with fixed order training. Figure 1 shows the comparison between applying different methods. For each inner loop, we train on mini-batches from a single algorithm alone, and then followed by the next task. The setting is analogous to the curriculum training procedure in [15], which reflects whether knowledge learned from one task alone benefits the learning of the other task.

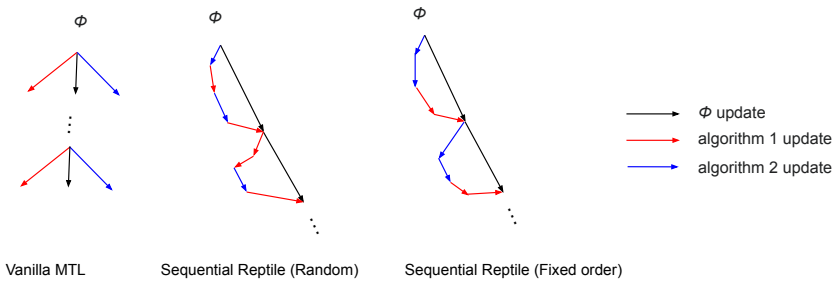

Figure 1: Comparison between different methods

4.1 Architecture

We follow the encode-process-decode architecture proposed in [17]. The architecture keeps a hidden state for each node $\mathbf{H^{(t)}} = \{h_i^{(t)}\}_{i \in V}$, where $\mathbf{H^{(0)}} = \mathbf{0}$. For each algorithm A, there is a latent encoder f_A and an edge encoder ε_A. The edge encoder ε_A consists of a linear layer and maps from edge features at step t to latent edge embeddings $\varepsilon_A : E^{(t)} \rightarrow E_e^{(t)}$. The latent encoder operates on each edge e_{ij} and considers both the nodes features and hidden states of nodes the edge connects to $[h_i^{(t)}, x_i^{(t)}, e_{ij}^{(t)}, h_j^{(t)}, x_j^{(t)}]$. Additionally, the latent encoder consists of a linear and non-linear encoder in parallel that are added together, which allows for more expressiveness. The output $\mathbf{Z^{(t)}}$ of the latent encoder f_A is sent to the processor network that is shared by all algorithms. The processor network P is a message passing neural network (MPNN) with a max aggregator with linear message and update functions. The processor compute the new hidden states $\mathbf{H^{(t+1)}}$ from $\mathbf{Z^{(t)}}$. Then the algorithm specific decoder D_A, which consists of a linear layer, predicts the next node features from the concatenation of $\mathbf{X^{(t)}}$, $\mathbf{H^{(t)}}$, $\mathbf{H^{(t+1)}}$. Besides the decoder that generates node state outputs, a linear predecessor prediction network S_A is used to predict for each node v, which of the neighboring nodes reaches v from the source at the current step. The predecessor prediction network only utilizes the latest hidden states $\mathbf{H^{(t+1)}}$. The architecture also predicts whether the running algorithm should terminate at the current step, with another MPNN network followed by a linear a layer $T : \mathbf{H^{(t+1)}} \times E_e^{(t)} \rightarrow \{0, 1\}$ that is algorithm agnostic. The MPNN layer operates on node level hidden states and the updated states are averaged across nodes. The averaged result is sent to the linear layer followed by sigmoid activation.

4.2 Loss Functions

We use the same loss function structures as [15, 17]. Since there are numerous evaluations unique to each algorithm, which will be explained in concrete details in Section 5.2. For Bellman-Ford, the prediction of the predecessor node utilizes smooth L1 loss and for the prediction of the intermediate distance information, it uses L2 loss. Breath-First-Search uses binary cross-entropy loss for reachability. Additionally, both algorithms use cross-entropy for termination accuracy.

4.3 Gradient Update Rules

The gradient and meta-gradient update rules are based on sequential reptile [9], where distinct variations are specified in detail below. Additionally, the result of each variant will be reflected in the experiment section below. Refer to section 2 above for the concrete formalism of sequential reptile. In the **Random** setting, we leverage the sampling strategy from sequential reptile [9], where we randomly sample a task with its associated mini-batch from a categorical distribution, between Bellman-Ford and Breadth-first search, within each inner step K per epoch, where we sequentially update the gradient for the K steps. Then we utilize the gradient to make a final update of the meta-gradient for the given epoch to estimate the model parameter.The **BF** $\rightarrow$ **BFS** setting follows a similar procedure as the Random setting with a minor caveat where instead of randomly sampling tasks within each inner step for K steps, it deterministically selects tasks where the first half of the $\frac{K}{2}$ tasks are Bellman-Ford mini-batches while the second half are Breadth-first search mini-batches for the sequential order of updating the gradients. The **BFS** $\rightarrow$ **BF** setting is boarder line identical to the

previous setting, but the order to update the gradients sequentially is swapped between Breadth-first search and Bellman-Ford. Although the **1BF + 1BFS** setting is not a unique paradigm for updating the meta-gradients, it serves as a form of a verification for the correctness of the sequential reptile method where it should produce identical results as the baseline.

5 Experiment Design and Evaluation

5.1 Datasets

Since there was no well-established datasets for our task at the time this project was started, and the authors of the original work [15] did not release their datasets, we generated our own train, validation, and test datasets. We note that a relevant benchmark [16] has been released recently, and we may test our method on it. To create the data, we first randomly generated 1400 Erdos-Renyi [4] graphs with 20 nodes and 200 graphs with 50 nodes, and ran both Bellman-Ford and Breadth First Search on the generated graphs and collected the intermediate step outputs, including the reachability status and the termination status for Breadth First Search, and current distance to the source node, the predecessor node, and the termination status for Bellman-Ford. The 1300 pairs of execution trajectories with 20 node graphs are split into 1100, 100, and 200 sets for train, validation, and test respectively. The 1100 graph training set is further split into a 1000 and a 100 graph training sets, in order to study the cases where data is imbalanced. The 200 pairs of execution trajectories with 50 node graphs are used for testing entirely.

5.2 Evaluation Metrics

We strictly follow the same evaluation procedure as [15]. Due to the dissimilar nature and complexity of each algorithm, different metrics are adopted to evaluate the performance of each algorithm under different settings to infer a holistic evaluation of the proposed methods. Each unique test-set size will be evaluated under the following metrics. Lastly, the loss and validation curves are reported for various settings and training graph sizes. For **Breath-First-Search** the mean(accuracy averaged over all time steps) and last (accuracy evaluated only at the last step) step accuracy of predicting the reachability are recorded in the table below to demonstrate the performance of BFS. The termination accuracy represents the prediction relative to the ground-truth. For **Bellman-Ford** the mean and last step accuracy of predicting the predecessor node in the shortest-path algorithm indicate which edge should be selected to reach the given node, essentially measuring the correctness of the solution. The mean and last squared error for predicting the intermediate distance information, in addition to termination accuracy, is reported for Bellman-Ford in the table below as well.

5.3 Experimental Setup

5.3.1 Graph Generation

To provide a consistent setting for our work, we followed [15, 17] in using the Erdos-Renyi [4] graph type, which is $G = (V, E)$ such that $e \in E$ exists with a probability of $\min\left(\frac{log_2|V|}{|V|}, 0.5\right)$. Each edge in the graph is given a weight from the range $[0.2, 1]$, and in addition to the edges generated during graph construction, we follow [15, 17] by inserting a self-edge to each node in the graph to ease retention of a node's information through message passing.

5.3.2 Algorithms

Following [15, 17], we consider *breadth-first search* (BFS) for reachability and *Bellman-Ford* (BF) for shortest paths. To represent intermediate steps of BFS, each node in the network maintains a single-bit value representing whether the node is reachable from the source. The source node for BFS is randomly selected and is initialized to 1 to signify it is reachable, while all other nodes are initialized to 0. For the representation of BF, each node maintains a scalar value that represents the distance from the source. Similar to BFS, the source node for BF is randomly selected, but it is initialized to 0 to represent the distance from itself, while all other nodes are initialized to the longest shortest path in the graph (to provide a stable value to substitute for positive infinity). Additionally,

the output representation of BF incorporates a predecessor node, thus $y_i^{(t)} = p_i^{(t)} || x_i^{(t+1)}$, where $||$ is concatenation, as shown in [15, 17].

5.3.3 Sequential Reptile Settings

For each setting among **Random**, **BF** $\rightarrow$ **BFS**, and **BFS** $\rightarrow$ **BF**, we additionally study the influence of training data size on the model's performance. There are three data specifications. For Full Data, we used the 1000 graph training sets for both Bellman-Ford and Breadth First Search. For Imbalanced Data, we used 1000 graph training set for one of the algorithms and 100 graph training set for the other. When using imbalanced data on the random setting, p_t, which is the probability that task t will be used to update the inner loop model parameters at iteration k, $\theta^{(k)}$, is assigned a value which corresponds to the relative size of that task's training set (e.g. if BF has 1000 training graphs and BFS has 100 training graphs, $p_{\text{BFS}} = \frac{100}{1100}$). Since each performance outcomes have large variance, we ran all experiments with 10 runs and present the average performance across different runs.

For all multi-task learning experiments, we used an inner learning rate of $\alpha = 5 \times 10^{-4}$ and an outer learning rate of $\eta = 1$. For the inner training trajectory, we used Adam optimization. Because ADAM uses the state from the previous training iteration to adaptively tune the gradients for the next iteration, we applied the state from the last iteration in the inner training loop to ADAM optimization for the first inner training iteration in the next outer training iteration.

5.4 Results and Discussion

Table 1: BFS reachability accuracy (unbalanced data settings for sequential reptile methods: 1000 training graphs for BF, 100 for BFS)

	Reachability (mean step / last-step accuracy)	
Model	*20 nodes*	*50 nodes*
SEGA (BFS->BF)	**98.82% / 98.82%**	**99.70% / 99.70%**
SEGA (BF-> BFS)	98.82% / 98.82%	99.70% / 99.70%
SEGA (Random)	98.82% / 98.79%	99.70% / 99.70%
Neural Executor	98.63% / 98.05%	99.50% / 99.25 %

Breadth First Search Results: For each setting of algorithm ordering and data specification, BFS shows similar mean step and last step reachability accuracy results, on both 20 and 50 node test sets. Tabel 1 shows reachability accuracy results for different settings of SEGA with 1000/100 training graphs for Bellman-Ford/BFS. For the Neural Executor baseline, we used 1000 graph training set for both algorithms. The consistent performance may be due to the fact that predicting reachability, which is essentially a binary classification problem, is simple. Since the reachability status for node from the previous time step will given when making predictions for the current step, the model only needs to learn whether the node of interest of is previously reachable or connected to a reachable node, which is straightforward for the maximization MPNN.

Table 2: BFS termination accuracy (unbalanced data settings for sequential reptile methods: 1000 training graphs for BF, 100 for BFS)

	BFS (mean termination accuracy)	
Model	*20 nodes*	*50 nodes*
SEGA (BFS->BF)	75.83%	**79.05%**
SEGA (BF-> BFS)	49.92%	49.92%
SEGA (Random)	76.94 %	78.45 %
Neural Executor	**79.12%**	73.50%

The termination accuracy for BFS also shows strong consistency across different settings. An outlier is when the data is imbalanced: 1000/100 training graphs for Bellman-Ford/BFS, and with fixed ordering of training Bellman-Ford first in the inner loop, as shown in Table 2. The reason for this

phenomenon is not immediately clear without further experiments. One possible explanation is that the training of Bellman-Ford is apt to minimizing the error in shortest distance and predecessor prediction. Then when we training on Bellman-Ford first, it shifts the processor's weights towards the direction that is suboptimal for maximizing termination accuracy, and this effect is amplified when the input space of Bellman-Ford is much larger (more training graphs).

Table 3: BF Accuracy of predicting the shortest-path predecessor node at different test-set sizes (unbalanced data settings for sequential reptile methods: 1000 training graphs for BF, 100 for BFS)

	Predecessor (mean step accuracy / last-step accuracy)	
Model	*20 nodes*	*50 nodes*
SEGA (BFS->BF)	**85. 96 % / 83.57 %**	**82.91 % / 78.47 %**
SEGA (BF-> BFS)	74.02 % / 69.05 %	69.26 % / 65.02%
SEGA (Random)	80.07 % / 76.01 %	76.54 % / 73.33%
Neural Executor	79.50 % / 74.98 %	76.34 % / 74.32 %

Table 4: Mean squared error for predicting the intermediate distance information from Bellman-Ford, and accuracy of the termination network compared to the ground-truth algorithm, averaged across all timesteps. (unbalanced data settings for sequential reptile methods: 1000 training graphs for BF, 100 for BFS)

	B-F mean squared error / last step mean squared error/ mean termination accuracy	
Model	*20 nodes*	*50 nodes*
SEGA (BFS->BF)	**0.1373 / 0.1333 / 82.08 %**	0.3965 / 0.4632 / $\mathbf{86.13}$ %
SEGA (BF-> BFS)	0.2315 / 0.2701 / 69.35 %	0.4936 / 0.5054 / 71.95 %
SEGA (Random)	0.1661 / 0.1804 / 74.97%	$\mathbf{0.2998}/\mathbf{0.3125}/78.39\%$
Neural Executor	0.205 / 0.2796 / 79.12%	0.5235 / 1.9640 / 73.50 %

Bellman-Ford Results: We demonstrate our results on each setting of sequential reptile and compare them with the Neural Executor baseline trained on BFS and BF in tables 3 and 4. The first thing to note is that for all metrics (predecessor accuracy, mean squared error, and termination accuracy), SEGA performs better than the Neural Executor baseline, demonstrating its positive knowledge transfer capabilities. **BFS** $\rightarrow$ **BF** performs best for all metrics on 20-node graphs, and performs best for all metrics except mean squared error on 50-node graphs. Ordering BFS training iterations before BF training iterations on the inner learning loop likely led to better results because BFS is fundamentally a simpler task than BF which will be explored further in the conclusion section. Therefore, presenting BFS training data before BF training data leads to a more natural learning trajectory for the model.

Table 5: BF Accuracy of predicting the shortest-path predecessor node at different test-set sizes and train-set sizes (100 or 1000)

	Predecessor (mean step accuracy / last-step accuracy)/ termination accuracy	
Model	*20 nodes*	*50 nodes*
SEGA (BF only,train-set=1000)	**79.63% / 81.89 % /80.57 %**	**77.11 % / 77.58 % / 83.15 %**
SEGA (BF only,train-set=100)	78.02% / 67.27% / 65.30 %/	71.49 % / 59.76% / 59.58 %

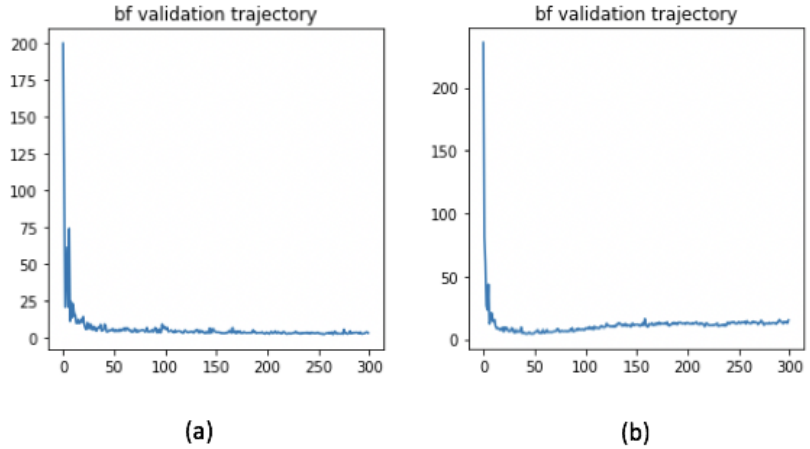

Figure 2: (a) validation loss for BF training on 1000 train-set (b) validation loss for BF training on 100 train-set

Bellman-Ford benefits from more data: Our experiments show that the performance of Bellman-Ford benefits from more training data, as in Table **??**, as opposed to BFS. This is possibly due to fact that Bellman-Ford has larger prediction space, since it has to predict real values for the shortest distance. In fact, the model can easily overfit if the training set size for Bellman-Ford is small, as shown in Figure , where we train Bellman-Ford alone with 100 and 1000 graphs, independently. With 100 training graphs, the model overfits at the early stage of training.

Table 6: BF Accuracy of predicting the shortest-path predecessor node at different test set graph sizes and different data settings (under setting SEGA: BFS->BF)

	Predecessor (mean step accuracy / last-step accuracy)	
Model	*20 nodes*	*50 nodes*
SEGA (BF=1000, BFS=1000)	79.13 % / 74.58%	75.31% / 70.70 %
SEGA (BF=100, BFS=1000)	80.37 % / 72.06%	75.60% / 66.79 %
SEGA (BF=1000, BFS=100)	**85.96 % / 83.57 %**	**82.91% / 78.47 %**

Bellman-Ford performs better for smaller graphs: We demonstrate the BF predecessor node accuracy for different training set sizes 6. Training with 1000 graphs for BF and 100 for BFS performs the best. The likely reason that BF=1000, BFS=100 performs better than BF=100, BFS=1000 is that BF is a more complex task and thus will need at least as much data as BFS. The likely reason that BF=1000, BFS=100 performs better than BF=1000, BFS=1000 is because the latter case puts too much weight on BFS during the optimization process and thus damages the performance of the more difficult task of BF.

5.5 Loss Curves

Based on the loss curves plotted below, we observe that Bellman-Ford (BF) tends to converge at a much slower rate relative to Breath-First-Search (BFS). Although, a common pattern that occurs throughout the experiments are the spikes for the validation loss of BFS and partly the training loss. Our hypothesis is that BFS can be considered as a speical case of shortest path algorithm like BF if the edge weights are set to 1, as a result BFS might be experiencing forgettability in the midst of training as the majority of the weights are consumed by learning the BF algoirhtm. We don't have strong theoretical results to support the given inference at the moment. Moreover, it's evident that training on large number of graphs, for instance 1000, for both algorithms produces the best performance. Overall, BFS tends to be more susceptibel for lower number of training graph even if BF is trained on smaller number of graphs.

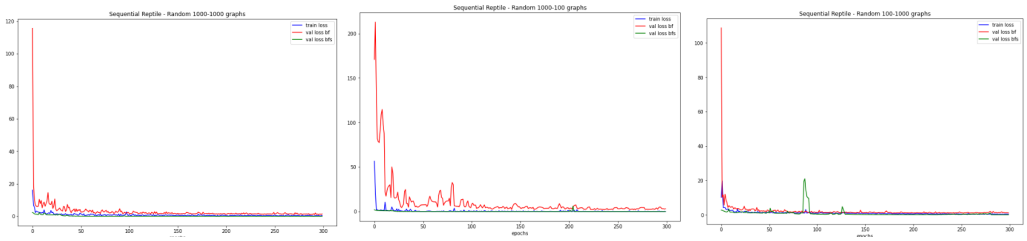

Figure 3: Random: BFS1000 BF1000(Left) BFS1000 BF100(Middle) BFS100 BF1000(Right)

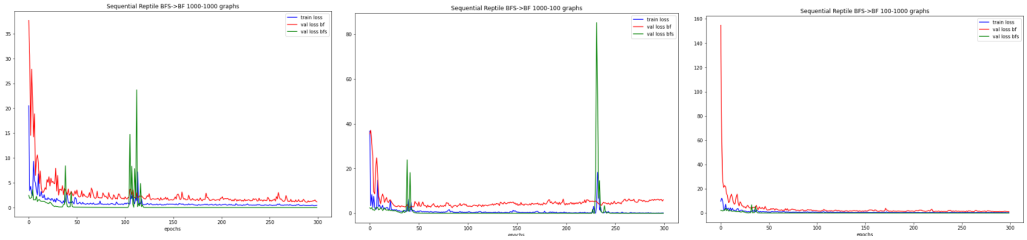

Figure 4: BF $\rightarrow$ BFS: BFS1000 BF1000(Left) BFS1000 BF100(Middle) BFS100 BF1000(Right)

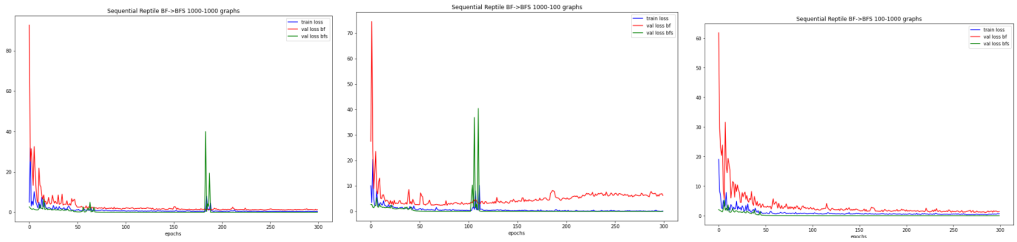

Figure 5: BFS $\rightarrow$ BF: BFS1000 BF1000(Left) BFS1000 BF100(Middle) BFS100 BF1000(Right)

6 Conclusions

In this work, we presented a novel approach to the task of neural execution of graph algorithms where we impose a schedule on the multi-task training framework laid out in [15]. Through our evaluation, we have found evidence that supports our claim that improving the representative power of shared algorithmic subroutines improves performance in a scheduled multi-task setting. Additionally, our results also indicate that a data imbalance in favor of the more "complex" task yields improved performance. We believe that the results shown here serve as a strong step forward in improving the neural representation of graph algorithms and will serve as a foundational example for future work in this area.

7 Task Distribution

Task	People
Evaluating and comparing models	Everyone
Implement Sequential Reptile for NE++	Jingdong, Armaan
SR/Baseline Codebase Contributions	Everyone
Writing report	Everyone
Creating presentation	Everyone
Hyperparameter Tuning	Ted, Alex, Yihang

References

[1] A. Buetti-Dinh, V. Galli, S. Bellenberg, O. Ilie, M. Herold, S. Christel, M. Boretska, I. V. Pivkin, P. Wilmes, W. Sand, M. Vera, and M. Dopson. Deep neural networks outperform human expert's capacity in characterizing bioleaching bacterial biofilm composition. *Biotechnology Reports*, 22:e00321, 2019.

[2] A. Deac, P. Velickovic, O. Milinkovic, P. Bacon, J. Tang, and M. Nikolic. Neural algorithmic reasoners are implicit planners. *CoRR*, abs/2110.05442, 2021. URL `https://arxiv.org/abs/2110.05442`.

[3] A. Dudzik and P. Veličković. Graph neural networks are dynamic programmers, 2022. URL `https://arxiv.org/abs/2203.15544`.

[4] P. Erdos and A. Renyi. On the evolution of random graphs. *Publ. Math. Inst. Hungary. Acad. Sci.*, 5:17–61, 1960.

[5] C. Finn, P. Abbeel, and S. Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017. URL `https://arxiv.org/abs/1703.03400`.

[6] D. Georgiev, P. Barbiero, D. Kazhdan, P. Velickovic, and P. Liò. Algorithmic concept-based explainable reasoning. *CoRR*, abs/2107.07493, 2021. URL `https://arxiv.org/abs/2107.07493`.

[7] T. M. Hospedales, A. Antoniou, P. Micaelli, and A. J. Storkey. Meta-learning in neural networks: A survey. *CoRR*, abs/2004.05439, 2020. URL `https://arxiv.org/abs/2004.05439`.

[8] K. Kurach, M. Andrychowicz, and I. Sutskever. Neural random access machines. *ICLR*, 2016. URL `http://arxiv.org/abs/1511.06392`.

[9] S. Lee, H. B. Lee, J. Lee, and S. J. Hwang. Sequential reptile: Inter-task gradient alignment for multilingual learning, 2021. URL `https://arxiv.org/abs/2110.02600`.

[10] A. Nichol, J. Achiam, and J. Schulman. On first-order meta-learning algorithms, 2018. URL `https://arxiv.org/abs/1803.02999`.

[11] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar. A survey on deep learning: Algorithms, techniques, and applications. 51(5), 2018. ISSN 0360-0300. doi: 10.1145/3234150. URL `https://doi.org/10.1145/3234150`.

[12] S. Reed and N. Freitas. Neural programmer-interpreters. 11 2015.

[13] A. Santoro, R. Faulkner, D. Raposo, J. Rae, M. Chrzanowski, T. Weber, D. Wierstra, O. Vinyals, R. Pascanu, and T. Lillicrap. Relational recurrent neural networks, 2018. URL `https://arxiv.org/abs/1806.01822`.

[14] P. Velickovic and C. Blundell. Neural algorithmic reasoning. *CoRR*, abs/2105.02761, 2021. URL `https://arxiv.org/abs/2105.02761`.

[15] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. Neural execution of graph algorithms, 2019. URL `https://arxiv.org/abs/1910.10593`.

[16] P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, and C. Blundell. The clrs algorithmic reasoning benchmark, 2022. URL `https://arxiv.org/abs/2205.15659`.

[17] L.-P. Xhonneux, A.-I. Deac, P. Veličković, and J. Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? In M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, editors, *Advances in Neural Information Processing Systems*, volume 34, pages 19500–19512. Curran Associates, Inc., 2021.

[18] W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL `http://arxiv.org/abs/1410.4615`.

[19] W. Zhou, Y. Yang, C. Yu, J. Liu, X. Duan, Z. Weng, D. Chen, Q. Liang, Q. Fang, J. Zhou, H. Ju, Z. Luo, W. Guo, X. Ma, X. Xie, R. Wang, and L. Zhou. Ensembled deep learning model outperforms human experts in diagnosing biliary atresia from sonographic gallbladder images. *Nature Communications*, 12(1):1259, 2021. doi: 10.1038/s41467-021-21466-z. URL `https://doi.org/10.1038/s41467-021-21466-z`.