Efficiently Learning Branching Networks for Multitask Algorithmic Reasoning

Dongyue Li[†], Zhenshuo Zhang[†], Minxuan Duan[†],
Edgar Dobriban[‡], and Hongyang R. Zhang[†]

[†]Northeastern University, Boston, Massachusetts
[‡]University of Pennsylvania, Philadelphia, Pennsylvania

July 30, 2026

Abstract

Algorithmic reasoning—the ability to perform step-by-step logical inference—is a synthetic benchmark for evaluating multi-step reasoning abilities, designed for graph neural networks and also for transformer models. Prior work has evaluated reasoning for executing a single algorithmic task, whereas a more desirable objective is to perform multiple algorithmic reasoning tasks simultaneously. We start by noting that this is inherently difficult due to differences arising from the execution traces of the algorithms (such as depth- vs. breadth-first search), which cause interference when they are trained together.

In this paper, we introduce branching neural networks, a new architecture for multitask algorithmic reasoning. The main idea is to search for a recursive tree-structured partition of n algorithmic tasks into a k-ary tree (divided into L layers). Naive search requires $O(k^{nL})$ complexity; we develop an algorithm that reduces this to $O(nL)$ by solving a convex relaxation at each layer to approximate an optimal partition. Our approach clusters these tasks using gradient-based affinity and can be used on top of any base model.

We validate our approach on algorithmic reasoning benchmarks and their extensions with text descriptions. We show that gradient-based affinity scores help estimate true performance with less than 5% error, measured across eight different architectures with up to 34 billion parameters. On the CLRS benchmark, our approach outperforms existing graph neural networks by 3.7% and baselines by 1.2%, while reducing runtime by 48% and memory usage by 26%. The learned branching structure shows a hierarchical clustering of related algorithms. On three text-based graph reasoning benchmarks, our approach improves over baseline methods by 3.2%. Finally, on a graph dataset with 21 million edges and 500 community labelings, we show a 28% accuracy gain over existing multitask and branching architectures, along with a 4.5× reduction in runtime.

1 Introduction

Reasoning is tied to a learning system's ability to make inductive inferences from its internal states, and it remains a central challenge in artificial intelligence [7]. Many formalisms—most notably probabilistic reasoning via Bayesian networks—have been developed to build models that can reason efficiently under uncertainty [32]. More recently, a complementary line of work has sought to

This is the full preprint of a paper that will appear in KDD 2026. Email correspondence: {li.dongyu, zhang.zhens, duan.mi, ho.zhang}@northeastern.edu and dobriban@wharton.upenn.edu.

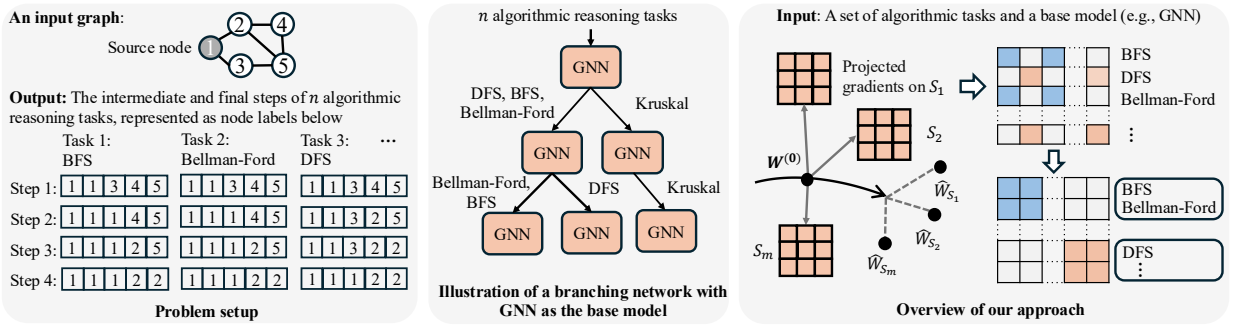

Figure 1: An illustration of our problem setup and the proposed solution. **Left:** We study learning graph algorithms such as BFS, Bellman-Ford, and DFS, formulating the prediction of intermediate algorithmic states as a node-labeling classification problem. **Center:** Given n algorithmic reasoning tasks and a base model (e.g., GNNs or low-rank adapters), we develop an efficient method to construct a branching network that automatically learns parameter-sharing structures across all tasks. **Right:** Our algorithm identifies a task partition at each layer and then searches for a corresponding tree structure over layers $1, 2, \ldots, L$. Overall, the procedure runs in time $O(nL)$—dramatically faster than the naive worst-case of $O(2^{nL})$.

characterize the reasoning ability of neural networks through the lens of algorithmic execution, viewing reasoning as the step-by-step computation performed by a classical algorithm. Surprisingly, for a wide range of basic graph algorithms such as shortest paths and reachability, neural networks can be trained to accurately predict not only the final output of the algorithm but also its intermediate states [37]. This raises a natural next question: can we design a model capable of solving multiple algorithmic reasoning tasks simultaneously? Since the number of possible combinations among n tasks grows combinatorially, the central challenge is to develop architectures and training methods that enable efficient and scalable multitask algorithmic reasoning.

A rigorous study of algorithmic reasoning is valuable not only from a foundational perspective [38], but also for its potential to improve the reasoning abilities of large language models (LLMs), especially on tasks involving graphs and structured computation [41]. In our experiments, we observe that prompting open-source LLMs (e.g., Llama-3-8B and Qwen-3-1.7B) with textual descriptions of algorithmic tasks, such as shortest paths, yields performance roughly 65% worse than directly training a graph neural network on the same task.

Several existing works have sought to design neural networks for tackling a single algorithmic reasoning task, while the problem of multitask algorithmic reasoning has not been studied in depth. Veličković et al. [37] introduce CLRS-30, a dataset of 30 algorithmic tasks, and find that message-passing neural networks can learn to accurately predict both the intermediate and final steps of a graph algorithm (where a graph is sampled from a fixed distribution). In particular, each intermediate step is treated as a node labeling sub-task, and the loss objective sums over all the intermediate node labeling sub-tasks (cf. equation (1), Section 2 for the definition). Consider breadth-first search (BFS) as an example (See also Figure 1 for an illustration). Starting from a graph and a source node for the search, at each intermediate step a network is trained to predict the index of the predecessor node on the current traversal path from the source. Ibarz et al. [14] conduct multitask experiments by training a single network across all CLRS-30 tasks, noting both positive and negative transfer compared to single-task training. Müller et al. [30] design a tri-attention mechanism in graph transformers, and use this new architecture to improve algorithmic reasoning over existing graph neural networks on a single task. Both works leave open the question of designing

specialized optimizers for multitask algorithmic reasoning.

A key insight of this paper is that training a single neural network for multitask algorithmic tasks is inherently difficult due to complex interference between the intermediate steps (or node labels) of different algorithms. For example, in Figure 1, on a toy graph example, we notice that BFS and Bellman-Ford share the same intermediate node labels, while depth-first search (DFS) differs in Steps 2 and 3. However, if we use a single processor to predict all three tasks, this interference is unavoidable (See our study in Figure 6, Section 4.4). In contrast, training a separate network for each task requires storing n models. This increases the memory usage at inference by a factor of n, since n networks must be evaluated.

To address this challenge, we explore the design of branching networks for multitask algorithmic reasoning by dividing algorithms into branches that account for their similarities. Searching over an L-layer, k-ary branching network for n tasks takes $O(k^{nL})$ time in the worst case. We introduce an efficient algorithm whose runtime is only $O(nL)$. Further, this algorithm can be applied on top of any base model, including GNNs or LLMs with low-rank adapters. A key technical ingredient of this algorithm is to recursively estimate the affinity scores among several algorithms, conditioned on the partitioning from the previous layers. In more detail, our algorithm involves two steps:

- First, we design an algorithm that, given a set of algorithmic reasoning tasks, partitions them into (at most) k groups via convex optimization. We quantify similarity scores using advances in the data attribution and influence functions literature [15, 31]. Crucially, we develop a new notion of layer-wise affinity scores, conditioned on the partitioning from previous layers. The main intuition of this procedure is that we can approximate the loss of a well-trained neural network via first-order approximations, a key geometric property of over-parameterized networks [16]. The runtime involves computing the gradients and functional values for the training samples of all tasks once at a pretrained initialization, leading to $O(n)$ runtime; additional computations, such as running the clustering algorithms, incur negligible cost.

- Second, we search for a branching network recursively from the first layer until layer L. Taken together, the overall running time of the algorithm is $O(nL)$. The benefit of this branching network is that the memory overhead over multitask training [14] is only $\frac{k}{L}$ (typically ≤ 2), making it amenable to larger network architectures such as edge transformers [30].

See Figure 1 for a high-level overview of our approach.

We evaluate our approach on various algorithmic reasoning tasks across graph and text datasets. First, we find that the gradient-based approximation can estimate the true performance with an error of less than **5**% for four GNNs and four LLMs with up to 34 billion parameters. Second, we apply AutoBRANE on the CLRS benchmark [37] using GNNs as the base model and observe that AutoBRANE outperforms the state-of-the-art single multitask network [30] by 3.7% on average across twelve graph algorithmic reasoning tasks. AutoBRANE improves over the most recent branching networks [11] and multitask networks [22] by 1.2% on average, while reducing GPU hours by **48%** and GPU memory by **26%**. In fine-tuning language models on three text-based graph reasoning benchmarks (including CLRS-Text [29], GraphQA [6], and GraphWiz [4]), AutoBRANE outperforms the strongest multitask learning baseline [22] by 3.2% on average. Additionally, we show that AutoBRANE extends beyond algorithmic reasoning, applying it to overlapping community detection (on a graph with over 21M edges and 500 node-labeling tasks) [42]. AutoBRANE improves accuracy by 28% over branching and multitask networks [11, 22], while achieving the best trade-off in runtime and memory.

Summary of contributions. This paper makes three contributions to the problem of algorithmic reasoning on graphs:

- First, we develop a fast approximation procedure to efficiently identify optimal parameter-sharing paradigms using a branching network, and we empirically validate that it accurately approximates true performance across various models.
- Second, we design an automatic branching network that improves average performance across multiple algorithmic reasoning tasks and can be built on any base model.
- Third, we evaluate our algorithm on graph- and text-based algorithmic reasoning tasks and show that it achieves competitive performance with the best trade-off between runtime and model memory.

Our work reinforces the importance of several open problems relating to algorithmic reasoning, in particular the learnability of an algorithm via neural networks [38, 26], and generalization in algorithmic reasoning [10]. It would be interesting to use the new techniques developed in this paper to better understand the sample complexity (and in-context learnability) of algorithmic reasoning tasks. The code for reproducing our findings can be found at https://github.com/VirtuosoResearch/Multitask-algorithmic-reasoning.

2 Preliminaries

We follow the definition of algorithmic reasoning from prior work [38, 37]. Let $A^{(1)}, A^{(2)}, \ldots, A^{(n)}$ denote n algorithms. Our goal is to learn a neural network f_W (such as a GNN) parameterized by W to minimize the risk averaged over the n algorithmic execution sequences. Given an input X, the encoding of the j-th intermediate step of $A^{(i)}$ on input X is denoted as $A^{(i)}_j(X)$, for $j = 1, 2, \ldots, S^{(i)}(X)$, where $S^{(i)}(X)$ denotes the total number of execution steps by running $A^{(i)}$.

Given an input X, an algorithmic reasoning task involves predicting $A^{(i)}_j(X)$ as accurately as possible, for all $j = 1, \ldots, S^{(i)}(X)$. Then, let ℓ denote a loss function (e.g., cross-entropy) over predictions $f^{(i)}_W(X; j)$ of $A^{(i)}$ at step j and ground truth labels $A^{(i)}_j(X)$. The test loss $L(f_W)$ of f_W is defined as:

$$\mathop{\mathbb{E}}_{X\sim\mathcal{D}}\left[\frac{1}{n}\sum_{i=1}^{n}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\ell\left(f^{(i)}_W(X;j), A^{(i)}_j(X)\right)\right], \tag{1}$$

where $\mathcal{D}$ denotes a fixed distribution from which X is drawn. For example, in the case of graphs, the input X corresponds to a graph, and the labels are represented as node labels. $\mathcal{D}$ can be Erdös-Rényi random graphs or preferential attachment graphs. For each algorithm, the node labels are generated by running the algorithm on an input graph. Then, the node traversal trajectories at each step are recorded as labels.

To help understand the above definition, consider the example illustrated in Figure 2 on a toy graph with five nodes, for three algorithms (breadth-first search, depth-first search, and Bellman-Ford). To understand the encoding, each intermediate step is labeled by the predecessor of each node along the traversal path, yielding a node-label vector for each step. The node labels are initially the indices of the nodes.

- In BFS, starting from node 1, the first step visits node 2. Hence, we update node 2's label to 1, its predecessor. Step 2 visits node 3, hence its node label is updated as 1. Step 3 visits node 4 (with predecessor 2) and Step 4 visits node 5 (with predecessor 2).

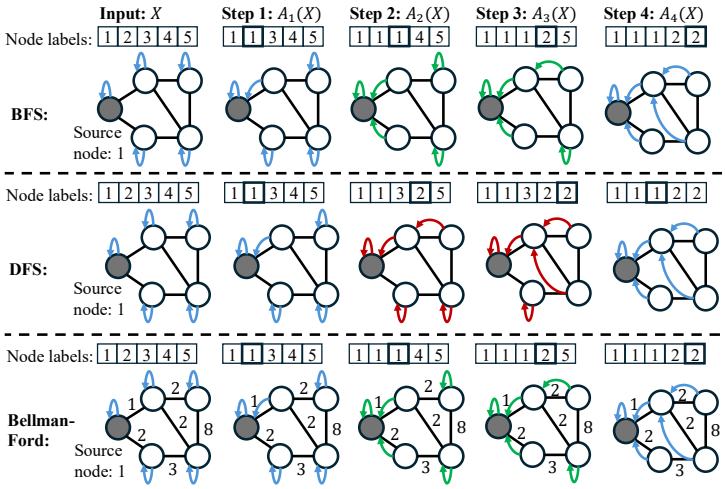

Figure 2: We give three examples of algorithmic reasoning tasks: Breadth-first search (BFS), depth-first search (DFS), and Bellman-Ford. The node labels of each intermediate step encode the predecessor of each node along the traversal path. We illustrate the predecessors with arrows, and we use the integers on edges to indicate the edge weights. One can see that these algorithms share some intermediate steps, but not all, and the goal of this paper is to automatically identify such similarities.

- In DFS, the first step visits node 2, whose predecessor becomes 1. Step 2 visits node 4, whose predecessor is 2. Step 3 visits node 5 from 4. Step 4 visits node 3 from 1.
- Notice that in step 2, BFS and Bellman-Ford follow the same traversal path, but DFS follows a different path.

Problem statement. Given n algorithmic reasoning tasks, can we design a neural network to predict the logical steps of all algorithms simultaneously? Notice that in the example of Figure 2, BFS and Bellman-Ford share the same node labels at every step, whereas DFS shares only the first step with BFS. More generally, we can expect algorithms that follow a similar logic to share similar node labels in the intermediate steps. Thus, the goal of *multitask algorithmic reasoning* is to design a neural network f_W that can minimize the test loss of $L(f_W)$ above.

As an extension of the above problem, suppose we are given a textual description of the same algorithmic reasoning task; the LLM's output would be to generate the correct execution steps for each algorithm.

3 Our Approach

We now describe a new algorithm that automatically learns one neural network for multitask algorithmic reasoning. Our approach involves learning a branching network that can be applied on top of any base model, enabling more flexible parameter sharing based on estimated task similarity scores. For example:

- For GNNs, one can instantiate multiple networks per layer and assign a specific GNN to each task at each layer. See Figure 3 for several examples of branching GNNs.
- For LLMs, one can apply parameter-efficient fine-tuning such as low-rank adapters (LoRA) [13], and design a branching structure of LoRAs on top of a pretrained LLM.

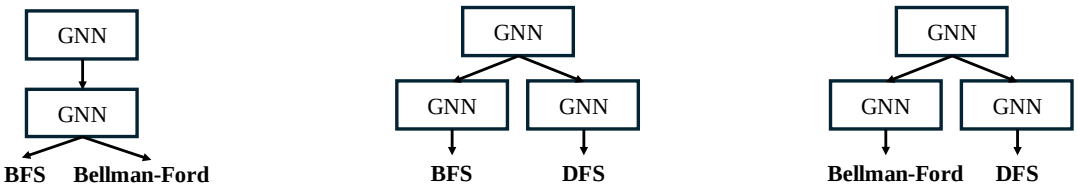

Figure 3: We present three examples of branching GNNs, each designed to learn a pair of algorithms. As shown in Figure 2, all three algorithms share identical node labels in the first step, so the same initial GNN layer applies to all. BFS and Bellman–Ford continue to share node labels in steps 2 and 3, thus reusing the second layer, while DFS branches out. These structures are learned automatically from data: across the training graphs, BFS and Bellman–Ford share 75% of their node labels, whereas DFS overlaps with each by 30%.

Specifically, consider building a network with L layers. Training a separate network for each task requires storing n models, resulting in a memory cost of $O(n)$, which can be expensive for storing larger network architectures such as edge transformers [30]. Memory reduction can be achieved by constructing a branching network that enables parameter sharing across selected layers across tasks. If each layer has at most k possible branches, then at each layer, there are $O(k^n)$ options for assigning n tasks into k branches. Iterating over layer 1 to L leads to $O(k^{nL})$. By contrast, our approach reduces the runtime to $O(nL)$ by using a top-down search to recursively partition tasks from the first to the last layer. Each search step, which runs in $O(n)$ time, estimates task affinity scores based on the task partition inherited from the previous layer. We use W to denote all the parameters of the branching network, and let W_i denote the parameters at the i-th layer, for $i = 1, \ldots, L$.

3.1 Partitioning at One Layer

We start by describing a procedure that, given $S \subseteq \{1, \ldots, n\}$ at layer l, determines a disjoint partition of S, corresponding to the branching structure at layer $l+1$. This procedure first estimates task affinity scores using partitions inherited from previous layers $1, 2, \ldots, l$, and then maximizes these estimates via a convex relaxation program.

Let S be a set of tasks that contain i, j, sampled from $\{1, 2, \ldots, n\}$. Let $\hat{L}^{(i)}(f_W(S))$ denote the validation loss of task i of $f_W(S)$ (trained on S). Let $S_1, S_2, \ldots, S_m$ denote m randomly sampled subsets from $\{1, 2, \ldots, n\}$. We define the affinity score between i, j, conditioned on layer l, as:

$$T^{(i)}_{j,l} := \frac{1}{m_{i,j}} \sum_{S_k:\ i \in S_k, j \in S_k} \hat{L}^{(i)}\big(f_W(S_k)\big), \tag{2}$$

where $m_{i,j}$ is the number of subsets S_k where i, j are both in S_k. This score is analogous to the feature importance score used in random forests [19, 20, 21].

Computing T requires training f_W m times. Instead, we design an algorithm that estimates T *without repeated training*. The key idea is to use a first-order approximation of the network output around an initialization $W^{(0)}$, with respect to the weights from layer l up to L:

$$f^{(i)}_W(X; j) = \ f^{(i)}_{W^{(0)}}(X; j) + \left[\nabla_{l:L} f^{(i)}_{W^{(0)}}(X; j)\right]^\top \left(W_{l:L} - W^{(0)}_{l:L}\right) + \epsilon^{(i)}_{X,j}, \tag{3}$$

for any step $j = 1, \ldots, S^{(i)}(X)$, where we recall that $S^{(i)}(X)$ is the number of execution steps of $A^{(i)}(X)$, and $W_{l:L}$ denotes the weights from layer l up to L. To obtain this initialization, we

Algorithm 1 FastApproxPar (Fast Approximate Partitioning)

Input: Training/validation data for a subset of tasks S; Layer l

Require: Network parameters W; Number of subsets m and their sizes α; Projection dimension d; Number of groups g

Output: A disjoint partition of S, a trained initialization $f_{W^{(0)}}$

$f_{W^{(0)}} \leftarrow$ A meta-initialization trained from W on S, with layers $1, 2, \ldots, l-1$ fixed
$S_1, S_2, \ldots, S_m \leftarrow m$ random subsets of S with size α
$W_{l:L} \in \mathbb{R}^p \leftarrow$ Parameters of layers l to L
$P \leftarrow$ A p by d isotropic Gaussian random projection matrix
for each training sample X, $j = 1, \ldots, S^{(i)}(X)$ **do**
    $\tilde{g}_j^{(i)} \leftarrow P^\top \nabla_{W_{l:L}} f_{W^{(0)}}^{(i)}(X; j)$, $\forall i \in S$
end for
for $k = 1, \ldots, m$ **do**
    $\hat{W}_d \leftarrow \arg\min \sum_{i \in S_k} \sum_X \tilde{\ell}(f_W^{(i)}(X))$
    $\hat{W}_{S_k} \leftarrow W^{(0)} + P\hat{W}_d$
    $\hat{L}^{(i)}\Big(f_{\hat{W}_{S_k}}(S_k)\Big) \leftarrow$ Validation loss of $f_{\hat{W}_{S_k}}$ on $A^{(i)}$, $\forall i \in S_k$
end for
$T \leftarrow |S|$ by $|S|$ affinity score matrix via equation (2)
$S_1, S_2, \ldots, S_g \leftarrow$ A disjoint partition of S via clustering on T
Return $(S_1, l+1), (S_2, l+1), \ldots, (S_g, l+1)$; $f_{W^{(0)}}$

first train f_W from layer l up to L jointly on all tasks of S. As the partition is conditioned on the structures before layer l, we keep the weights in the first $l-1$ layers frozen. This leads to an initialization $f_{W^{(0)}}$. Our experiments (See Table 3, Section 4.2) show that this approximation yields less than 5% relative errors on algorithmic reasoning tasks across four GNNs and four LLMs. This estimation leverages the fact that a well-trained neural network uses its gradients as features [16]. Hence, the linear approximation is accurate around a local vicinity of the initialization [21].

Building on the above approximation, we propose to estimate $\hat{L}^{(i)}(f_W(S_k))$, for any $k = 1, \ldots, m$, given $W^{(0)}$. We estimate the model weights trained on a subset of tasks (denoted as $\hat{W}_{S_k}$) using the gradients evaluated at the initialization. For simplicity, we illustrate the case of binary classification, and the same logic applies to multi-class and regression problems. By applying the first-order approximation in equation (3) to the training loss, we solve a logistic regression problem using the gradients as features, which minimizes the approximated log-loss on the training examples from a subset of tasks:

$$\tilde{\ell}(f_W^{(i)}(X)) = \sum_{j=1}^{S^{(i)}} \log\Big(1 + \exp\big(-A_j^{(i)}(X)\big[g_j^{(i)}\big]^\top \Big(W_{l:L} - W_{l:L}^{(0)}\Big) - A_j^{(i)}(X) f_{W^{(0)}}^{(i)}(X; j)\big)\Big),$$

where $g_j^{(i)} = \nabla_{l:L} f_{W^{(0)}}^{(i)}(X; j)$. In practice, we use random projections of the gradients, following the Johnson-Lindenstrauss lemma. Solving the above logistic regression problem, we evaluate the model with the estimated model weights $\hat{W}_{S_k}$ to obtain $\hat{L}^{(i)}(f_W(S_k))$.

Repeating the estimation to m randomly sampled subsets leads to an approximation of $T_{j,l}^{(i)}$ after averaging over subsets including tasks i and j. The output of this step is an $|S| \times |S|$ task affinity matrix. Crucially, this step runs in $O(|S|)$ time, which involves training $W^{(0)}$ and evaluating the gradients on the samples in S. Additional costs include running m logistic regressions, which can be performed on CPUs and completed in a few seconds.

Algorithm 2 AutoBRANE (Automated BRAnching NEtworks)

Input: Training and validation datasets for n algorithmic tasks
Output: A branching neural network f_W

$f_W \leftarrow$ An L-layer network with initialized parameters W, with one model at each layer
$Q \leftarrow \{(\{1, 2, \ldots, n\}, 1)\}$
while $Q \neq \varnothing$ **do**
  $S \leftarrow$ Dequeue one subset from Q with layer index l
  if $l < L$ **then**
    $(S_1, l+1), \ldots, (S_k, l+1)$; $f_{W^{(0)}} \leftarrow \textsc{FastApproxPar}\big(S, l; W\big)$
    $W_l \leftarrow W_l \cup \left\{W_l^{(0)}\right\}$ ▷ Update the branching network at layer l with weights trained on S
    $Q \leftarrow Q \cup \big\{(S_1, l+1), \ldots, (S_k, l+1)\big\}$
  end if
end while
Return f_W

Finally, we apply a convex relaxation program to maximize the average affinity score density within clusters. We adjust the cluster size by regularizing the trace of the assignment variable. The details are provided in Appendix B.1. This clustering step takes less than a few seconds. In practice, we can apply the clustering step several times to find the optimal cluster size. In summary, the partitioning procedure at one layer is described in Algorithm 1.

3.2 Learning Branching Structures

Next, we search for a branching network via a top-down procedure. The algorithm begins with a single network with one module per layer. Starting at $l = 1$, suppose that tasks are grouped into k_1 clusters. This creates k_1 modules at layer 1. If $k_1 = 2$, tasks are split into two groups, denoted as S_1 and S_2. The procedure continues recursively: Each group is further split at the next layer. If both are split into two groups, the second layer then contains four modules, denoted as S_3, S_4, S_5, and S_6. This continues until layer L. The full procedure is in Algorithm 2.

In terms of running time, at each layer the algorithm takes $O(n)$ time to find a partition, since the union of the sets is at most n. In total, Algorithm 2 takes $O(nL)$ time. Regarding memory usage, suppose the last layer contains k clusters, and at each layer, the number of clusters grows by a constant factor a. Then the total number of nodes in the tree is roughly $\sum_{i=0}^{L-1} a^i = \frac{k^{L/(L-1)}-1}{k^{1/(L-1)}-1}$, where $a^{L-1} \approx k$.

Let the running time for training a single L-layer network on a single task be T, and the memory usage be B. In summary, our algorithm runs in time nLT and uses approximately $\frac{kB}{L}$ memory to store the network. For comparison, training a separate network for each task (i.e., single-task learning) requires nT time and nB memory to store n models. A mixture-of-experts architecture with k networks requires knT time and kB memory. Task affinity grouping [8] with fully-computed affinity scores takes n^2T time and kB memory. LearningToBranch [11] takes $k^L nT$ time and uses nB memory since it trains with n modules per layer. This comparison is summarized in Table 1.

Remark 3.1. Our approach does not assume that tasks exhibit hierarchical dependencies. The branching design applies when tasks exhibit various levels of similarity, where more similar tasks share more layers, and dissimilar tasks share fewer. The procedure can, in principle, handle any type of task dependencies.

Table 1: Summary of runtime and memory cost of AUTOBRANE and existing multitask and branching networks. Here n is the number of tasks, L is the number of layers, and k is the number of branches at the last layer. T denotes the runtime of training a single network of L layers for one model. B denotes the memory usage of one base model.

Methods	**Runtime**	**Memory**
Single task learning (STN)	nT	nB
Multitask learning (MTN)	nT	B
Multi-gate MoE (MMoE) [28, 12]	knT	kB
Task affinity grouping (TAG) [8, 19]	n^2T	kB
LearningToBranch [11]	k^Lnt	nB
AUTOBRANE (Algorithm 2)	$\boldsymbol{nLT}$	$\boldsymbol{\frac{k}{L}B}$

Additionally, this approach can be extended to incorporate new tasks into the branching network. One way is to determine the group partition of the new tasks, layer by layer, by incrementally fine-tuning each layer on the new tasks and estimating its task affinities. Our approximation procedure can also be extended to handle this extension.

Remark 3.2. For LLMs, our approach constructs a single branching structure of LoRA adapters on top of each model, which can be computed efficiently using a fast approximation algorithm. Note that the approach can be applied with other parameter-efficient fine-tuning methods besides LoRA to construct the adapters.

3.3 Theoretical Analysis

In this section, we show that the error introduced by the approximation in FASTAPPROXPAR is bounded. We will show that the training loss of the estimated parameters $\hat{W}_S$ by solving the logistic regression is bounded.

Proposition 3.3. *Let $\mathcal{D}$ be a search space of model weights W whose radius is at most D. Suppose the gradient of $f_{W^{(0)}}$ at the initialization $W^{(0)}$ in the training set is at most G in Euclidean norm. Let T be the training set of inputs. For each algorithmic reasoning task $i \in \{1, 2, \ldots, n\}$, the training data is collected by executing the algorithm on each input. Suppose that for every task i, the average error of the linear approximation is bounded as follows:*

$$\frac{1}{|T|}\sum_{X\in T}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\epsilon_{X,j}^{(i)} \le \delta,$$

where

$$\epsilon_{X,j}^{(i)} = \left| f_W^{(i)}(X;j) - f_{W^{(0)}}^{(i)}(X;j) - \left[\nabla f_{W^{(0)}}^{(i)}(X;j)\right]^\top \left(W - W^{(0)}\right)\right|.$$

Let $\hat{L}_S(f_W)$ be the training loss of a subset of tasks $S \subseteq \{1, 2, \ldots, n\}$ defined as:

$$\hat{L}_S(W) = \frac{1}{|T|\,|S|}\sum_{X\in T}\sum_{i\in S}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\ell\left(f_W^{(i)}(X;j), A_j^{(i)}(X)\right).$$

Provided that the projection dimension $d = O\left(\frac{\log p}{\epsilon^2}\right)$*, the training loss of* $\hat{W}_S$ *is bounded away from the minimum training loss for any* S *as*

$$\hat{L}_S(\hat{W}_S) \leq \min_{W \in \mathcal{D}} \hat{L}_S(W) + 2\delta + 2GD\epsilon. \tag{4}$$

The proof is based on using the Johnson-Lindenstrauss Lemma [17]. The idea is to relate the loss $\hat{L}_S(\hat{W}_S)$ and $\min_{W \in \mathcal{D}} \hat{L}_S(W)$ by the 1-Lipschitz continuous property of the logistic loss. The error in the random projection is bounded by the JL Lemma, and the error in the linear approximation is bounded by Taylor's expansion error δ. As ϵ goes to zero, equation (4) guarantees the gap between $\hat{L}_S(\hat{W}_S)$ and $\min \hat{L}_S(W)$ will be small. We note that this error bound also applies when approximating certain layers by restricting the search space of the parameters in those layers.

4 Experiments

We now evaluate AutoBRANE on algorithmic reasoning tasks in various settings. The evaluation focuses on the following questions. First, how accurate is the first-order loss approximation for algorithmic reasoning tasks? Second, how well does AutoBRANE perform on the CLRS benchmark, and what hierarchical structures does it discover? Third, how effectively does AutoBRANE generalize to text-based graph reasoning and large-scale multitask scenarios?

Our experiments show that the first-order approximation estimates the losses of fully trained networks with a relative error of less than 5% across four GNNs and four language models. On twelve algorithmic tasks from the CLRS benchmark, AutoBRANE outperforms state-of-the-art networks by an average of **3.7**% and the strongest multitask baselines, LearningToBranch [11] and GradTAG [21], by **1.2**%, while reducing GPU hours by **48**% and memory usage by **26**%. AutoBRANE reveals branching structures that group tasks with similar intermediate steps. On three text-based graph-reasoning benchmarks, it achieves up to a **3.2**% gain over the strongest multitask baseline. Finally, on a large-scale community dataset comprising 500 labeling tasks and 21M edges, AutoBRANE achieves a **28**% relative improvement over the baselines, with the best trade-off between runtime and memory usage.

4.1 Experimental Setup

4.1.1 Datasets and Models

We evaluate AutoBRANE on various algorithmic reasoning tasks in both graph and text formats, focusing on performance, runtime, and memory trade-offs. First, we use the graph-format datasets of twelve graph algorithmic reasoning tasks in the CLRS benchmark [37]. The full list of tasks is described in Table 2. Following the protocol of CLRS, for each algorithm, we use a training dataset of 1,000 Erdös-Rényi graphs with 16 nodes and a validation set of 32 graphs with 16 nodes. We then evaluate performance on a test set of 32 graphs with 64 nodes. On CLRS, we evaluate AutoBRANE with the edge transformer [30], which achieves state-of-the-art performance. We also evaluate AutoBRANE using a standard MPNN [37] and graph attention networks [39].

Second, we evaluate AutoBRANE on text-based graph algorithmic reasoning tasks, where we construct a branching network of LoRA adapters to fine-tune LLMs. We first use the CLRS-Text benchmark [29], which encodes graphs as flattened adjacency lists and represents outputs as the intermediate steps from CLRS. We use the same twelve graph algorithms as in CLRS, with 1,000 training graphs of 10 nodes and 200 graphs each for validation and testing. Second, we evaluate on GraphQA [6], which comprises text-based graph reasoning tasks, including edge existence, node

Table 2: Definitions of input and intermediate steps of 12 graph-based algorithmic reasoning tasks from the CLRS benchmark. The input graphs are sampled from an Erdös-Rényi random graph distribution with edge sampling probability $p = 0.5$, with 16 vertices in total on each graph. The edge weights are uniformly sampled from $[0, 1]$. The source node is sampled randomly from the entire vertex set.

Task	Input Graph	Additional Input	Encodings of Intermediate Steps
Breadth-first search	Undirected	Source node	Node-level labels indicating its predecessor
Depth-first search	Directed	Source node	Node-level labels indicating its predecessor
Topological sort	Directed acyclic	None	Node-level label indicating its predecessor
Articulation points	Undirected	None	Node-level binary label for the articulation point
Bridges	Undirected	None	Edge-level binary label for the bridge
SCC Kosaraju	Directed	None	Node-level label for the SCC index
MST Kruskal	Undirected, weighted	None	Edge-level binary label for MST
MST Prim	Undirected, weighted	Root of MST	Edge-level binary label for MST
Dijkstra	Directed, weighted	Source node	Node-level labels indicating its predecessor
Bellman-Ford	Directed, weighted	Source node	Node-level labels indicating its predecessor
DAG shortest paths	Directed, acyclic, weighted	Source node	Node-level labels indicating its predecessor
Floyd-Warshall	Undirected, weighted	None	Edge-level labels indicating the source node

degree, and cycle detection. We use twelve tasks from this benchmark. Third, we evaluate on the datasets from GraphWiz [4], which constructs textual descriptions of the reasoning processes for solving graph-related tasks by prompting from GPT-4. We use the nine tasks from this dataset. The description of the tasks, their inputs, and outputs is provided in Appendix C. For the datasets, we fine-tune Qwen-3-1.7B and Llama-3-1B using LoRA [13] with rank 16.

Lastly, we evaluate AutoBRANE on a large-scale multi-label prediction task involving 500 labels. We use the Orkut social network dataset from SNAP [45], which contains 395K nodes, 21M edges, and 500 community labels. Each community is a subgraph of a graph. Given a partially labeled subgraph, the task is to predict the remaining node labels. For each community, we sample 10% of the nodes for training, 20% for validation, and use the remaining nodes for testing. As the encoder, we use a 3-layer SIGN model [9] with 256 hidden units, an efficient variant of GCNs.

4.1.2 Metrics

In the CLRS benchmark, we follow the official protocol and report task scores on the test set using the metric specified for each task, such as the F_1 score for node-label predictions. The F_1-scores are evaluated between model outputs and task labels at the final step. For text-based graph reasoning tasks, we evaluate test accuracy between model outputs and task labels at every intermediate step. In community detection, we evaluate the average macro F_1-score over 500 tasks.

We compute the average task score across metrics such as accuracy or F_1-score over all tasks and report the error rate as one minus the average task score. We measure the efficiency in GPU hours and GPU memory usage in GB.

4.1.3 Implementations

Recall that AutoBRANE requires setting the number of subsets m, the subset size α, the gradient projection dimension d, and the number of clusters g. To determine the task clusterings, we tune the hyperparameters sequentially, one at a time. We begin by increasing the number of subsets until the affinity scores converge. For task counts n from 12 to 500, we vary m between 200 to 5000 and vary α between 3 to 25. We observe that affinity scores typically stabilize once m reaches over $10n$.

Table 3: We measure the approximation error of losses on algorithmic reasoning datasets using a first-order Taylor expansion around the initialization trained on all tasks. Results are averaged over 50 random subsets of algorithmic reasoning tasks. We measure the errors across four GNNs, including MPNN [37], GAT [39], Triplet-GMPNN [14], and Edge Transformer [30]. We measure four LLMs with up to 34 billion parameters for text-based reasoning tasks (by applying the approximation to LoRA parameters).

	MPNN	GAT	Triplet-GMPNN	Edge Transformer
Dist.	RSS	RSS	RSS	RSS
2%	$2.3_{\pm 0.7} \times 10^{-3}$	$1.2_{\pm 0.1} \times 10^{-3}$	$3.6_{\pm 0.2} \times 10^{-4}$	$3.9_{\pm 0.4} \times 10^{-3}$
4%	$7.0_{\pm 0.8} \times 10^{-3}$	$1.3_{\pm 0.2} \times 10^{-3}$	$1.6_{\pm 0.2} \times 10^{-3}$	$6.5_{\pm 0.5} \times 10^{-3}$
6%	$8.2_{\pm 1.4} \times 10^{-3}$	$1.6_{\pm 0.4} \times 10^{-3}$	$3.9_{\pm 0.7} \times 10^{-3}$	$7.7_{\pm 1.6} \times 10^{-3}$
8%	$8.6_{\pm 1.2} \times 10^{-3}$	$2.3_{\pm 1.4} \times 10^{-3}$	$4.1_{\pm 0.3} \times 10^{-3}$	$8.4_{\pm 2.4} \times 10^{-3}$
10%	$9.2_{\pm 2.4} \times 10^{-3}$	$3.8_{\pm 2.0} \times 10^{-3}$	$7.9_{\pm 1.2} \times 10^{-3}$	$9.6_{\pm 1.1} \times 10^{-3}$
	Qwen-3-1.7B	Llama-3-8B	Qwen-3-14B	CodeLlama-34B
Dist.	RSS	RSS	RSS	RSS
2%	$3.9_{\pm 1.8} \times 10^{-3}$	$4.8_{\pm 1.9} \times 10^{-3}$	$2.0_{\pm 1.1} \times 10^{-3}$	$2.0_{\pm 1.8} \times 10^{-3}$
4%	$5.5_{\pm 2.1} \times 10^{-3}$	$4.9_{\pm 2.0} \times 10^{-3}$	$2.1_{\pm 1.4} \times 10^{-3}$	$2.0_{\pm 1.7} \times 10^{-3}$
6%	$1.1_{\pm 0.4} \times 10^{-2}$	$5.5_{\pm 2.0} \times 10^{-3}$	$2.5_{\pm 1.6} \times 10^{-3}$	$2.1_{\pm 1.7} \times 10^{-3}$
8%	$2.3_{\pm 0.9} \times 10^{-2}$	$7.1_{\pm 2.0} \times 10^{-3}$	$2.6_{\pm 1.8} \times 10^{-3}$	$2.4_{\pm 1.7} \times 10^{-3}$
10%	$4.6_{\pm 1.6} \times 10^{-2}$	$9.0_{\pm 3.0} \times 10^{-3}$	$3.0_{\pm 1.6} \times 10^{-3}$	$2.7_{\pm 1.9} \times 10^{-3}$

Next, we compute task affinities using the linear approximation technique, varying the dimension of the gradient projection d from 200 to 1000. We find that d greater than 400 is sufficient to yield approximation errors of less than 5%. After obtaining affinity scores, we tune the cluster size and select the configuration with the highest average within-cluster affinity score, which can be done efficiently by running the clustering step. We discuss key parameters in deciding the branching structure in Section 4.3.4.

4.2 Approximation Results

First, we assess the accuracy of the first-order approximation used in our search algorithm. Recall that we apply the approximation to the model loss around an initialization $W^{(0)}$. We show that the error term induced by the approximation is small. Given the input X and an algorithm $A^{(i)}$, we compute the average residual sum of squares (RSS) across all output steps as follows:

$$\frac{1}{s^{(i)}(X)} \sum_{j=1}^{s^{(i)}(X)} \frac{\left\| f_W^{(i)}(X;j) - f_{W^{(0)}}^{(i)}(X;j) - \left[g_j^{(i)}\right]^\top \left(W - W^{(0)}\right) \right\|^2}{\left\| f_W^{(i)}(X;j) \right\|^2}.$$

We conduct experiments across various models, using twelve graph algorithmic reasoning tasks from both the CLRS benchmark [37] and its text-format version [29]. On graph-format datasets, we test four GNN architectures, including a standard MPNN [37], GAT [39], Triplet-GMPNN [14], and Edge Transformer [30]. On text-format datasets, we test four language models, including Qwen-3-1.7B, Llama-3-8B, Qwen-3-14B, and CodeLlama-34B. We use LoRA [13] as the base fine-tuning procedure.

To evaluate the RSS, we first obtain an initial model $W^{(0)}$ by training on all datasets. We then fine-tune the model on random task subsets to obtain W and evaluate the RSS under various

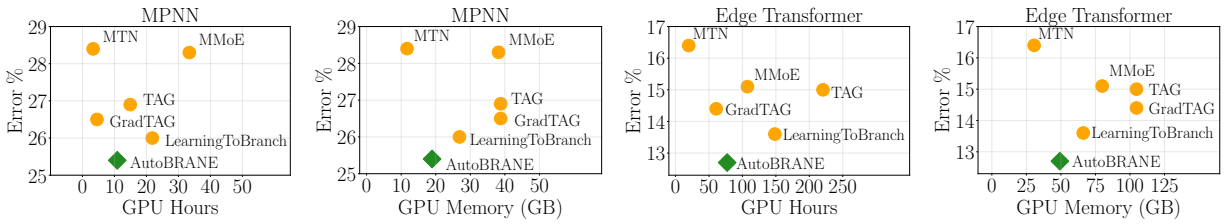

Figure 4: Illustration of the trade-off between error rate, GPU hours, and memory usage for AutoBRANE compared to existing multitask and branching network baselines. We present results obtained using MPNNs or edge transformers [30] as the base model. AutoBRANE outperforms a single multitask network by 3.7%, demonstrating the effectiveness of branching networks in leveraging positive task transfer. It also achieves the best overall trade-off, reducing the average error rate by 1.2% compared to the strongest baseline, while using 48% fewer GPU hours and 26% less memory.

relative distances $\frac{\|W-W^{(0)}\|}{\|W\|}$. For language models, W denotes the parameters of the LoRA adapters. Across 50 random task subsets, we find that the fine-tuned weights remain close to the initialization, typically within 10% relative distance.

The results are shown in Table 3. For all four GNN architectures, the first-order approximation yields under **1**% relative error when the distance from initialization is within 10%. On language models, the error remains below **5**%, with larger models generally yielding lower errors. We further observe that freezing early layers improves approximation accuracy. When the first three layers are frozen, the RSS drops below **0.06**%, shown in Appendix C.2. This supports the premise of our algorithm: avoiding repeated training on task subsets.

4.3 Comparison Results

4.3.1 Baselines

We compare AutoBRANE against multitask optimization baselines. Multitask network (MTN): This trains a single shared network on all tasks. Multi-gate Mixture-of-Experts model (MMoE) [28]: This uses task-specific gating networks to combine outputs from multiple expert models. Task Affinity Grouping (TAG) [8]: This clusters tasks based on pairwise affinities and trains a separate model for each group. LearningToBranch [11]: This is the most recent branching network that learns branching decisions using the Gumbel-Softmax technique. GradTAG [21]: This is the most recent multitask learning approach that estimates task affinities based on gradients and trains separate networks per group, without a branching structure. For the baselines, we select network size based on average validation performance across tasks.

4.3.2 Results on GNNs

Shown in Figure 4, AutoBRANE achieves the best trade-off between performance, runtime, and memory among all baselines. Compared with a single Edge Transformer, which previously achieved the state-of-the-art on CLRS, our method improves average accuracy by **3.7**%. Relative to the strongest multitask baselines, including LearningToBranch [11] and GradTAG [21], AutoBRANE yields a **1.2**% accuracy gain while reducing GPU hours by **48**% and memory usage by **26**%. We observe similar performance gains when using MPNN as the backbone. Our approach achieves **1.2**% accuracy gains over the best multitask baselines, while reducing GPU hours by **50**% and GPU memory usage by **29**%. We report the full results in Table 4.

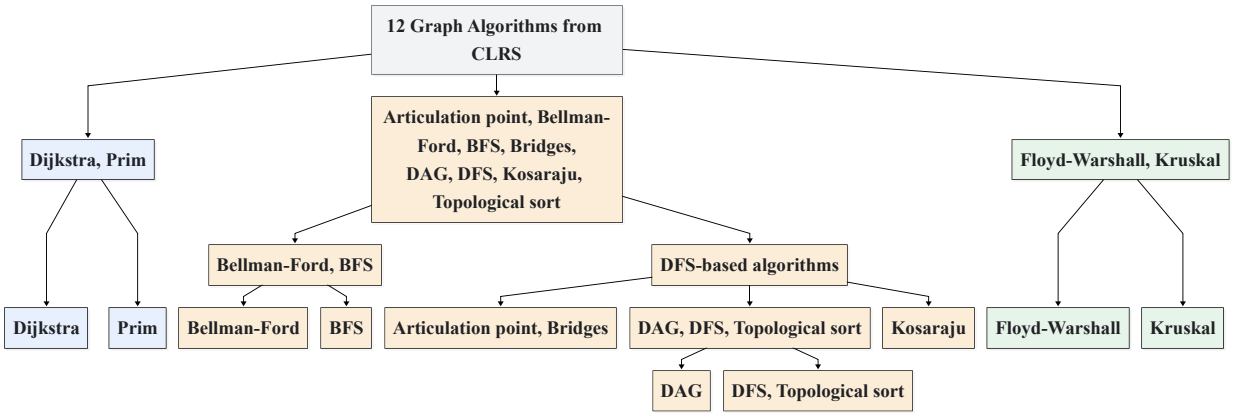

Figure 5: Illustration of the tree structure of the branching network learned on twelve algorithmic reasoning tasks from the CLRS benchmark. Our algorithm identifies clusters of algorithms that follow similar intermediate steps.

Figure 5 illustrates the branching network structure discovered by AutoBRANE when training Edge Transformers on the twelve graph-based algorithmic reasoning tasks. The resulting structure aligns well with task similarities in their intermediate steps, revealing three major clusters. The largest includes BFS, Bellman-Ford, and several DFS-based algorithms. Notably, BFS and Bellman-Ford are grouped together, consistent with observations from [40]. Five DFS-related tasks, including topological sort and DAG shortest paths, are clustered around DFS. Prim's and Dijkstra's algorithms form a group, reflecting their shared greedy edge-selection strategy. Kruskal's and Floyd-Warshall are grouped as well, both involving edge selection within components. A similar structure is observed using MPNN, which we omit for brevity.

4.3.3 Results on LLMs

We now present results on text-based graph reasoning tasks. AutoBRANE is compared against MTN, which fine-tunes a single LoRA adapter across all tasks, and GradTAG [21], the strongest multitask baseline. We exclude LearningToBranch [11], as it is not applicable to text-based datasets.

On CLRS-Text, AutoBRANE improves average test accuracy by **5.5**% relative to MTN and by **3.2**% over GradTAG. This highlights the advantage of the branching structure in capturing varying levels of task similarity. To demonstrate broader applicability, we also evaluate on the GraphQA and GraphWiz datasets and observe similar gains. On GraphQA, AutoBRANE surpasses MTN by **3.3**% and GradTAG by **2.2**% on average. On GraphWiz, it achieves gains of **3.8**% over MTN and **1.8**% over GradTAG. Complete results are provided in Table 5.

4.3.4 Results on Community Detection

We then apply AutoBRANE to a large multitask learning instance comprising 500 tasks on the community detection dataset. We compare AutoBRANE with MTN, LearningToBranch [11], and GradTAG [21]. To illustrate the relative improvement, we also compare our algorithm with a well-known community detection method, BigClam [45].

The results are shown in Table 6. We observe that our algorithm outperforms the most competitive MTL baselines by **28**%. Compared to LearningToBranch, our algorithm uses **4.5**× less GPU hours. Compared to GradTAG, our algorithm uses **28**% less GPU memory.

Table 4: Test score (%) evaluated on twelve graph algorithmic reasoning tasks. We report the average F_1-score over all tasks, GPU memory of the model, and runtime in terms of GPU hours, for each method. The test score is evaluated between model outputs and task labels at the final step. For each experiment, we run the experiment with three random seeds and report the average.

MPNN	STN	MTN	MMoE	TAG	LearningToBranch	GradTAG	AutoBRANE
BFS	100.0 ± 0.0	98.0 ± 0.4	89.4 ± 0.2	99.4 ± 0.5	93.2 ± 0.4	99.4 ± 0.4	99.6 ± 0.3
DFS	38.7 ± 1.5	29.7 ± 1.6	30.2 ± 2.7	36.7 ± 1.4	34.4 ± 3.2	25.1 ± 4.5	38.7 ± 1.5
Topo. sort	66.0 ± 2.6	71.3 ± 3.6	74.8 ± 2.4	72.8 ± 1.2	75.4 ± 2.1	61.4 ± 3.1	74.2 ± 1.7
Articulation points	99.8 ± 0.2	99.8 ± 0.1	97.1 ± 1.2	97.4 ± 1.3	95.5 ± 1.6	85.7 ± 0.8	93.0 ± 0.2
Bridges	82.6 ± 2.0	69.3 ± 0.4	91.1 ± 1.2	92.1 ± 1.9	92.2 ± 2.4	81.2 ± 2.5	96.2 ± 1.4
SCC Kosaraju	93.1 ± 3.1	92.4 ± 4.7	93.5 ± 2.6	89.0 ± 1.3	93.4 ± 2.6	90.1 ± 3.5	92.0 ± 1.8
MST Kruskal	69.9 ± 1.5	63.6 ± 2.0	66.2 ± 1.4	64.5 ± 0.8	68.0 ± 2.5	69.4 ± 1.1	66.4 ± 1.3
MST Prim	53.8 ± 1.3	54.3 ± 3.6	52.3 ± 1.4	50.1 ± 3.5	54.3 ± 2.7	95.3 ± 1.7	52.2 ± 2.4
Dijkstra	70.9 ± 2.3	69.3 ± 1.9	69.3 ± 2.4	67.3 ± 1.5	72.1 ± 1.4	78.3 ± 1.0	68.8 ± 1.7
Bellman-Ford	63.8 ± 4.1	59.2 ± 3.3	56.8 ± 1.6	61.9 ± 2.7	58.0 ± 3.4	85.9 ± 0.6	61.8 ± 2.4
DAG shortest paths	89.2 ± 1.4	90.0 ± 2.7	83.6 ± 1.6	89.8 ± 1.0	86.1 ± 2.4	88.6 ± 0.7	89.1 ± 1.5
Floyd-Warshall	69.4 ± 0.5	62.5 ± 0.7	56.1 ± 0.7	58.4 ± 1.7	58.1 ± 2.8	25.4 ± 1.4	63.1 ± 1.3
Avg. score	75.2	71.6	71.7	73.1	73.7	73.5	74.6
GPU hours	3.2	3.3	33.4	14.9	21.8	4.5	10.8
GPU memory	49.6	11.7	38.2	38.8	26.8	38.8	18.9
Edge Transformers	STN	MTN	MMoE	TAG	LearningToBranch	GradTAG	AutoBRANE
BFS	99.7 ± 0.4	98.6 ± 0.1	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0	99.7 ± 0.0	99.8 ± 0.0
DFS	65.6 ± 1.6	51.4 ± 1.5	39.9 ± 2.6	34.1 ± 1.8	36.7 ± 1.8	37.6 ± 3.7	42.6 ± 2.4
Topo. sort	98.7 ± 0.2	99.0 ± 0.9	97.7 ± 0.6	97.7 ± 0.7	98.7 ± 0.9	97.6 ± 2.3	96.2 ± 0.2
Articulation points	93.0 ± 3.6	96.4 ± 1.4	89.2 ± 2.3	89.2 ± 2.1	91.2 ± 1.4	99.0 ± 0.3	98.1 ± 1.0
Bridges	91.9 ± 0.3	92.2 ± 0.3	99.0 ± 0.0	99.2 ± 0.1	99.2 ± 0.2	93.5 ± 1.7	97.3 ± 0.2
SCC Kosaraju	65.8 ± 3.6	69.1 ± 2.9	64.2 ± 3.4	75.3 ± 3.4	76.7 ± 1.2	68.1 ± 2.8	63.9 ± 1.2
MST Kruskal	84.0 ± 1.4	82.2 ± 0.8	80.4 ± 1.8	80.4 ± 1.6	81.4 ± 1.8	79.5 ± 0.5	81.9 ± 0.3
MST Prim	93.0 ± 1.6	71.9 ± 1.4	85.5 ± 3.0	85.5 ± 1.8	87.1 ± 3.0	84.6 ± 1.0	92.8 ± 1.4
Dijkstra	91.9 ± 1.6	93.1 ± 3.8	94.0 ± 1.6	92.5 ± 1.1	93.7 ± 2.0	96.9 ± 0.4	98.7 ± 0.9
Bellman-Ford	89.9 ± 1.4	90.3 ± 2.3	92.7 ± 1.7	90.2 ± 1.4	92.4 ± 1.9	90.4 ± 0.6	96.7 ± 1.4
DAG shortest paths	97.6 ± 0.8	96.1 ± 0.2	98.3 ± 0.7	98.3 ± 0.3	99.3 ± 0.4	94.7 ± 0.5	99.0 ± 0.6
Floyd-Warshall	61.5 ± 3.4	76.9 ± 2.3	77.7 ± 1.4	77.7 ± 1.5	78.7 ± 2.7	82.2 ± 0.7	82.8 ± 1.8
Avg. score	86.0	83.6	84.9	85.0	86.4	85.4	87.3
GPU hours	30.4	19.6	107.5	220.5	148.5	60.8	77.2
GPU memory	126.1	30.6	79.9	104.7	66.2	104.7	49.2

4.3.5 Ablation Analysis

Next, we discuss the key parameters in building the branching network, including the number of layers L and the number of clusters k in the final layer. For GNNs, we tune L on a single multitask network from 3 to 6 and select the one with the best average validation performance. We set $L = 5$ for the CLRS benchmark and $L = 3$ for the community detection dataset. For language models, L is set as the depth of the pretrained language model, which is 28 for Qwen-3-1.7B and 16 for Llama-3-1B. We perform task partitioning every 4 layers.

For the cluster size, we determine the number of clusters at each layer based on the average affinity score within clusters. We control the growth of the cluster size (within a factor of 5) to prevent a dramatic expansion of the tree. On the CLRS benchmark, this leads to $k = 10$ clusters in the final layer. On the community detection dataset, there are $k = 22$ clusters in the final layer. We note that once task affinity scores are computed, clustering can be repeated within seconds to select the best configuration. We report all hyperparameters, along with base model runtime and memory usage, in Appendix C.

Table 5: Test accuracy (%) evaluated on text-based graph reasoning tasks from the CLRS-Text, GraphQA, and GraphWiz benchmarks. We compare our approach with fine-tuning a single adapter (MTN) and the most competitive MTL baseline, GradTAG [21]. We evaluate the average test accuracy between outputs and task labels at every intermediate step. We then compute the average accuracy over all tasks. We run each experiment with three random seeds and report the average.

CLRS-Text	MTN	GradTAG	AutoBRANE
BFS	79.3 ± 0.0	83.5 ± 0.4	89.6 ± 0.2
DFS	51.1 ± 0.7	55.7 ± 0.8	62.5 ± 1.9
Topological sort	19.0 ± 1.0	18.8 ± 0.1	18.2 ± 0.4
Articulation points	96.9 ± 0.1	97.4 ± 0.3	98.4 ± 0.2
Bridges	73.4 ± 0.0	73.3 ± 0.8	73.2 ± 0.7
SCC Kosaraju	93.6 ± 0.2	94.1 ± 0.5	95.2 ± 0.1
MST Kruskal	98.5 ± 0.2	98.6 ± 0.2	98.9 ± 0.9
MST Prim	62.1 ± 0.1	62.9 ± 0.3	64.2 ± 0.2
Dijkstra	63.6 ± 0.8	64.1 ± 0.7	65.0 ± 0.3
Bellman-Ford	74.8 ± 0.3	79.1 ± 0.8	85.0 ± 0.1
DAG	91.1 ± 0.9	89.1 ± 0.9	85.7 ± 0.2
Floyd-Warshall	29.3 ± 0.5	34.5 ± 0.9	43.0 ± 0.9
Avg. accuracy	69.4	70.9	73.3
GraphQA	MTN	GradTAG	AutoBRANE
Edge existence	96.7 ± 0.3	100.0 ± 0.0	99.6 ± 0.4
Node degree	98.1 ± 0.1	99.2 ± 0.3	99.7 ± 0.3
Node count	100.0 ± 0.0	100.0 ± 0.0	100.0 ± 0.0
Edge count	67.2 ± 2.2	68.0 ± 2.4	73.1 ± 1.7
Connected nodes	99.5 ± 0.2	99.9 ± 0.1	100.0 ± 0.0
Cycle check	99.4 ± 0.2	99.4 ± 0.1	99.8 ± 0.2
Disconnected nodes	81.2 ± 1.7	81.2 ± 0.6	94.6 ± 0.8
Reachability	98.0 ± 0.2	97.7 ± 0.2	98.2 ± 0.9
Shortest path	86.6 ± 0.1	91.0 ± 1.0	90.4 ± 0.4
Maximum flow	48.1 ± 0.3	46.9 ± 0.8	47.7 ± 0.9
Triangle counting	60.8 ± 0.2	62.1 ± 0.4	61.8 ± 0.9
Node classification	94.9 ± 3.5	94.9 ± 0.5	99.1 ± 0.3
Avg. accuracy	85.9	86.7	88.7
GraphWiz	MTN	GradTAG	AutoBRANE
Cycle Detection	45.6 ± 0.6	43.6 ± 0.5	43.8 ± 0.7
Connectivity	75.6 ± 0.7	76.2 ± 0.5	78.3 ± 0.8
Bipartite Graph	62.7 ± 0.6	64.7 ± 0.7	65.6 ± 0.5
Topological Sort	13.7 ± 0.3	20.1 ± 0.3	22.1 ± 0.3
Shortest Path	24.7 ± 0.9	25.2 ± 0.2	21.6 ± 0.8
Maximum Triangle Sum	29.3 ± 0.7	29.3 ± 0.5	29.6 ± 0.6
Maximum Flow	10.2 ± 0.7	13.2 ± 0.2	15.7 ± 0.7
Hamilton Path	48.5 ± 0.4	46.4 ± 0.5	47.8 ± 0.3
Subgraph Matching	84.0 ± 0.9	83.1 ± 0.2	85.2 ± 0.8
Avg. accuracy	43.8	44.7	45.5

Furthermore, we validate the branching structure by randomly varying the positions of tasks across branches. Compared to the tree in Figure 5, we find that swapping positions between branches, such as between Kruskal's and BFS or DFS and Bellman-Ford, leads to more than 2% test F_1-score drops.

Table 6: We report the test macro-averaged F_1-score over all community labels, the number of GPU hours, and GPU memory usage (in GB) of our algorithm, evaluated on the Orkut community detection dataset with 500 community node labeling tasks.

	Macro F_1-score	GPU hours	Memory
BigCLAM [45]	22.69 ± 0.25	**11.2**	39.1
MTN [43]	27.24 ± 1.66	37.8	**5.7**
LearningToBranch [11]	34.53 ± 3.50	366.6	91.2
GradTAG [21]	38.77 ± 4.63	68.0	125.4
AutoBRANE (Ours)	**49.69 ± 2.97**	81.4	89.5

4.4 Measuring Sample Complexities

Finally, we evaluate the empirical sample complexity for learning three graph algorithms: BFS, Dijkstra's algorithm, and Prim's algorithm. In particular, we set the target error rate to 1% and increase the number of samples until the trained GNN's validation error falls below 1%. We evaluate the error between model predictions and task labels at the last step of the algorithm. The results are shown in Figure 6.

We find that the number of samples needed for learning the three algorithms differs, with Prim being the most difficult to learn. We also compare training with and without the loss evaluated on the node labels of intermediate steps. We found that training with intermediate steps reduces the number of samples by approximately a factor of two, on both Dijkstra's and Prim's algorithms. Furthermore, we found that standard multitask training (MT) on Dijkstra and Prim tasks with a single GNN network increases the sample complexity of single-task training. It remains an interesting direction for future work to explain this observation. Specifically, what determines the learnability of an algorithm? One hypothesis is that the higher sample complexity of learning the Prim algorithm comes from the (non-local) nature of its executions.

5 Related Work

Recent work has examined whether neural networks can be trained to imitate classical algorithms step-by-step. For graph algorithms, Veličković et al. [40] showed that message-passing neural networks with maximization aggregation closely align with the internal computations of algorithms such as reachability and shortest paths. Their results further indicate that parameter sharing across layers yields positive transfer between related algorithms. To systematically investigate this challenging problem, the CLRS benchmark was introduced [37], providing a unified evaluation suite covering thirty algorithms from the standard textbook by Cormen et al. [5].

Several recent works have proposed architectures tailored for algorithmic reasoning. Ibarz et al. [14] introduce a triplet-GMPNN model that assigns values to edges and updates each edge based on the values of its adjacent edges. When applied to tasks such as reachability and shortest paths, the network follows similar execution steps [14]. More recently, augmenting transformers with tri-attention over node pairs, relative to other-conditioning, has been shown to substantially improve prediction accuracy on algorithmic graph tasks [30]. Branching architectures have also been studied in computer vision [25, 11]. Branched multitask networks [36] seek a tree-structured architecture that minimizes the sum of task-affinity scores across layers.

Another relevant direction is the design of mixture-of-experts models for multitask learning. Such models comprise multiple expert subnetworks together with a gating mechanism that produces

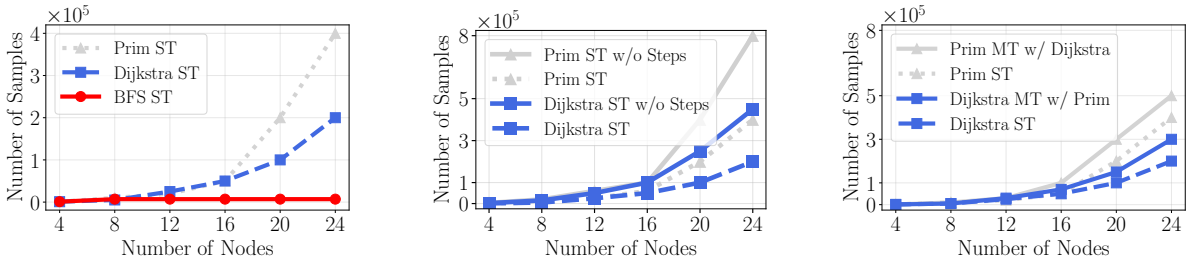

Figure 6: We illustrate the sample complexity scaling for learning three graph algorithms on graphs with different numbers of nodes. We find that the sample complexity for single-task training (ST) on the three algorithms differs when they are trained separately, with Prim requiring the most training samples. Second, training without the intermediate steps (w/o intermediate Steps) yields higher sample complexity compared to training with intermediate steps. Further, with multitask training (MT), the number of samples increases compared with single-task training. In this experiment, we conduct multitask training on the combined dataset of Dijkstra and Prim tasks.

weighted combinations of expert outputs. Examples include task-specific gating functions [28] and differentiable sparse gating [12]. A complementary line of work studies multitask learning from a regularization perspective [43, 44]. Recent analyses investigate how implicit and explicit regularization influence task interference and representation sharing [20, 23], and apply gradient-based task features to supervised fine-tuning [23] and in-context learning [50], highlighting the role of gradient geometry in understanding task-relatedness. Our work establishes a first connection between this body of literature and the problem of multitask algorithmic reasoning.

We now discuss recent studies evaluating the reasoning capabilities of LLMs on graph problems. GraphQA [6] systematically evaluates prompting strategies for encoding graphs (e.g., adjacency or incidence matrices) across twelve algorithms and retrieval tasks, finding that even large models such as PaLM-62B can underperform simple majority-vote baselines. NLGraph [41] introduces eight graph reasoning tasks and shows that GPT-3.5 succeeds on problems like graph connectivity and shortest paths but struggles with more complex computations, such as Hamiltonian paths. GraphWiz [4] applies instruction tuning with explicit reasoning traces, while GraphInstruct [27] develops instruction-tuned datasets over twenty-one graph reasoning tasks with detailed intermediate steps. It is worth clarifying that existing work focuses on *predicting the final output of the algorithmic task*. By contrast, we focus on *models capable of generating the intermediate steps*. Our fine-tuning experiments with Llama on GraphQA versus CLRS-Text further indicate that producing intermediate states is substantially more challenging—a gap that warrants further exploration. CLRS-Text [29] reformulates CLRS tasks as natural-language descriptions, showing that transformers can be trained to mimic algorithm execution, though they typically require significantly more training data than GNNs. Bounsi et al. [3] introduce a multimodal architecture that jointly pretrains transformers and GNNs using cross-attention to integrate their representations. GraphArena [33] proposes broader benchmarks evaluating feasibility, hallucination, and other dimensions of graph-reasoning behavior.

Various techniques have been proposed to enhance LLM performance on graph-related problems. One class of approaches develops multimodal architectures that integrate GNNs with LLMs. GreaseLM [49] augments LLMs with knowledge-graph signals through a fusion layer that combines textual and graph-based representations. GNP [34] incorporates a GNN module to encode knowledge graphs as embeddings within LLM prompts. Other lines of work explore code-generation pipelines (e.g., fine-tuning code LMs in graph coders), specialized tool-calling instructions (GraphTool-Instruction), and LLM-based multi-agent frameworks. Our work is complementary to these works

and focuses on multitask algorithmic reasoning, with an emphasis on predicting not only final answers but also intermediate algorithmic states. An interesting direction is to provide better explanations of GNNs for algorithmic reasoning, for instance by measuring the topology of complex predictions [24], as well as to study broader algorithmic tasks including randomized computation [35, 1, 18, 48].

Besides algorithmic reasoning, latent multi-hop reasoning has also been studied to evaluate large language models in factual information retrieval [46]. This line of work examines how models internally retrieve and utilize intermediate factual knowledge stored in their parameters to answer a question when such information is not explicitly provided in the prompt. Prior work [2] identifies a mechanism for latent two-hop reasoning in which the first hop is resolved in earlier layers through identifying the intermediate answer, which then propagates to later layers to resolve the second hop. An interpretability method has been proposed to analyze failures in latent multi-hop reasoning by tracing how logits propagate across layers and positions [47]. This analysis shows that errors can arise from conflicts among entity logits extracted in higher layers. In contrast, our work focuses on algorithmic reasoning, where models explicitly generate intermediate steps. It is intriguing to study whether branching neural networks can capture reasoning over multiple factual knowledge.

6 Conclusion

This work introduces a branching network capable of simultaneously solving multiple algorithmic reasoning tasks. We design an algorithm that automatically learns an optimal branching structure to capture the varying similarities between tasks. This algorithm efficiently finds the structure, enabled by a fast sub-procedure that determines the task partition at every layer. Across various algorithmic reasoning tasks and model architectures, our method outperforms existing multitask and branching networks, while achieving an optimal trade-off between runtime and model memory usage. Overall, these findings highlight the effectiveness of branching architectures for multitask algorithmic reasoning.

Acknowledgments

Thanks to Jonathan Ullman, David Gleich, Stratis Ioannidis, and Virgil Pavlu for several discussions about this work. Thanks to the anonymous referees for their feedback. The work of D. Li, Z. Zhang, M. Duan, and H. R. Zhang is partially funded by NSF award IIS-2412008. D. Li is also partially funded by a PhD fellowship from JPMorgan Chase & Co.

References

[1] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng. "Local computation of pagerank contributions". In: *International Workshop on Algorithms and Models for the Web-Graph*. Springer. 2007, pp. 150–165 (19).

[2] E. Biran, D. Gottesman, S. Yang, M. Geva, and A. Globerson. "Hopping too late: Exploring the limitations of large language models on multi-hop queries". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2024, pp. 14113–14130 (19).

[3] W. Bounsi, B. Ibarz, A. Dudzik, J. B. Hamrick, L. Markeeva, A. Vitvitskyi, R. Pascanu, and P. Veličković. "Transformers meet Neural Algorithmic Reasoners". In: *arXiv preprint arXiv:2406.09308* (2024) (18).

[4] N. Chen, Y. Li, J. Tang, and J. Li. "GraphWiz: An Instruction-Following Language Model for Graph Problems". In: *KDD* (2024) (3, 11, 18, 28, 29).

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022 (17).

[6] B. Fatemi, J. Halcrow, and B. Perozzi. "Talk like a Graph: Encoding Graphs for Large Language Models". In: *ICLR*. 2023 (3, 10, 18, 27–29).

[7] E. A. Feigenbaum and J. Feldman. "Computers and thought." In: (1963) (1).

[8] C. Fifty, E. Amid, Z. Zhao, T. Yu, R. Anil, and C. Finn. "Efficiently identifying task groupings for multi-task learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 27503–27516 (8, 9, 13).

[9] F. Frasca, E. Rossi, D. Eynard, B. Chamberlain, M. Bronstein, and F. Monti. "Sign: Scalable inception graph neural networks". In: *arXiv preprint arXiv:2004.11198* (2020) (11).

[10] D. G. Georgiev, P. Lio, J. Bachurski, J. Chen, T. Shi, and L. Giusti. "Beyond erdos-renyi: Generalization in algorithmic reasoning on graphs". In: *The Second Learning on Graphs Conference*. 2023 (4).

[11] P. Guo, C.-Y. Lee, and D. Ulbricht. "Learning to branch for multi-task learning". In: *ICML*. PMLR. 2020 (3, 8–10, 13, 14, 17, 26).

[12] H. Hazimeh, Z. Zhao, A. Chowdhery, M. Sathiamoorthy, Y. Chen, R. Mazumder, L. Hong, and E. Chi. "Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning". In: *NeurIPS* (2021) (9, 18).

[13] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. "Lora: Low-rank adaptation of large language models". In: *ICLR* (2021) (5, 11, 12, 27, 30).

[14] B. Ibarz, V. Kurin, G. Papamakarios, K. Nikiforou, M. Bennani, R. Csordás, A. J. Dudzik, M. Bošnjak, A. Vitvitskyi, Y. Rubanova, A. Deac, B. Bevilacqua, Y. Ganin, C. Blundell, and P. Veličković. "A generalist neural algorithmic learner". In: *Learning on graphs conference*. 2022 (2, 3, 12, 17).

[15] A. Ilyas, S. M. Park, L. Engstrom, G. Leclerc, and A. Madry. "Datamodels: Predicting predictions from training data". In: *International conference on machine learning* (2022) (3).

[16] A. Jacot, F. Gabriel, and C. Hongler. "Neural tangent kernel: Convergence and generalization in neural networks". In: *Advances in neural information processing systems* 31 (2018) (3, 7).

[17] W. B. Johnson. "Extensions of Lipshitz mapping into Hilbert space". In: *Conference modern analysis and probability, 1984*. 1984, pp. 189–206 (10, 24).

[18] K. Kloster and D. F. Gleich. "Heat kernel based community detection". In: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. 2014, pp. 1386–1395 (19).

[19] D. Li, H. Ju, A. Sharma, and H. R. Zhang. "Boosting multitask learning on graphs through higher-order task affinities". In: *KDD*. 2023 (6, 9).

[20] D. Li, H. Nguyen, and H. R. Zhang. "Identification of Negative Transfers in Multitask Learning Using Surrogate Models". In: *Transactions on Machine Learning Research* (2023) (6, 18).

[21] D. Li, A. Sharma, and H. R. Zhang. "Scalable Multitask Learning Using Gradient-based Estimation of Task Affinity". In: *KDD*. 2024 (6, 7, 10, 13, 14, 16, 17).

[22] D. Li, Z. Zhang, L. Wang, and H. Zhang. "Scalable Fine-tuning from Multiple Data Sources: A First-Order Approximation Approach". In: *Findings of the Association for Computational Linguistics: EMNLP 2024*. 2024, pp. 5608–5623 (3).

[23] D. Li, Z. Zhang, L. Wang, and H. R. Zhang. "Efficient Ensemble for Fine-tuning Language Models on Multiple Datasets". In: *ACL* (2025) (18).

[24] M. Liu, T. K. Dey, and D. F. Gleich. "Topological structure of complex predictions". In: *Nature Machine Intelligence* 5.12 (2023), pp. 1382–1389 (19).

[25] Y. Lu, A. Kumar, S. Zhai, Y. Cheng, T. Javidi, and R. Feris. "Fully-adaptive feature sharing in multi-task networks with applications in person attribute classification". In: *CVPR*. 2017 (17, 26).

[26] A. B. de Luca and K. Fountoulakis. "Simulation of Graph Algorithms with Looped Transformers". In: *International Conference on Machine Learning*. 2024 (4).

[27] Z. Luo, X. Song, H. Huang, J. Lian, C. Zhang, J. Jiang, and X. Xie. "Graphinstruct: Empowering large language models with graph understanding and reasoning capability". In: *arXiv preprint arXiv:2403.04483* (2024) (18).

[28] J. Ma, Z. Zhao, X. Yi, J. Chen, L. Hong, and E. H. Chi. "Modeling task relationships in multi-task learning with multi-gate mixture-of-experts". In: *KDD*. 2018 (9, 13, 18).

[29] L. Markeeva, S. McLeish, B. Ibarz, W. Bounsi, O. Kozlova, A. Vitvitskyi, C. Blundell, T. Goldstein, A. Schwarzschild, and P. Veličković. "The CLRS-Text Algorithmic Reasoning Language Benchmark". In: *arXiv preprint arXiv:2406.04229* (2024) (3, 10, 12, 18, 26, 27, 30).

[30] L. Müller, D. Kusuma, B. Bonet, and C. Morris. "Towards Principled Graph Transformers". In: *NeurIPS* (2024) (2, 3, 6, 10, 12, 13, 17, 30).

[31] S. M. Park, K. Georgiev, A. Ilyas, G. Leclerc, and A. Madry. "Trak: Attributing model behavior at scale". In: *ICML* (2023) (3).

[32] S. Russell and P. Norvig. "Artificial Intelligence: A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25.27 (1995), pp. 79–80 (1).

[33] J. Tang, Q. Zhang, Y. Li, N. Chen, and J. Li. "Grapharena: Evaluating and exploring large language models on graph computation". In: *International Conference on Learning Representations* (2024) (18).

[34] Y. Tian, H. Song, Z. Wang, H. Wang, Z. Hu, F. Wang, N. V. Chawla, and P. Xu. "Graph neural prompting with large language models". In: *AAAI*. 2024 (18).

[35] H. Tong, C. Faloutsos, and J.-Y. Pan. "Fast random walk with restart and its applications". In: *Sixth international conference on data mining (ICDM'06)*. IEEE. 2006, pp. 613–622 (19).

[36] S. Vandenhende, S. Georgoulis, B. De Brabandere, and L. Van Gool. "Branched multi-task networks: Deciding what layers to share". In: *BMVC* (2020) (17, 26).

[37] P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, and C. Blundell. "The CLRS algorithmic reasoning benchmark". In: *ICML*. 2022 (2–4, 10, 12, 17, 28, 30).

[38] P. Veličković and C. Blundell. "Neural algorithmic reasoning". In: *Patterns* 2.7 (2021) (2, 4).

[39] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. "Graph Attention Networks". In: *International Conference on Learning Representations*. 2018 (10, 12).

[40] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell. "Neural execution of graph algorithms". In: *ICLR* (2020) (14, 17).

[41] H. Wang, S. Feng, T. He, Z. Tan, X. Han, and Y. Tsvetkov. "Can Language Models Solve Graph Problems in Natural Language?" In: *NeurIPS* (2023) (2, 18).

[42] J. J. Whang, D. F. Gleich, and I. S. Dhillon. "Overlapping community detection using neighborhood-inflated seed expansion". In: *IEEE Transactions on Knowledge and Data Engineering* 28.5 (2016), pp. 1272–1284 (3).

[43] S. Wu, H. R. Zhang, and C. Ré. "Understanding and improving information transfer in multi-task learning". In: *ICLR* (2020) (17, 18).

[44] F. Yang, H. R. Zhang, S. Wu, C. Ré, and W. J. Su. "Precise high-dimensional asymptotics for quantifying heterogeneous transfers". In: *Journal of Machine Learning Research* 26.113 (2025), pp. 1–88 (18).

[45] J. Yang and J. Leskovec. "Overlapping community detection at scale: a nonnegative matrix factorization approach". In: *WSDM*. 2013 (11, 14, 17).

[46] S. Yang, E. Gribovskaya, N. Kassner, M. Geva, and S. Riedel. "Do large language models latently perform multi-hop reasoning?" In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2024, pp. 10210–10229 (19).

[47] Z. Yu, Y. Belinkov, and S. Ananiadou. "Back attention: Understanding and enhancing multi-hop reasoning in large language models". In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2025, pp. 11268–11283 (19).

[48] H. Zhang, P. Lofgren, and A. Goel. "Approximate personalized pagerank on dynamic graphs". In: *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016, pp. 1315–1324 (19).

[49] X. Zhang, A. Bosselut, M. Yasunaga, H. Ren, P. Liang, C. D. Manning, and J. Leskovec. "Greaselm: Graph reasoning enhanced language models for question answering". In: *ICLR* (2022) (18).

[50] Z. Zhang, Z. Zhang, D. Li, L. Wang, J. Dy, and H. R. Zhang. "Linear-Time Demonstration Selection for In-Context Learning via Gradient Estimation". In: *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 2025, pp. 16470–16488 (18).

A Proof of Proposition 3.3

We provide a proof for the Proposition 3.3 using the logistic loss in binary classification. We note that the extension to multiclass classification loss is straightforward using the same technique, which requires additional notations.

Proof of Proposition 3.3. Recall that the estimated weight $\hat{W}_S$ is obtained from the minimizer of the logistic regression using the projected gradients as the features. To make it clear, we annotate the vector with its dimension so that it is easy to distinguish. Let $\hat{W}_d$ denote the minimizer of the logistic regression in dimension d. We have $\hat{W}_S = P\hat{W}_d + W^{(0)}$ given a p by d random projection matrix P and the weight initialization $W^{(0)}$.

Specifically, $\hat{W}_d$ is the minimizer of the following problem:

$$h_1(W) = \frac{1}{|T|\,|S|} \sum_{X \in T} \sum_{i \in S} \frac{1}{S^{(i)}(X)} \sum_{j=1}^{S^{(i)}(X)} \log\Big(1 + \exp\Big(- A_j^{(i)}(X) \big[g_j^{(i)}\big]^\top PW + b_j^{(i)}\Big)\Big),$$

for $W \in \mathbb{R}^d$, where $g_j^{(i)} = \nabla f_{W^{(0)}}^{(i)}(X; j), b_j^{(i)} = -A_j^{(i)}(X) f_{W^{(0)}}^{(i)}(X; j)$.

To relate this loss with the training loss $\hat{L}_S(W)$, we define an intermediate solution $\bar{W}_p$ in dimension p for the following problem:

$$h_2(W) = \frac{1}{|T|\,|S|} \sum_{X \in T} \sum_{i \in S} \frac{1}{S^{(i)}(X)} \sum_{j=1}^{S^{(i)}(X)} \log\Big(1 + \exp\Big(- A_j^{(i)}(X) \big[g_j^{(i)}\big]^\top PP^\top (W - W^{(0)})) + b_j^{(i)}\Big)\Big).$$

We can know that $h_1(\hat{W}_d) \le h_2(\bar{W}_p)$, since the second problem is a specific case of the first one. Denote $W^\star$ as the minimizer for the training loss for $W \in \mathbb{R}^p$:

$$\min \hat{L}_S(W) = \frac{1}{|T|\,|S|} \sum_{X \in T} \sum_{i \in S} \frac{1}{S^{(i)}(X)} \sum_{j=1}^{S^{(i)}(X)} \log\Big(1 + \exp\Big(- A_j^{(i)}(X) f_W^{(i)}(X; j)\Big)\Big).$$

We will complete the proof by showing that the following two bounds hold. First, we will show that the error between $h_1(\hat{W}_d)$ and $\hat{L}_S(W^\star)$ is bounded:

$$h_1(\hat{W}_d) \le h_2(\bar{W}_p) \le h_2(W^\star) \le \hat{L}_S(W^\star) + \delta + 2GD\epsilon. \tag{5}$$

Second, we will show that the error between $h_1(\hat{W}_d)$ and $\hat{L}_S(P\hat{W}_d + W^{(0)})$ is also bounded:

$$\Big|h_1(\hat{W}_d) - \hat{L}_S(P\hat{W}_d + W^{(0)})\Big| \le \delta. \tag{6}$$

Next, we prove the first bound in Equation (5). We have known that $h_1(\hat{W}_d) \le h_2(\bar{W}_p)$ and $h_2(\bar{W}_p) \le h_2(W^\star)$, as $\bar{W}_p$ is minimizer of $\min h_2(W)$. We will bound the error between $h_2(W^\star)$ and $\hat{L}_S(W^\star)$. Let's expand the training loss at $W^\star$. To simplify the notations, we write down one logistic loss:

$$\begin{aligned}
&\log\Big(1 + \exp\Big(- A_j^{(i)}(X) f_{W^\star}^{(i)}(X; j)\Big)\Big) \\
=&\log\Big(1 + \exp\Big(- A_j^{(i)}(X) \big[g_j^{(i)}\big]^\top (W^\star - W^{(0)}) + b_j^{(i)} + \bar{\epsilon}_j^{(i)}\Big)\Big) \\
=&\log\Big(1 + \exp\Big(- A_j^{(i)}(X) \big[g_j^{(i)}\big]^\top PP^\top (W^\star - W^{(0)}) + b_j^{(i)} + \bar{\epsilon}_j^{(i)} + \tilde{\epsilon}_j^{(i)}\Big)\Big),
\end{aligned}$$

where $\bar{\epsilon}_j^{(i)} = -A_j^{(i)}(X)\epsilon_{X,j}^{(i)}$ involves the Taylor's expansion error and

$$\tilde{\epsilon}_j^{(i)} = \left[g_j^{(i)}\right]^\top (\mathrm{Id}_d - PP^\top)(W^\star - W^{(0)}).$$

Then, we leverage the fact that the logistic loss is 1-Lipschitz continuous. Based on the mean value theorem, we can have that:

$$\log(1+\exp(-x+\epsilon)) \le \log(1+\exp(-x)) + |\epsilon|\,.$$

Therefore, we can bound the error between $h_2(W^\star)$ and $\hat{L}_S(W^\star)$ as follows:

$$h_2(W^\star) \le \hat{L}_S(W^\star) + \frac{1}{|T|\,|S|}\sum_{X\in T}\sum_{i\in S}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\left(\left|\bar{\epsilon}_j^{(i)}\right| + \left|\tilde{\epsilon}_j^{(i)}\right|\right).$$

From the assumption that the averaged Taylor's expansion error is at most δ, we have:

$$\frac{1}{|T|\,|S|}\sum_{X\in T}\sum_{i\in S}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\left|\bar{\epsilon}_j^{(i)}\right| \le \delta.$$

We then bound the second error term by the Johnson-Lindenstrauss Lemma [17]. Provided that $d = O\big(\frac{\log p}{\epsilon^2}\big)$, we have

$$\left|\left\langle g_i, W^\star - W^{(0)}\right\rangle - \left\langle Pg_i, P(W^\star - W^{(0)})\right\rangle\right| \le \epsilon\left|\left\langle g_i, W^\star - W^{(0)}\right\rangle\right| \le 2GD\epsilon.$$

Therefore, we have proved the first bound in equation (5):

$$h_2(W^\star) \le \hat{L}_S(W^\star) + \frac{1}{|T|\,|S|}\sum_{X\in T}\sum_{i\in S}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\left(\left|\bar{\epsilon}_j^{(i)}\right| + \left|\tilde{\epsilon}_j^{(i)}\right|\right) \le \delta + 2GD\epsilon.$$

Next, we prove the second bound in Equation (6) using the same idea. This is based on observing the training loss on $P\hat{W}_d + W^{(0)}$:

$$\begin{aligned}&\log\Big(1+\exp\Big(-A_j^{(i)}(X) f^{(i)}_{P\hat{W}_d+W^{(0)}}(X;j)\Big)\Big)\\ =&\log\Big(1+\exp\Big(-A_j^{(i)}(X)\left[g_j^{(i)}\right]^\top P\hat{W}_d + b_j^{(i)} + \bar{\epsilon}_j^{(i)}\Big)\Big),\end{aligned}$$

where $\bar{\epsilon}_j^{(i)}$ involves the Taylor expansion error.

Using the 1-Lipschitz continuous property of the logistic loss again, we can have that

$$\left|h_1(\hat{W}_d) - \hat{L}_S(P\hat{W}_d + W^{(0)})\right| \le \frac{1}{|T|\,|S|}\sum_{X\in T}\sum_{i\in S}\frac{1}{S^{(i)}(X)}\sum_{j=1}^{S^{(i)}(X)}\left|\bar{\epsilon}_j^{(i)}\right| \le \delta.$$

We have now completed the proof.

B Detailed Approach

We list the notations we use in the paper as follows.

Table 7: A list of mathematical notations used in the paper.

Notation	Meaning
$A^{(i)}$	The i-th algorithm
X	Input, including the graph structure
$A_j^{(i)}$	j-th intermediate step of the i-th algorithm
$S^{(i)}(X)$	Total number of steps of applying the i-th algorithm on input X
S	Subset of tasks
f_W	A neural network with parameters W
$f_W^{(i)}(X;j)$	The prediction of the step j of the i-th algorithm
$g_j^{(i)}$	The gradient of $f_W^{(i)}(X;j)$ with respect to the network parameter W
$T_{j,l}^{(i)}$	Layer-wise task affinity between task i and j at layer l

B.1 Clustering Algorithms

We provide further details of our fast approximation algorithm, which uses a first-order expansion of the network output and logistic regression on gradient-based features, combined with a dimension reduction step. In the following discussion, we focus on binary classification, such that $A_j^{(i)}(X) \in \{+1, -1\}$. Recall the gradient-based approximation of $f_W^{(i)}(X, j)$, given the input X and intermediate step j:

$$f_W^{(i)}(X,j) \approx f_{W^{(0)}}^{(i)}(X,j) + \big[\nabla_{W_{l:L}} f_{W^{(0)}}^{(i)}(X,j)\big]^\top \big(W_{l:L} - W_{l:L}^{(0)}\big) + \epsilon_{X,j}^{(i)}. \tag{7}$$

Let us denote $\nabla_{W_{l:L}} f_{W^{(0)}}^{(i)}(X,j)$ as $g_j^{(i)}$ and $-A_j^{(i)}(X) f_{W^{(0)}}^{(i)}(X,j)$ as $b_j^{(i)}$, for any intermediate step j. Using logistic loss, the approximate loss term for each sample is

$$\tilde{\ell}(f_W^{(i)}(X)) = \log\Big(1 + \exp\big(-A_j^{(i)}(X) g_j^{(i)\top}\big(W_{l:L} - W_{l:L}^{(0)}\big) + b_j^{(i)}\big)\Big), \tag{8}$$

for $W \in \mathbb{R}^p$. Denote the combined data set in the task subset S as $\mathcal{D}_S$, where n_S is the combined number of data samples in the set $\mathcal{D}_S$. The main idea is to solve a logistic regression problem with $g_j^{(i)}$ being the feature vector and $A_j^{(i)}(X)$ being the response label.

Since $g_j^{(i)}$ can be high-dimensional (on the order of the neural network's parameter count), we apply a Johnson-Lindenstrauss (JL) random projection to reduce dimension without losing much precision. Specifically, let $P \in \mathbb{R}^{p\times d}$ be a Gaussian random matrix, where each entry is drawn i.i.d. from $\mathcal{N}(0, d^{-1})$. We project the gradient from dimension p onto dimension d as $\tilde{g}_j^{(i)} = P^\top g_j^{(i)}$. Then, we solve the following d-dimensional logistic regression, which is now in dimension d, which aims to minimize the approximated logistic loss on all training samples for tasks in S:

$$\hat{W}_d \leftarrow \underset{W\in\mathbb{R}^d}{\arg\min} \frac{1}{n_S} \sum_{X\in\mathcal{D}_S} \sum_{i\in S} \frac{1}{S^{(i)}(X)} \sum_{j=1}^{S^{(i)}(X)} \log\Big(1 + \exp\Big(-A_j^{(i)}(X)\big[g_j^{(i)}\big]^\top PW + b_j^{(i)}\Big)\Big).$$

Finally, we map the learned $\hat{W}_d \in \mathbb{R}^d$ back to the full p-dimensional space: $\hat{W}_S = P\hat{W}_d + W^{(0)}$. The $\hat{W}_S$ then is the estimation of the fine-tuned weights on a subset S.

Task Affinity based Clustering. Let T denote the task affinity matrix at a layer l. The goal is to cluster n tasks into k groups such that tasks with higher affinity are grouped together. Given the number of clusters k, let $v_1, \ldots, v_k$ be binary indicator vectors indicating whether a task is in each cluster. The average density is computed as $\sum_{i=1}^{k}(v_i^\top T v_i / v_i^\top v_i)$. Specifically, define a new variable as $X = \sum_{i=1}^{k}(v_i v_i^\top / v_i^\top v_i)$. As $v_i v_i^\top$ is a rank-one semi-definite matrix, the rank and trace of X are equal to k. This can be formulated as a rank-constrained maximization problem:

$$\begin{aligned} \max_{X \in \mathbb{R}^{n \times n}} \quad & \langle T, X \rangle \\ & Xe = e, \mathrm{Tr}[X] = k, \mathrm{rank}(X) = k \\ & X \geq 0, X \succeq 0. \end{aligned}$$

The integral objective is NP-hard in general. Therefore, we relax it using semi-definite programming (SDP), followed by a rounding step to obtain discrete clusters.

Since splitting tasks into k clusters at layer l increases the network size by $(L-l)k$, we apply the SDP relaxation with a regularization term on the number of clusters k to control the network size. The regularized SDP objective becomes:

$$\begin{aligned} \max_{X \in \mathbb{R}^{n \times n}} \quad & \langle T, X \rangle - \lambda(L-l)\,\mathrm{Tr}[X] \\ & Xe = e, X \geq 0, X \succeq 0 \end{aligned} \tag{9}$$

where λ is a hyperparameter that controls how much we penalize the increase in network size. Once we obtain the SDP solution $\hat{X}$, we round $\hat{X}$ into an integer solution using a threshold of $1/n$. Solving the SDP for an $n \times n$ matrix is computationally efficient, taking less than 5 seconds, making it negligible compared to the overall model training time.

Discussion. Let the running time for training a single L-layer network on one task be T, with memory usage B. Our algorithm discovers the branching network in nLT time and uses kB/L memory. By contrast, training a mixture of expert networks with k networks takes knT time and kB memory. Task grouping based on fully computed task affinity takes n^2T time and kB memory. LearningToBranch [11] that searches for branching decisions takes $k^L nT$ time and uses nB memory since it trains with n modules per layer. One existing method [25] takes a layer-by-layer search approach from the last layer to the first. At each layer, their method determines the number of groups for the branching by optimizing a criterion over task-relatedness and model complexity. In contrast, LearningToBranch [11] parameterizes the branching decisions at each layer and optimizes the variables in an end-to-end training pipeline. Vandenhende et al. [36] designs a method that exhaustively searches over all possible trees of L layers to find the tree structure that has the minimum sum of task affinity scores over all layers. These methods use a search space of $O(k^{nL})$. Other strategies have also been explored for tree construction. For example, one can still proceed from the last layer toward the first layer. Our algorithm can be applied generically to these different branching strategies, providing flexibility when building a branched multi-task network.

B.2 Using Node Embeddings for Graph Algorithmic Reasoning

Next, we use the embeddings learned from Algorithm 2 to deal with text descriptions of a graph reasoning task. To motivate our approach, we first evaluate whether existing open-source language models can solve the tasks accurately. We use four CLRS-Text tasks, including BFS, Bellman-Ford, Dijkstra, and Prim's algorithm [29]. We use Llama-3-8B as the base language model. The input

Algorithm 3 Fine-tuning LLMs with AutoBRANE Embeddings

Input: Training and validation datasets of n algorithmic reasoning tasks in text formats, Pretrained language model LM($\cdot$)

Require: A text instruction $P_{\text{instruction}}$, Number m of subsets and their size α, Projection dimension d, Regularization parameter λ

Output: A fine-tuned language model with an AutoBRANE

1: Convert the text input descriptions into graphs for every task
2: Train an L-layer AutoBRANE on the n tasks with graph inputs, using Algorithm 2
3: For each training sample, concatenate the node embeddings of AutoBRANE with the text embeddings of $P_{\text{instruction}}$
4: Fine-tune the LM along with the AutoBRANE, using the concatenated embeddings as input, and minimizing the training loss averaged over intermediate steps.

involves a text description of a graph instance and the algorithmic task. The output involves a text description of the intermediate steps and the final output. The results are evaluated by comparing the outputs to the ground truth.

We find that there is a large gap between the LLM results and the GNN results. We evaluate directly prompting the text description to the language model by showing m demonstrations and also a query. We also fine-tune Llama-3-8B with LoRA [13] based on the text description. For fine-tuning, we use the same number of training samples as in the MPNN. Table 8 shows the results. We find that directly prompting the Llama model performs poorly. While fine-tuning helps, it still lags behind MPNNs by up to 23%.

Can we combine the embeddings from the branching network along with a language model to enhance its graph reasoning performance? A natural idea is to use the node embeddings of the input graph trained from AutoBRANE and combine that with the embeddings of the text description. To align these two embeddings, we add a linear layer after the node embeddings. Then, these are used as the input to the language model for fine-tuning. The overall procedure is summarized in Algorithm 3.

Another benefit of using GNN embeddings is that they reduce the memory cost of LLMs by reducing the input lengths. Suppose a graph has $|V|$ nodes and $|E|$ edges, text-based descriptions of graphs use the flattened adjacency matrix (as in [29]) and incidence matrix (as in [6]), which have a length of $|V|^2$ and $|V||E|$, respectively. As the model's memory requirements typically scale with the square of these lengths, the overall memory usage becomes $O(|V|^4)$ and $O(|V|^2|E|^2)$. In contrast, using GNN-produced node embeddings creates an input of length $|V|$, thereby reducing the memory requirement to $O(|V|^2)$.

Other methods also exist for fusing the graph embeddings with the text embeddings. For instance, one approach is to apply a cross-attention layer between the graph and text embeddings, and then feed the transformed embeddings to the LLM. We find that training a cross-attention layer between the embeddings converges more slowly than directly training on the concatenated embeddings. Thus, we choose the simpler concatenation in our method.

Evaluation results. We observed that fine-tuning LMs on the text descriptions of tasks underperforms training GNNs. We now evaluate our algorithm that uses the embeddings of the GNN-based branching network to fine-tune LLMs. We use the text-based graph reasoning tasks from the CLRS-Text benchmark [29], training on 1,000 graphs and evaluating on validation and test sets, each containing 200 graphs. We use Llama-3-8B as the base model. For each task, we evaluate the accuracy between model outputs and the labels, averaged over intermediate steps

Table 8: We compare GNN training, prompt demonstration with m examples, parameter-efficient fine-tuning, and our approach that leverages GNN embeddings in fine-tuning. For prompting and fine-tuning, we use Llama-3-8B. We evaluate the average test accuracy between outputs and labels at every intermediate step. We run each experiment with three random seeds and report the average.

	BFS	Bellman-Ford	Dijkstra	MST
MPNN [37]	99.80	99.40	99.20	99.60
Prompting, $m = 0$	22.47	2.78	4.60	3.95
Prompting, $m = 5$	34.28	31.54	25.17	25.31
Prompting, $m = 10$	34.65	32.44	25.32	25.98
Prompting, $m = 20$	35.88	33.56	25.80	26.59
PrefixTuning	92.1 ± 0.4	80.2 ± 0.6	65.3 ± 0.3	74.6 ± 0.8
Adapter	96.5 ± 0.6	83.1 ± 0.6	69.4 ± 0.4	74.3 ± 0.3
LoRA	98.2 ± 0.3	89.2 ± 0.2	75.6 ± 0.4	78.5 ± 0.5
Algorithm 3 (Ours)	**99.3 ± 0.2**	**92.6 ± 0.6**	**78.7 ± 0.6**	**79.4 ± 0.3**

We compare our algorithm with several parameter-efficient fine-tuning methods for fine-tuning Llama-3-8B. These include prefix tuning, adapter tuning, and LoRA fine-tuning. In our algorithm, we use LoRA to fine-tune the LM together with AutoBRANE. For all baselines, we train the same number of parameters as in our algorithm. Table 8 shows the results. Our algorithm outperforms the other fine-tuning baselines by **2.1**% on average.

C Omitted Experiments

C.1 Datasets and Models

CLRS Tasks. We summarize the input and output definitions of the algorithmic reasoning tasks in the CLRS benchmark [37] in Table 2. Each task represents an algorithmic problem where the input includes the graph structure itself.

GraphQA Tasks. We use twelve text-based graph reasoning tasks from the GraphQA benchmark [6]. We summarize the input and output definitions of the tasks in Table 10. While not all tasks involve an algorithmic problem, we evaluate on this benchmark to demonstrate the broad applicability of our algorithm.

GraphWiz Tasks. We use nine graph-related tasks from GraphWiz [4]. Compared to other datasets, this dataset provides textual descriptions of graph problem inputs and the reasoning processes that describe the rationale for solving the tasks generated by chain-of-thought prompting GPT-4. We summarize the input and output definitions of the tasks in Table 11.

Implementation. On CLRS, we use the encode-process-decode architecture introduced in Veličković et al. [37]. The encoder embeds inputs into feature representations of the algorithm. The processor performs message-passing computations of the input embeddings over the graph using GNNs. The decoder is a linear layer that transforms the final embeddings into a label space. The output of the decoder is viewed as the predicted steps for the next step and as the output at the final step. We train the encode-process-decode model with the sum of the losses computed on each step. At

Table 9: List of hyperparameters for each dataset tested in the experiments, corresponding to the results that we reported in Section 4.

Dataset	Model	m	α	L	k	T	B	Learning rate	Epochs
CLRS	Edge Transformer	200	3	5	10	2.5 hours	10.5 GB	$2e^{-4}$	8
CLRS	MPNN	200	3	5	10	0.3 hours	4.2 GB	$2e^{-4}$	8
CLRS-Text	Qwen-3-1.7B	200	3	28	7	6.0 hours	13.6 GB	$2e^{-5}$	10
GraphQA	Llama-3-1B	200	3	16	8	0.5 hours	12.4 GB	$1e^{-5}$	10
GraphWiz	Llama-3-1B	200	3	16	8	0.9 hours	21.5 GB	$1e^{-5}$	10
Orkut	SIGN	5000	25	3	22	0.1 hours	5.7 GB	$1e^{-2}$	100

Table 10: Definition of input and output of 12 graph reasoning tasks from GraphQA [6]. The input graphs are sampled from an Erdös-Rényi random graph distribution with edge sampling probability $p = 0.5$, with 20 vertices in total on each graph. The edge weights are uniformly sampled from the integers between 1 and 10. The source node is sampled randomly from the vertex set.

Task	Input Graph	Additional Input	Output
Edge existence	Undirected	Two target nodes	Binary label for edge existence
Node degree	Undirected	Target node	Scalar for node degree
Node count	Undirected	None	Scalar for number of nodes
Edge count	Undirected	None	Scalar for number of edges
Connected nodes	Undirected	Target node	List of connected nodes
Cycle check	Undirected	None	Binary label for cycle presence
Disconnected nodes	Undirected	Target node	List of disconnected nodes
Reachability	Undirected	Source and target nodes	Binary label for path existence
Shortest path	Undirected, weighted	Source and target nodes	Scalar for shortest path length
Maximum flow	Undirected, weighted	Source and target nodes	Scalar for maximum flow capacity
Triangle counting	Undirected	None	Scalar for number of triangles
Node classification	Undirected	Target node and other node labels	Node label of the target node

Table 11: Definition of input and output of 9 graph reasoning tasks from GraphWiz [4]. The input graphs are sampled from an Erdös-Rényi random graph distribution with edge sampling probability $p = 0.5$, with 100 vertices in total on each graph. The edge weights are uniformly sampled from the integers between 1 and 10. The source node is sampled randomly from the vertex set.

Task	Input Graph	Additional Input	Output
Cycle Detection	Undirected	None	Binary label for cycle presence
Connectivity	Undirected	Two target nodes	Binary label for path existence
Bipartite Graph	Undirected	None	Binary label for bipartite structure
Topological Sort	Directed acyclic	None	Node labels for the ordering of nodes
Shortest Path	Undirected, weighted	Source and target nodes	List of node labels for the shortest path
Maximum Triangle Sum	Undirected, weighted	None	Scalar for maximum triangle sum
Maximum Flow	Directed, weighted	Source and sink nodes	Scalar for maximum flow capacity
Hamilton Path	Undirected	None	Binary label for path existence
Subgraph Matching	Two undirected graphs	None	Binary label for subgraph isomorphism

each algorithm step t, the encoder generates embeddings for the nodes, edges, and graph-level features. The processor updates node embeddings, incorporating both the initial input features and the embeddings from previous steps. Finally, the decoder predicts intermediate steps or the final output step. The entire model is trained to optimize predictions across all steps.

Table 12: Measuring the error for approximating the algorithmic reasoning task loss, using the first-order Taylor's expansion around the initialization trained on all tasks. The results are averaged over 50 random subsets of graph algorithmic reasoning tasks. We apply the approximation to parameters after layer l and use MPNN [37] or edge transformer [30] as the base model.

RSS: Base model = MPNN				RSS: Base model = Edge Transformer			
Dist.	Freeze layer 1	Freeze layer $1, 2$	Freeze layer $1, 2, 3$	Dist.	Freeze layer 1	Freeze layer $1, 2$	Freeze layer $1, 2, 3$
2%	$2.3_{\pm 0.7} \times 10^{-3}$	$2.0_{\pm 0.6} \times 10^{-3}$	$2.3_{\pm 0.4} \times 10^{-5}$	2%	$3.9_{\pm 0.4} \times 10^{-3}$	$3.0_{\pm 0.2} \times 10^{-3}$	$4.1_{\pm 0.2} \times 10^{-5}$
4%	$7.0_{\pm 0.8} \times 10^{-3}$	$6.0_{\pm 1.7} \times 10^{-3}$	$9.4_{\pm 1.7} \times 10^{-5}$	4%	$6.5_{\pm 0.5} \times 10^{-3}$	$6.4_{\pm 1.0} \times 10^{-3}$	$8.6_{\pm 2.1} \times 10^{-5}$
6%	$8.2_{\pm 1.4} \times 10^{-3}$	$8.0_{\pm 1.2} \times 10^{-3}$	$2.1_{\pm 0.4} \times 10^{-4}$	6%	$7.7_{\pm 1.6} \times 10^{-3}$	$7.4_{\pm 0.9} \times 10^{-3}$	$3.4_{\pm 0.7} \times 10^{-4}$
8%	$8.6_{\pm 1.2} \times 10^{-3}$	$9.1_{\pm 1.6} \times 10^{-3}$	$3.7_{\pm 0.7} \times 10^{-4}$	8%	$8.4_{\pm 2.4} \times 10^{-3}$	$8.5_{\pm 2.0} \times 10^{-3}$	$4.5_{\pm 1.8} \times 10^{-4}$
10%	$9.2_{\pm 2.4} \times 10^{-3}$	$9.4_{\pm 2.1} \times 10^{-3}$	$5.8_{\pm 1.1} \times 10^{-4}$	10%	$9.6_{\pm 1.1} \times 10^{-3}$	$9.0_{\pm 0.8} \times 10^{-3}$	$6.7_{\pm 2.2} \times 10^{-4}$

C.2 Omitted Results

First-order approximation. In Table 12, we measure the first-order approximation error by freezing different numbers of bottom layers of the models. Linear approximation consistently yields under 1% error, for edge transformers [30] and MPNN [37]. The approximation becomes better in higher layers. The RSS falls below 0.06% after freezing the first three layers.

Depth-first search. To understand the limitations of graph neural networks on Depth-First Search (DFS), we conduct an empirical case study using star graphs. We hypothesize that standard GNNs would fail to learn the DFS task on star graphs, because the algorithm requires predicting distinct node labels, whereas standard message-passing aggregations cannot distinguish between leaf nodes. To test this, we train a GIN model with the max aggregation operation and evaluate the final-step node prediction accuracy on star graphs (a central node connected to leaf nodes). We vary the number of nodes between 10, 20, 50, and 100, and the number of layers between 1, 2, and 3. We use 2,000 graphs in the training dataset.

To test our hypothesis, we evaluate two types of input node-level features: one where all nodes are initialized with identical input features, and another where node indices are included in the input features. As shown in Table 13, when all nodes are provided with identical features, the GNN model cannot fully predict the node labels. When the input features encode node indices, a 1-layer model achieves near 100% test accuracy. This is because executing DFS on the star graphs requires identifying the center node (which the model infers from its high degree) and subsequently visiting the leaf nodes in increasing index order. Including the node indices in the input features enables the model to learn this ordering.

Furthermore, performance degrades as the number of layers increases. While the 1-layer model achieves near 100% accuracy, the 2-layer and 3-layer models perform progressively worse. This is because two layers aggregate features across the entire star graph and leave the model unable to accurately differentiate the leaf node labels.

Additional evaluations on CLRS-Text. Next, we conduct an empirical evaluation on graph algorithmic reasoning datasets, as studied in the CLRS benchmark [37, 29]. We use twelve text-based graph algorithmic datasets, including the Bellman-Ford algorithm, BFS, and DFS. The input sample is encoded as a flattened adjacency matrix of the graph, and intermediate steps are encoded as lists of node labels. We fine-tune the pretrained Qwen-3-1.7B model on graphs of 10 nodes, using LoRA [13] as the base fine-tuning method. We evaluate the error rates as one minus the average exact match across output text sequences.

Table 13: Test accuracy (%) evaluated at the final step of the DFS task on star graphs. We train a GIN model using varying graph sizes and number of layers, comparing node input features with and without node indices. l denotes the number of layers, and n denotes the number of nodes. We report the average accuracy and the standard deviations across three random seeds.

		$n = 10$	$n = 20$	$n = 50$	$n = 100$
Using input features including node indices	$l = 1$	100.0 ± 0.0	98.8 ± 1.1	98.1 ± 2.2	95.9 ± 2.1
	$l = 2$	100.0 ± 0.0	66.1 ± 1.5	61.0 ± 2.0	57.0 ± 2.0
	$l = 3$	100.0 ± 0.0	64.7 ± 1.0	68.7 ± 1.1	67.4 ± 0.9
Using input features without node indices	$l = 1$	64.3 ± 0.0	58.0 ± 0.0	53.5 ± 0.0	52.9 ± 0.0
	$l = 2$	64.3 ± 0.0	58.0 ± 0.0	53.5 ± 0.0	52.9 ± 0.0
	$l = 3$	64.3 ± 0.0	58.0 ± 0.0	53.5 ± 0.0	52.9 ± 0.0

First, we find that the sample complexity of generating intermediate steps can be higher than predicting only the final step. We compare (A1) fine-tuning with intermediate steps and (A2) fine-tuning with only the final step, in terms of the errors of the final step. We observe that under low-sample regimes (e.g., within 1,000 samples for the Bellman-Ford algorithm), A1 performs worse than A2. However, with sufficient data samples, A1 ultimately outperforms A2 in all algorithms.

Second, we find that training on certain combinations of datasets can either positively or negatively affect the average sample complexity. Consider the example shown in Figure 2. The Bellman-Ford algorithm shares all steps with BFS on the graph, whereas DFS shares only the first step with BFS. One might expect that combining the datasets of Bellman-Ford and BFS would improve their average sample complexity. We fine-tune a model on each pair of datasets (B1) and compare it with training on each dataset alone (B2). We find that for Bellman-Ford and BFS, B1 yields 4.8% lower error rates than B2 on average across multiple sample sizes. On the contrary, for Bellman-Ford and DFS, B2 yields 12.1% higher error rates on average. In all $\frac{12\times11}{2} = 66$ cases, we find 34 cases where B1 improves sample complexity relative to B2.