Generalization in Graph Reasoning: A Systematic Comparison of LLM Training Approaches

Sola Shirai, Kavitha Srinivas, Julian Dolby,
Michael Katz, Shirin Sohrabi, Horst Samulowitz
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Abstract

For large language models (LLMs), reasoning over graphs can help solve many problems. Prior work has tried to improve LLM graph reasoning through different training methods, but the merits of such approaches remain unclear and the limitations of each approach with respect to generalizability of reasoning are often not thoroughly explored. In this paper we systematically compare the ability of LLMs to learn fundamental graph tasks across a variety of training methods and their ability to generalize out of distribution across various dimensions. We highlight key tradeoffs between training methods, e.g., training specialized graph encoders and fusing their embeddings with LLMs consistently collapses in terms of generalizability; however, no single method shows clear superiority across all dimensions of generalizability, regardless of the size of the model.

1 Introduction

For large language models (LLMs), reasoning over graphs helps solve many problems which involve answering questions that combine graphs with textual data – for instance, in knowledge graph question answering (Ji et al., 2024), graph-RAG (Edge et al., 2024), or code localization and search (Chen et al., 2025). Because this ability is often limited in pre-trained LLMs, approaches often serialize the neighborhood structure of a node into natural language so that it can be unified with textual data. This usually works well for smaller neighborhoods (1 or 2 hops) but often breaks down for larger ones. Hence it would be ideal if LLMs could process reasonable sized graphs directly.

In general, in existing work, graphs used for testing have been small (20 or fewer nodes), and the generalizability has been a problem when examined; performance on new tasks is often not better than chance (and chance is often not reported). Generalization to different graph sizes is often poor, even in going from 10 node graphs to 25 node graphs, as has generalization to different tasks (Zhang et al., 2024). Reinforcement learning (RL) is often believed to be a better option for generalizability, but even work using RL suggests strong generalization to different tasks shows rather severe degradation in performance on graph sizes of 25 node graphs, which is not practical for real world problem solving (Peng et al., 2025). Generalization to 35-100 node graphs has been shown using GRPO (Guo et al., 2025), but again, it is unclear what random performance is on these tasks.

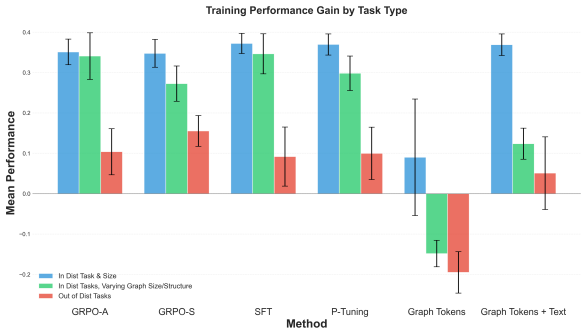

Figure 1: Average performance gains of the LLM on graph tasks for different training methods compared to base model performance on Phi-4-mini.

Another dimension of generalizability is the task type. Here too, existing work has targeted widely different tasks, often showing generalization from training on graph-theoretic tasks such as reachability to testing on knowledge graph question answering, link prediction, node classification, or even mathematical reasoning (e.g., Peng et al. (2025); Guo et al. (2025)). However, it is unclear *why* such generalization might be expected or which tasks contribute to generalizability across domains.

Training methods have also varied significantly to teach LLMs graph reasoning. It is unclear which of these methods are effective to teach LLMs graph reasoning. The goal of this work is to therefore compare these different training methods, and observe how well they teach an LLM to generalize

Proceedings of the First Workshop on Structured Understanding, Retrieval, and Generation in the LLM Era (SURGeLLM 2026), pages 275–297

across different graph sizes and different (but related) tasks. Specifically we compare the following training methods: (a) **Graph Tokens**, which encodes the graph with a graph encoder, and a projection layer to integrate with the LLM (Perozzi et al., 2024). Training is on the encoder and projection layer; LLM weights are frozen. (b) **Graph Tokens + Text** where a textual representation of the graph is added, so training can establish their correspondence as in multi modal training. (c) **supervised fine tuning (SFT)**, which fine tunes the LLM using LORA adaptors for efficient training (Hu et al., 2021), (d) **P-Tuning** as a soft prompt technique (Liu et al., 2022), where the LLM gets the graph solely as text, and a continuous prompt encoding is learned during training (e) **GRPO** (enhanced with DAPO (Yu et al., 2025)) where RL is used to train the model using a reward function - we tried two different reward functions.

Figure 1 highlights key tradeoffs between training methods. Graph Tokens consistently collapse in out of distribution tasks or out of distribution graph sizes (even when their in distribution performance is strong). SFT and RL based approaches show better generalizability; however, no approach demonstrates clear superiority across all dimensions. Efficacy of RL methods seem very sensitive to slight variations of the reward functions, with no seeming explanation for the sensitivity. Further as we show later, this result holds for the 3.7 times larger Phi-4 model, where RL performance drops off severely with slight modifications in the reward. RL does not appear to be the magic bullet for training on graph theoretic tasks, and generalization for graph reasoning continues to be fragile for LLMs.

2 Related work

2.1 Limitations of LLMs for Graph Tasks

Several papers (Liu and Wu, 2023; Dai et al., 2024) found that even state of the art models struggle to solve simple graph tasks. Sanford et al. (2024) and Abbe et al. (2024) described theoretical limitations of learning graph tasks using transformer models, suggesting that learning certain tasks such as cycle checks are inherent to their design.

2.2 Graph Theoretic Benchmarks

Benchmark datasets have also been developed for better evaluation, and better training of graph reasoners such as NLGraph (Wang et al., 2023), GraphOmni (Xu et al., 2025), GraphQA (Sanford et al., 2024), and NLGift (Zhang et al., 2024), but most have graphs with fewer than 50 nodes; our work explores testing models on graphs with up to 140 nodes and further evaluated how well LLMs learn to solve tasks on out-of-distribution graphs and tasks. GraphWiz (Chen et al., 2024a) introduced a new dataset for training on graphs called GraphInstruct, which is similar to the GraphQA dataset but enhanced with instructions. A key difference of our approach is that we produce instructions deterministically, while GraphWiz generates reasoning text using an LLM. GraphSilo (Peng et al., 2025) is a step wise reasoning benchmark for graph theoretic tasks, the steps are generated using automated trajectories and Monte Carlo Tree Search, but the largest graphs included in their test set is 26-35 node graphs. Erdõs (Guo et al., 2025) consists of 50 graph theoretic tasks with automated generation of solutions and verification (for reinforcement learning) using NetworkX, as in our approach. To perform a systematic analysis of generalizability, we focus our choice of tasks on a much more limited set, where task pairs were deliberately chosen to examine if generalization occurred where it might be expected to occur. Compared to all other benchmarks, we also more carefully control the distribution of task answers, so we know exactly what performance is expected by chance – for example, while GraphWiz claimed to show zero-shot generalizability to unseen tasks, inspecting their data revealed that the model performance was on par with random chance.

2.3 LLMs as Graph Reasoners

Prior work such as InstructGLM (Ye et al., 2024) explored how language models can serve as valid replacements for GNNs in learning to solve graph tasks. LLaGa (Chen et al., 2024b) and GraphEdit (Guo et al., 2024) convert graphs into node sequences and answers questions such as node classification and link prediction; our work is geared more to understand how well LLMs understand graphs as a basic structure, in graph theoretic tasks. PromptGFM (Zhu et al., 2025) similarly simulated a GNN workflow within an LLM, but as with LLaGa, the focus is not in testing graph reasoning directly in LLMs. GraphPRM (Peng et al., 2025) used DPO as a training objective to train a process reward model on small graphs (under 35 nodes). DPO is integrated within existing LLMs to guide the choice of reasoning paths that the model

provides through beam search or best-of-n search. GraphPRM showed generalizability to math reasoning tasks but did report a dramatic decline in performance as a function of graph sizes. Our work uses GRPO to directly tune the generation of an LLM, like Guo et al. (2025), but we compare across multiple training methods. Interestingly, Guo et al. (2025) reported that direct SFT without any chain of thought (CoT) does out perform GRPO on some tasks (and oddly CoT-SFT does worse), but they did not test if different methods allow different generalization. Zhang et al. (2024) examined whether LLMs generalize across semantic, numeric, structural, reasoning differences in the synthetic training data, and if this training helps them improve on real world graph reasoning tasks. As in many other works, the tested graphs tended to be small, but even in this case, the authors observed poor generalization across tasks, and suggest that using DPO for post training alignment might yield the best generalization across tasks. Our work contrasts multiple training regimens and how well they generalize across tasks, graph structures and graph sizes.

2.4 Multi Modal Approaches

In Graph Tokens (Perozzi et al., 2024) (see also Chai et al. (2025)), GNNs are used to encode graphs into node or edge vectors, after which aggregation is used to produce a final set of embeddings to input to the LLM. Both of these works only explored very small graphs (less than 20 nodes). In addition, Graph Tokens uses a different GNN and tokenization architecture for each task. Performance of a random baseline was not clear in either work, making relative improvement unclear, and generalization was not measured systematically, as it is in our work.

GraphGPT (Tang et al., 2024) used a pretrained graph encoder and an alignment mechanism to project graph tokens to an LLM. The LLM was trained in an unsupervised manner to output the order of node text to learn the alignment between text and the graph. GraphGPT was tested with knowledge graph completion tasks; the focus was not on testing graph theoretic reasoning in LLMs directly. GOFA (Kong et al., 2024) created a graph model integrated with LLMs by interweaving LLM compressors with GNN layers that capture the structure of the graph, targeting knowledge graph completion, rather than graph theoretic tasks.

3 Methods

3.1 Graph Question Answering Tasks

We investigate 8 graph tasks inspired by GraphQA (Sanford et al., 2024). The easiest 4 are retrieval tasks, solvable by a simple lookup or aggregation: **Node Count** (NC) asks the number of nodes in the graph. **Node Degree** (ND) is the outgoing degree of a particular node. **Edge Existence** (EE) asks if an edge from node A to node B exists. **Edge Count** (EC) is the number of edges in the graph.

We consider three perspectives on the reachability (or connectivity) task, which asks whether a path exists in the graph from node A to node B. We investigate *how to solve* the task, using **Depth First Search** (DFS) or **Breadth First Search** (BFS). Additionally, **Cycle Check** (CC) asks if the graph has a cycle starting from a target node (i.e., a reachability task from node A to itself). Note that this differs slightly from related work, which typically ask if a graph contains *any* cycle.

Our last task is **Shortest Path** (SP), which returns a list of nodes visited in the shortest path connecting node A to node B, or an empty list if there is none. We determine a single correct shortest path based on a BFS traversal, with nodes ordered by their node ID during traversal.

We use only four tasks at training (**NC**, **ND**, **DFS**, **BFS**), and the rest are used for testing to see if the model can generalize to new, but related, tasks. We also control the distribution of answers for tasks, to reduce class imbalance and control for the performance of a random chance baseline. Each task is asked in natural language, for which we use templates to produce several variations of each question. (Examples in Appendix A).

3.2 Textual Graph Representation

Fatemi et al. (2024) show that the textual representation of a graph affects LLM performance. We investigate the performance of models across multiple tasks on a single representation style, using insights from prior works.

Figure 2 shows an example of our graph representation. Ellipses indicate prompt text omitted for brevity. First, nodes are indicated using integer IDs, and edges as an incidence list; this helps reduce context length. Second, nodes are associated with semantics using a dictionary mapping node IDs to labels. These labels are randomly generated words from an English dictionary and random numbers (to simulate e.g. dollar values and percentages),

and serve to add some textual components to the graph while avoiding any implicit reliance on semantics associated with real text graphs. Last, in our question text, we use the node labels in the question rather than node IDs.

...
G describes a graph among nodes with the following mapping from IDs to labels:
{0: visible, 1: askance, 2: oikoplast, ..., 5: sleepingly}
The edges in G are:
0 → 1, 3
2 → 0, 4
...
Q: Is there a breadth first traversal path from “oikoplast" to “sleepingly"?

Figure 2: Example of our graph representation.

We use this format to encode graphs for the input prompts for all models except Graph Tokens; there, we omit the text describing the graph's edges but include the ID to label mapping. While prior work has investigated how to encode semantic information about nodes into the graph encoding, in our preliminary studies we found that simple semantics such as random labels were not effectively encoded by the GNN. We therefore opted to use LLM prompting to add this semantic information about nodes instead[1].

3.3 Reasoning Steps for Task Solutions

Having an LLM produce a chain of thought (CoT) (Wei et al., 2022), especially as a type of reasoning chain, has become a mainstay approach in enhancing the capabilities of LLMs. Our own experimental validation (see Appendix B) demonstrated the utility of this approach for graph reasoning, and apply it to all model trained using SFT.

For each task, we implement a template to produce reasoning steps deterministically from code as part of the training and validation data. At test time, accuracy is only judged based on the final answer. As in prior work, the LLM was instructed to output steps within `<think/>` tags and the final answer in `<answer/>` tags.

Following Aytes et al. (2025)'s style of concise sketches as CoT, our reasoning steps use a mixture of python-like lists and arrows to indicate elements such as edges and paths, along with natural language describing the approach at the beginning of the CoT. An example for our BFS task is shown in Figure 3 – ellipses indicate text omitted for brevity. A full listing of CoT text examples for our tasks can be seen in Appendix A.

For GRPO, we assess the validity of the steps using code to ignore order of node choice in traversal. The input prompt for GRPO training is slightly modified to specify a structure for steps (e.g. edge tuples), which are then parsed for validation – checking for the correct structure, such that steps can be validated, is also a component of our reward function.

<think> Starting breadth first traversal from oikoplast to see if sleepingly is reachable. ...
Queue: [2]
2 → 0, 4
Queue: [0, 4]
...
Queue: []
No unvisited nodes remain.</think>
<answer>No</answer>

Figure 3: Instructive CoT for the BFS task.

3.4 Training Methods

We explore how well LLMs can learn graph tasks using six different methods, training adapter models (e.g., LoRA (Hu et al., 2022)) rather than full model training. Across all models, we use standard cross entropy loss to perform causal language model training. Further details on each training method are available in Appendix C.

GRPO (Shao et al., 2024): We train two variants of our model using GRPO-style training, where we produce multiple responses for each input question which are scored using a reward function to influence the LLM's preference. Our reward functions focus on rewarding the correct structure for output tasks, algorithm steps (e.g. the next edge to explore in a DFS trace), and final answer. The first variation, **GRPO-A** is trained using a reward function which places relatively more reward on the final answer. The second variation, **GRPO-S**, places a larger reward on correctly producing thinking steps for solving the problem. We used the TRL implementation of GRPO with DAPO (Yu et al., 2025) for training.

SFT: We train a LoRA adapter using supervised fine-tuning (SFT) over our training data, where each graph task has an associated CoT.

P-Tuning (Liu et al., 2024): We train a prompt encoder adapter, which inserts a fixed number of learnable embeddings into the LLM together with the input prompt. Tokens produced by P-Tuning augment the prompt, providing embeddings as a “prompt” rather than discrete text.

[1]Preliminary experiments (Appendix E) showed poor performance on the graph tokens approach without this text.

Graph Tokens (Perozzi et al., 2024): We train a graph encoder model to encode an input graph into a set of tokens using a GNN, augmenting the question input to an LLM. The GNN is trained using the cross entropy loss of the LLM. Unlike P-Tuning's static tokens, graph tokens differ for each input graph. In this approach, textual descriptions of the graph are not provided to the LLM.

Graph Tokens + Text: This method combines Graph Tokens with textual descriptions of the graph (i.e., the same input as LoRA and P-tuning). This model type allows graph tokens created by the GNN to benefit from alignment with the textual representation of the graph.

4 Experiments

4.1 LLM Choice

Our main experiments are performed using Phi-4-Mini-Instruct (Abouelenin et al., 2025), a 4-billion parameter LLM. Phi-4-mini was selected due to strong performance shown during early experimentation, as well as suitability in terms of memory requirements versus performance. All training and inference performed with this model can fit within the memory limits of a single 80GB GPU. While our main results focus on Phi-4-mini, additional experiments on Phi-4 (14B parameters) and different LLM architectures (Granite3, Qwen3) shows similar patterns (see Appendix F and H for results).

4.2 Datasets

As in GraphQA (Sanford et al., 2024), we synthetically generate graph data and associated questions. All graph data used in our experiments are directed and unweighted. To address our research questions surrounding how well LLMs can learn to solve graph tasks as well as different facets of generalizability, our test datasets are constructed primarily with graph structures, graph sizes, and tasks that were unseen or poorly represented during training.

Unlike prior work, answer classes are balanced. Fatemi et al. (2024) note that Erdös-Rényi (ER) graphs (Erdös and Rényi, 1959) resulted in cycle check tasks where 82% of the samples contained cycles. To address such issues of class imbalance, we enforce constraints on the number of samples which have any one answer. Table 1 summarizes our constraints for each task, with random accuracy indicating the constraint we place on answer distributions. Similar constraints are placed on both training and testing data, to ensure that class imbalance is addressed at each stage.

Task	Ans. Type	Random Acc.
Node Count (NC)	Numeric	10%
Node Degree (ND)	Numeric	15%
Depth-First Search (DFS)	Boolean	50%
Breadth-First Search (BFS)	Boolean	50%
Edge Existence (EE)	Boolean	50%
Edge Count (EC)	Numeric	5%
Shortest Path (SP)	List	15%*
Cycle Check (CC)	Boolean	50%

Table 1: Task and random accuracy overview. *For SP, 15% of answers have the same shortest path *length*, and actual path lists are rarely repeated.

Data Split	Nodes	Edges
Train	[5, 20]	[1, 300]
Validation	[5, 20]	[1, 300]
In Dist. Test	[5, 20]	[1, 300]
OOD Graph Size	[20, 100]	[1, 500]
OOD Graph Size - XL	[140, 160]	[1, 500]
OOD Structure - Trees	[20, 100]	[20, 100]
OOD Structure - Cycles	[20, 120]	[20, 120]
OOD Tasks	[20, 100]	[1, 500]
Edge-Labeled Graphs	[5, 20]	[1, 300]

Table 2: Test dataset graph sizes.

4.3 Training Data

We generate training and validation data for graphs containing 5 to 20 nodes, and bound the number of edges to 300. The constraint on the edges is motivated by memory limitations during training, ensuring that training could be performed on a single 80GB GPU. This also gives us an opportunity to test the generalization ability of our learned models to graphs with many more edges than were seen during training.

We generate **3,000** training samples for each training tasks, totaling **12,000** samples for multi-task training. As in GraphQA, we select 7 algorithms to generate graphs; Erdös-Rényi, Barabási–Albert (Barabási and Albert, 1999), Stochastic Block Model (Holland et al., 1983), Scale-Free graphs (Bollobás et al., 2003), Star graphs, Path graphs, and Directed Acyclic graphs.

We train our two GRPO models for 5 epochs (with 8 generations for each sample), SFT and P-Tuning models for up to 20 epochs, and our Graph Token and Graph Token + Text models using a Relational Transformer (**?**) as the graph encoding model for up to 40 epochs. Training details can be found in Appendix C.

4.4 Testing

We generate several variations of testing data for graph tasks to better assess the ability of various approaches to learning to solve graph tasks and generalizing to out-of-distribution (OOD) conditions and tasks. Unless otherwise noted, test graphs are generated using the same 7 graph structures as for training. Key statistics about the tasks and graph sizes for our test data can be seen in Table 2. Additional details on data generation and statistics for our test data can be found in Appendix D.

In Distribution: 500 samples are generated for each task using the same procedure as for training data generation.

OOD Graph sizes: 500 samples are generated for each task for graphs containing 20 to 100 nodes. We also produce extra-large OOD graphs, containing 140 to 160 nodes.

OOD Graph Structures (Trees and Cycles): We experiment with two types of graph structures which were not generated by our default set of graph generation algorithms. One set is constructed by generating random power law trees, using networkx, containing 20 to 100 nodes (500 test samples per task). A second set is constructed for only the BFS and DFS tasks; *cycle graphs* are generated similarly as in Abbe et al. (2024), where graphs consist of either 1 large cycle, for tests where nodes in question are reachable, or two smaller cycles, for tests where the nodes are not reachable. Graphs are constructed such that for a graph n nodes can be correctly answered by following a DFS trace for $n/2$ steps – this structure also presents much longer search traces than those seen during training. Cycle graphs are constructed for graphs containing 20 to 120 nodes.

OOD Tasks: 500 test samples are generated for each of 4 graph tasks which were not used for training – EE, EC, SP, and CC – using the same procedure as our in-distribution test generation. Each OOD task was designed to see if the model would be able to generalize from the training task to a related out of distribution task.

Edge-Labeled Graphs: Lastly, 500 test samples are produced for edge-labeled graphs, where the graph is split into subgraphs based on randomly assigned edge labels. Each graph has between 2 to 5 different edge types. For this dataset, tasks are variations of the tasks seen during training, where tasks are specified as only applying to nodes connected by specific edges (e.g., BFS following only one type of edge) or following any of the edges. An example of this task is shown in Figure 4.

...
Edges are represented as x { i → j,k }, meaning that there is an edge from node i to node j, and another edge from node i to k, and all edges within the subgraph have an edge type of x. G describes a set of subgraph among nodes with the following mapping from IDs to labels:
{0: $-13,776, 1: panoplist, ... }
The sub-graphs and edges in G are:
subgraph loutish {
0 → 1, 3, 4, 10, 12
1 → 11, 12
...}
subgraph reunion {
0 → 5, 8
1 → 0, 6, 8, 9
...}
Q: Can you reach “panoplist" from “$-13,776" using a depth first traversal along edges “reunion"?

Figure 4: Example edge-labeled graph and question.

The ND, DFS, and BFS tasks have edge-specific variants for this set of graphs, as well as the standard in-distribution tasks where the edge labels are ignored. This set of test graphs therefore introduces both OOD structure and tasks, although the tasks are much more directly comparable to the training tasks than the other OOD tasks.

5 Results

5.1 Generalization across graph sizes

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
In Distribution (smaller) Graphs				
0-Shot Prompt	.552	.614	.678	.668
GRPO-A	1.00	.982	.970	.964
GRPO-S	.998	.992	.966	.946
SFT	1.00	1.00	1.00	1.00
P-Tuning	1.00	1.00	.994	.996
Graph Tokens	1.00	.256	.798	.818
Graph Tokens + Text	1.00	1.00	.988	1.00
Out of Distribution (larger) Graphs				
0-Shot Prompt	.130	.296	.532	.514
GRPO-A	**.834**	.844	.818	.830
GRPO-S	.586	.894	.716	.762
SFT	.418	**.982**	**.932**	**.964**
P-Tuning	.354	.934	.900	.902
Graph Tokens	.138	.144	.560	.592
Graph Tokens + Text	.254	.718	.704	.748
Out of Distribution (extra large) Graphs				
0-Shot Prompt	.004	.226	.360	.336
GRPO-A	**.748**	.700	.582	.598
GRPO-S	.018	.764	.508	.516
SFT	.026	**.916**	**.728**	**.782**
P-Tuning	.066	.720	.654	.624
Graph Tokens	.002	.028	.120	.106
Graph Tokens + Text	.048	.290	.224	.194

Table 3: Test results on graphs with 5-20 nodes (smaller), 20-100 nodes (larger), and 140-160 nodes (extra large).

Table 3 shows results for base model (0-shot

prompt[2]) and trained models. Training is performed on 4 main tasks: NodeCount (NC), NodeDegree (ND), DFS, BFS using 5-20 node graphs. As shown in Table 3, the base model starts with significantly enhanced performance on the simple tasks of node count and node degree, but struggles on reachability. All training methods train the model well on all 4 tasks for small graphs, with the notable exception of Graph Tokens which actually degrades base model performance on node degree, but does enhance performance on DFS and BFS. Interestingly, prompt tuning which we added as a baseline for graph tokens did not suffer as much as graph tokens.

On the larger graph sizes, performance on even NC drops to chance for the base model. And as can be seen in the table, the most effective methods for out of distribution generalization varied, but graph token performance collapsed. Among other methods, GRPO-A was effective for learning NC, but was clearly less effective than SFT or prompt tuning for ND, and reachability. SFT shows strong generalization to larger graphs, but the drop in NC performance is likely explained by the fact that the distribution of *answers* differs – DFS and BFS are always answered as "Yes" or "No", while NC outputs integers between 5 to 20 during training.

5.2 Generalization to related tasks

Table 4 shows results for applying the models to related tasks (EdgeExistence (EE), ShortestPath (SP), EdgeCount (EC), CycleCheck (CC)) with no extra training. Graph sizes are also larger than those seen during training, containing 20-100 nodes. In general, GRPO-S seemed to generalize best to related tasks as did SFT or prompt tuning. GRPO-A and -S both performed well on cycle check while as all SFT based methods seemed to overfit and actually degrade base model performance – given that the BFS/DFS tasks never used the same start and end nodes during training, a typical failure pattern of the SFT-based methods was to endlessly continue searching even after the target node was reached. One thing to note is the inconsistency in GRPO performance as a function of slight changes in reward functions – GRPO-A seemed to work well on CC, while its performance on DFS/BFS was not as good on out of distribution graph sizes.

[2]We found 3-shot prompts to yield similar results

Method	EE	SP	EC	CC
Random	.500	.150	.050	.500
0-Shot Prompt	.884	.276	.008	.726
GRPO-A	.684	.208	.060	**.930**
GRPO-S	**.922**	**.376**	.094	.728
SFT	.878	.304	**.108**	.352
P-Tuning	.842	.314	.070	.458
Graph Tokens	.464	.044	.034	.346
Graph Tokens + Text	.584	.258	.046	.340

Table 4: Results from applying trained models to related, but unseen, tasks with no extra training.

5.2.1 Generalization to graph types

Edge labeled graphs: Table 5 shows results for generalization to edge labeled graphs. Here the efficacy of the training method relies on the task type. When asked to perform the same tasks as in training but only on a given edge, GRPO style training seems most effective, but when the task is to ignore edge labels and only generalize across a change in how the graph is formatted (now each edge label has a subgraph) the graph tokens plus text approach and Prompt tuning approaches are competitive with GRPO. Graph tokens by itself continues to perform poorly across all tasks, degrading base model performance. Note that the NC task has no edge-specific version as it asks for the total number of nodes in the graph.

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
Tasks only target one edge type				
0-Shot Prompt	-	.438	.802	.808
GRPO-A	-	.908	**.876**	.838
GRPO-S	-	**.910**	.854	**.852**
SFT	-	.788	.822	.782
P-Tuning	-	.800	.786	.782
Graph Tokens	-	.178	.436	.424
Graph Tokens + Text	-	.588	.792	.754
Tasks apply to any edge type				
0-Shot Prompt	.680	.242	.662	.588
GRPO-A	.996	.330	.850	.838
GRPO-S	.954	**.428**	.900	**.906**
SFT	.994	.386	.892	.886
P-Tuning	**1.00**	.396	.906	.878
Graph Tokens	.458	.204	.472	.484
Graph Tokens + Text	.998	.426	**.922**	.898

Table 5: Results on edge labeled graphs. For both task variants, the graph is split into subgraphs based on edge labels, but the question asks to only follow one edge type or use any edge type.

Trees: Table 6 shows results for the main tasks for graphs which are shaped like trees. Here again we see that except for NC, where GRPO-A does better, SFT and prompt tuning perform best in terms of generalizing to tree structures.

Cycles: Table 7 shows results for DFS and BFS

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
0-Shot Prompt	.150	.394	.604	.600
GRPO-A	**.888**	.890	.728	.738
GRPO-S	.488	.926	.840	.870
SFT	.318	**.994**	**.936**	**.976**
P-Tuning	.366	.978	.880	.942
Graph Tokens	.170	.130	.416	.406
Graph Tokens + Text	.138	.792	.822	.860

Table 6: Results over graphs shaped like trees with 20-100 nodes.

Method	DFS	BFS
Random	.500	.500
0-Shot Prompt	.405	.411
GRPO-A	.500	.473
GRPO-S	.602	.527
SFT	**.693**	**.834**
P-Tuning	.650	.734
Graph Tokens	.334	.293
Graph Tokens + Text	.557	.620

Table 7: Results over graphs shaped like cycles with 20-120 nodes.

over the cycle graphs. We continue to observe that SFT and prompt tuning work best when applied to different graph structures.

A comparison of model performance against graph size for BFS on cycle graphs can be seen in Figure 5. Random chance is indicated by the dashed red line. Given that node count directly correlates with the path length for cycle graphs, we can observe a clear trend in how each approach's accuracy decreases as the difficulty of the problem increases. For other graphs we tested, reachability was not directly correlated to the size of the graph, and since source and targets were chosen at random, there was no direct connection between path length and graph size. From the figure, it is clear that base model accuracy drops to chance for ∼40-50 node graphs, but SFT training allows models to continue performing above chance for ∼100 node graphs. This is a significant generalization in terms of trace length seen during training.

5.3 Phi-4 Model Training Methods Efficacy

To ensure that the comparative similarity of RL methods against SFT was not purely due to model capacity, we compared performance gains of our two GRPO-based models and SFT using Phi4 to see how it compares. Summary results are shown in Figure 6 (see Appendix F for full result tables). Zero-shot performance of the base model is much stronger for Phi 4 than Phi 4 mini, and as a result we see that performance gains are therefore smaller.

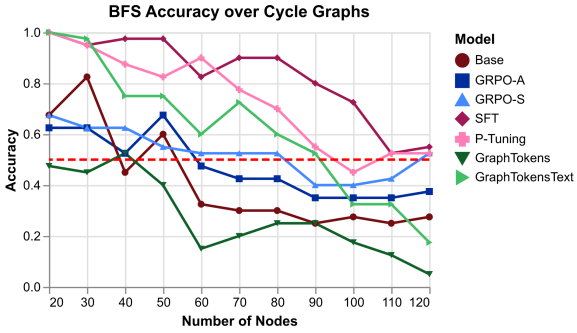

Figure 5: BFS accuracy v.s. graph size on cycle graphs.

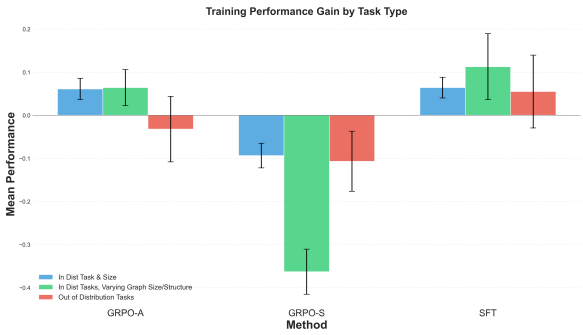

Figure 6: Performance gains for select training methods on Phi4.

Here SFT appears to be more useful than RL, even for OOD tasks. Note that GRPO-S's performance is almost entirely negative compared to baseline performance, mostly due to the model continuously outputting thinking/steps and not reaching an answer within the maximum output token limit, despite having the same context limit as our results using Phi 4 mini. It is likely possible to adjust that using a penalty for answer length but even in the GRPO-A case, it seems that training has at best no improvement on OOD tasks.

6 Conclusion

Through extensive experiments, we reveal many shortcomings of existing approaches for training LLMs to reason over graphs when it comes to generalization. With good, programatically generated CoT, SFT can be used to train models to solve tasks, but we can observe how performance drops when applied to tasks and structures of graphs that were not seen during training. RL does not show consistent improvement in terms of generalization, and is sensitive to minor variations in reward functions. Overall, LLMs continue to be rather fragile on graph-theoretic tasks, with patterns in generalization being inconsistent.

Limitations

The explorations of this paper primarily focus on LLMs which are relatively small by modern standards – most results focus on Phi-4-mini (4 billion parameters), and we expanded our investigation up to a 32 billion parameter Qwen3 model. While we continued to observe issues with generalization in this span of model sizes, we cannot make ddfinitive claims about how generalization performance might change with even larger models. Our exploration was constrained to model sizes which could be trained using 80GB GPUs.

References

Emmanuel Abbe, Samy Bengio, Aryo Lotfi, Colin Sandon, and Omid Saremi. 2024. How far can transformers reason? the globality barrier and inductive scratchpad. In *Advances in Neural Information Processing Systems*, volume 37, pages 27850–27895. Curran Associates, Inc.

Abdelrahman Abouelenin, Atabak Ashfaq, Adam Atkinson, Hany Awadalla, Nguyen Bach, Jianmin Bao, Alon Benhaim, Martin Cai, Vishrav Chaudhary, Congcong Chen, et al. 2025. Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras. *arXiv preprint arXiv:2503.01743*.

Simon A Aytes, Jinheon Baek, and Sung Ju Hwang. 2025. Sketch-of-thought: Efficient llm reasoning with adaptive cognitive-inspired sketching. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 24307–24331.

Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science*, 286(5439):509–512.

Béla Bollobás, Christian Borgs, Jennifer Chayes, and Oliver Riordan. 2003. Directed scale-free graphs. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, page 132–139, USA. Society for Industrial and Applied Mathematics.

Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiang Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. 2025. Graphllm: Boosting graph reasoning ability of large language model. *IEEE Transactions on Big Data*.

Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. 2024a. Graphwiz: An instruction-following language model for graph computational problems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 353–364, New York, NY, USA. Association for Computing Machinery.

Runjin Chen, Tong Zhao, Ajay Jaiswal, Neil Shah, and Zhangyang Wang. 2024b. Llaga: large language and graph assistant. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727.

Xinnan Dai, Qihao Wen, Yifei Shen, Hongzhi Wen, Dongsheng Li, Jiliang Tang, and Caihua Shan. 2024. Revisiting the graph reasoning ability of large language models: Case studies in translation, connectivity and shortest path. *arXiv preprint arXiv:2408.09529*.

Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. 2019. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *International semantic web conference*, pages 69–78. Springer.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *ArXiv*, abs/2404.16130.

P. Erdös and A. Rényi. 1959. On random graphs i. *Publicationes Mathematicae Debrecen*, 6:290.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a graph: Encoding graphs for large language models. In *International conference on learning representations*, volume 2024, pages 43909–43934.

Xiaojun Guo, Ang Li, Yifei Wang, Stefanie Jegelka, and Yisen Wang. 2025. G1: Teaching llms to reason on graphs with reinforcement learning. *arXiv preprint arXiv:2505.18499*.

Zirui Guo, Lianghao Xia, Yanhua Yu, Yuling Wang, Kangkang Lu, Zhiyong Huang, and Chao Huang. 2024. Graphedit: Large language models for graph structure learning. *arXiv preprint arXiv:2402.15183*.

Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. 1983. Stochastic blockmodels: First steps. *Social Networks*, 5(2):109–137.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *CoRR*, abs/2106.09685.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3.

Yixin Ji, Kaixin Wu, Juntao Li, Wei Chen, Mingjie Zhong, Xu Jia, and Min Zhang. 2024. Retrieval and reasoning on KGs: Integrate knowledge graphs into large language models for complex question answering. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 7598–7610, Miami, Florida, USA. Association for Computational Linguistics.

Lecheng Kong, Jiarui Feng, Hao Liu, Chengsong Huang, Jiaxin Huang, Yixin Chen, and Muhan Zhang. 2024. Gofa: A generative one-for-all model for joint graph language modeling. *arXiv preprint arXiv:2407.09709*.

Chang Liu and Bo Wu. 2023. Evaluating large language models on graphs: Performance insights and comparative analysis. *arXiv preprint arXiv:2308.11224*.

Xiao Liu, Kaixuan Ji, Yicheng Fu, Weng Tam, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2022. P-tuning: Prompt tuning can be comparable to fine-tuning across scales and tasks. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 61–68, Dublin, Ireland. Association for Computational Linguistics.

Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2024. Gpt understands, too. *AI Open*, 5:208–215.

Miao Peng, Nuo Chen, Zongrui Suo, and Jia Li. 2025. Rewarding graph reasoning process makes llms more generalized reasoners. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, page 2257–2268, New York, NY, USA. Association for Computing Machinery.

Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. 2024. Let your graph do the talking: Encoding structured data for llms. *arXiv preprint arXiv:2402.05862*.

Clayton Sanford, Bahare Fatemi, Ethan Hall, Anton Tsitsulin, Mehran Kazemi, Jonathan Halcrow, Bryan Perozzi, and Vahab Mirrokni. 2024. Understanding transformer reasoning capabilities via graph algorithms. *Advances in Neural Information Processing Systems*, 37:78320–78370.

Abulhair Saparov and He He. 2023. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. In *The Eleventh International Conference on Learning Representations*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. 2024. Graphgpt: Graph instruction tuning for large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 491–500, New York, NY, USA. Association for Computing Machinery.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems*, 36:30840–30861.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Hao Xu, Xiangru Jian, Xinjian Zhao, Wei Pang, Chao Zhang, Suyuchen Wang, Qixin Zhang, Zhengyuan Dong, Joao Monteiro, Bang Liu, et al. 2025. Graphomni: A comprehensive and extendable benchmark framework for large language models on graph-theoretic tasks. *arXiv preprint arXiv:2504.12764*.

Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, and Yongfeng Zhang. 2024. Language is all a graph needs. In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 1955–1973, St. Julian's, Malta. Association for Computational Linguistics.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, YuYue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, LingJun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. 2025. DAPO: An open-source LLM reinforcement learning system at scale. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xiaochuang Han, Tianxing He, and Yulia Tsvetkov. 2024. Can LLM graph reasoning generalize beyond pattern memorization? In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 2289–2305, Miami, Florida, USA. Association for Computational Linguistics.

Xi Zhu, Haochen Xue, Ziwei Zhao, Wujiang Xu, Jingyuan Huang, Minghao Guo, Qifan Wang, Kaixiong Zhou, Imran Razzak, and Yongfeng Zhang. 2025. Llm as gnn: Graph vocabulary learning for text-attributed graph foundation models. *arXiv preprint arXiv:2503.03313*.

A Task Details and Example Text

A.1 Graph Representation

Figure 7 shows the full textual representation of a graph, which is prepended to the input question for all tasks and models, except for the Graph Tokens approach where we omit the portion from "The edges in G are:" onwards.

In a directed graph, the mapping of node IDs to their labels is given by a dictionary. Edges are represented as i -> j,k means that there is an edge from node i to node j, and another edge from node i to k. G describes a graph among nodes with the following mapping from IDs to labels:
{0: Phasmatida,
1: mandruka,
2: eulogy,
3: benzoiodohydrin,
4: Krishnaitic}
The edges in G are:
0 -> 1, 2
1 -> 3
2 -> 1, 4
4 -> 0

Figure 7: Example representation of a graph and question.

All subsequent example texts shown for the tasks in this section will use the above graph as input.

A.2 Task Examples

The following figures will show example questions as well as answer CoTs for each task. During inference, only text that comes before the <think> tag are input to the model. Each question has 5 alternate phrasings to try to reduce overfitting, including re-ordering the nodes in reachability questions (e.g. "can NodeA reach NodeB" versus "is NodeB reachable from NodeA").

Q: How many nodes does the graph contain?
A:
<think>To count the nodes in the graph, gather the nodes into a list:
The list of nodes is [0,1,2,3,4]
The number of nodes is the length of the list
5</think><answer>5</answer>

Figure 8: Node Count task.

Q: How many other nodes can be reached by "eulogy" in a single step?
A:
<think>From the label mapping, we know that eulogy maps to 2.
Counting neighbors of 2:
(2, 1) count:1
(2, 4) count:2
</think><answer>2</answer>

Figure 9: Node Degree task.

Q: Can a depth first search be used to reach "benzoiodohydrin" from "eulogy"?
A:
<think>Starting depth first traversal from eulogy to see if benzoiodohydrin is reachable. From the label mapping, eulogy maps to 2, and benzoiodohydrin maps to 3.
Starting depth first traversal from 2. Backtrack once there are no more new neighbors to visit or 3 is reached.
2 -> 1 -> 3
Reached 3
</think><answer>Yes</answer>

Figure 10: Depth-First Search task.

Q: Can a breadth first search be used to reach "benzoiodohydrin" from "eulogy"?
A:
<think>Starting breadth first traversal from eulogy to see if benzoiodohydrin is reachable. From the label mapping, eulogy maps to 2, and benzoiodohydrin maps to 3. Visit all new neighbors of nodes in the current depth's queue, skipping visited nodes.
Queue: [2]
2 -> 1, 4
Queue: [1, 4]
1 -> 3
Reached 3
</think><answer>Yes</answer>

Figure 11: Breadth-First Search task.

Q: Is there an edge from "eulogy" to "benzoiodohydrin"?
A:
<think>To check if the edge exists, eulogy is mapped to 2, and benzoiodohydrin is mapped to 3. Check if 2->3:
2 has edges to [4,1]
</think><answer>No</answer>

Figure 12: Edge Existence task.

Q: How many total edges are there in the entire graph? Write out a list of all edges in the graph, then answer the size of the list.
<think>To count edges, count the edges from each node in the graph:
The list of edges is: [0->1,0->2,1->3,2->4,2->1,4->0] The answer is the length of the list 6
</think><answer>6</answer>

Figure 13: Edge Count task.

B Comparison of Model Performance using CoT

As part of our preliminary exploration, we conducted experiments to validate the performance of various models when we trained the model to output a chain-of-thought reasoning process to answer questions versus when we trained the models to directly output an answer. For this set of experiments, we used a set of graphs containing only 5 to 20 nodes, similar to many other prior works. We

Q: What is the shortest path from "eulogy" to "benzoiodohydrin"? Use BFS to find the shortest path, then output the path as a list. If there is no path output an empty list.
A:
<think>Starting breadth first traversal from eulogy to benzoiodohydrin to find the shortest path. From the label mapping, eulogy maps to 2, and benzoiodohydrin maps to 3. Visit all new neighbors of nodes in the current depth's queue, skipping visited nodes.
Queue: [2]
2 -> 1, 4
Queue: [1, 4]
1 -> 3
Reached 3
Work backwards from reached 3 to nodes that have an backwards edge to 2 The shortest path is therefore:
</think><answer>[2, 1, 3]</answer>

Figure 14: Shortest Path task.

Q: Does the graph contain any cycles starting from "Krishnaitic"? Check if there is a path from "Krishnaitic" to itself.
A:
<think>Starting depth first traversal from Krishnaitic to see if Krishnaitic is reachable. From the label mapping, Krishnaitic maps to 4, and Krishnaitic maps to 4.
Starting depth first traversal from 4. Backtrack once there are no more new neighbors to visit or 4 is reached.
4 -> 0 -> 1 -> 3
0 -> 2 -> 1
2 -> 4
Reached 4
</think><answer>Yes</answer>

Figure 15: Cycle Check task.

train a LoRA adapter, P-tuning, and graph token model using SFT training.

Model	CoT?	NC	ND	DFS	BFS
LoRA	No	1.0	.99	.96	.97
LoRA	Yes	1.0	1.0	.99	1.0
P-Tuning	No	1.0	.31	.75	.81
P-Tuning	Yes	1.0	.99	.98	1.0
Graph Tokens	No	.96	.36	.85	.87
Graph Tokens	Yes	.96	.91	.99	.99

Table 8: Model performance when trained with or without CoT.

As can be seen in Table 8, across all methods using CoT leads to better test performance. In particular, P-Tuning and Graph Tokens see significant performance decreases for the Node Degree task when no CoT is used, despite the fact that the task should be very easy.

LoRA's performance looks quite good even without CoT. To investigate this point further, we also conducted additional experiments with LoRA on our main experiments' graph sizes (graphs containing 20 to 100 nodes and 500 or fewer edges). We find that LoRA without CoT still appears to show very good performance, shown in Table 9.

However, without CoT, LoRA is **not** able to generalize so well to graph structures which are further out of distribution. The lack of generalization is especially apparent when testing on the cycle graphs, shown in Table 10. Here, the model clearly is not able to learn how to generalize to novel graph structures or longer path lengths. On the other hand, our LoRA model trained with CoT continues to show incredibly strong generalization to this graph structure.

Model	CoT?	NC	ND	DFS	BFS
LoRA	No	.99	.99	.96	.95
LoRA	Yes	1.0	1.0	.98	.99

Table 9: LoRA performance with or without CoT, trained and tested on larger graphs

Model	CoT?	DFS	BFS
LoRA	No	.48	.48
LoRA	Yes	.99	.99

Table 10: LoRA performance with or without CoT, tested on cycle graphs.

C Model Parameters and Architecture

For the adapter of our GRPO-A, GRPO-S, and SFT models, we make use of the PEFT library's implementation of LoRA for our experiments, mostly using default parameters. We select the target modules `["q_proj", "k_proj", "v_proj", "o_proj"]`, rank `r=16`, alpha `lora_alpha=32`, and dropout `lora_dropout=0.1`.

C.1 GRPO

For our GRPO-A and GRPO-S models, we use TRL's implementation of the GRPO trainer[3]. For configurations, we set the number of generations for each sample to be 8, and adjusted the total effective batch size to equal 512, as prior work has suggested larger batch sizes improve training stability. We set the loss type to use DAPO (Yu et al., 2025), set `epsilon=0.2` and `epsilon_high=0.28`, and set `scale_rewards=False`.

For GRPO training, we modified the input prompt to output 3 sections. The first, `<setup>`, would have the LLM setup the task and describe how to solve it. The second would have the LLM produce `<think>` and `<step>` content, to walk through the steps to solve the task one by

[3]https://huggingface.co/docs/trl/main/en/grpo_trainer

one. Lastly, the final answer would be output in <answer> tags.

C.1.1 Rewards

We used four different rewards for both of our GRPO training methods, where scores for tasks were computed deterministically based on the ground-truth answer and solution traces.

Correct Final Answer: Provide a reward of 1.0 if the final answer is correct and 0 otherwise.

Correct Structure: Provide a reward of 0.2 for each of the xml tags (<setup>, <think>, <step>, <answer>) if they occur in the response at lease once. The range of values for this score is between 0 and 0.8.

Step Structure: Parse through all of the output <step> tags to check that they are the desired structure. If they are, provide a reward of 0.5, and if not (or if there are no <step> tags) provide a reward of -1.0.

For the DFS and BFS tasks, we expect the correct structure to be a tuple representing an edge, like <step>(1, 2)</step>.

For the node count and node degree task, we instruct the model to output a python-style dictionary to accumulate and count, like <step>{'nodeID': 0, 'count': 1}<step>.

This reward only targets the structure of the step, and does not check whether steps are correct.

Correct Steps: This reward checks the solution trace for each task and each step output by the LLM, and checks the correctness of each step.

If a solution requires no steps (e.g. BFS trace for a node with no outgoing edges) and the model outputs no steps, a reward of 1.0 is given, and otherwise the reward is 0.

If a solution requires multiple steps, the number of correct steps is counted. Here, once an incorrect step occurs, we consider all subsequent steps to also be incorrect. The reward given is then normalized, as the number of correct steps divided by the total number of steps output by the model and multiplied by 2. The range of rewards output for this situation is 0 to 2.0.

If the solution has no <step> content, we give a reward of -1.0.

C.1.2 GRPO-A

Training for GRPO-A uses the Correct Final Answer, Correct Structure, Step Structure, Correct Steps rewards as described above.

C.1.3 GRPO-S

Training for GRPO-S uses the same rewards, except for Correct Steps. Here, GRPO-S adjusts the rewards given for the multiple steps by **not** normalizing by the total number of steps. Instead, the count of correct steps is taken, and given an upper bound based on the ground-truth solution (the node count / node degree, or the shortest path length for BFS/DFS traces). Additionally, we give an additional 0.5 reward for each correct step which also has a <think> tag before it.

In this way, GRPO-S provides the model with a much stronger reward signal when it produces solutions with multiple steps. In practice, we found that this reward sometimes teaches the model to go too far with outputting steps, in some cases failing to ever produce <answer> tags or continuing a BFS/DFS trace unnecessarily after the target node was already reached.

C.2 SFT

LoRA adapter is trained using a batch size of 32 for up to 20 epochs using SFT, with early stopping based on loss on the validation set with a patience of 5 epochs. In practice, LoRA often achieved its best validation loss around epoch 10-15. Training is computed only on the "completion" portion of the text, i.e. only the portion of data producing the CoT and answer text.

C.3 P-Tuning

We make use of PEFT's implementation of P-Tuning, again mostly using default parameters provided by the PEFT library. The only specialized parameter we set is the number of virtual tokens, `num_virtual_tokens=100`.

As with SFT, P-Tuning is trained using a batch size of 32 for up to 20 epochs, with early stopping based on loss on the validation set with a patience of 5 epochs. P-Tuning also tended to reach its best validation loss around epoch 10-15.

C.4 Graph Tokens

In the Graph Tokens approach presented by Perozzi et al., the graph encoding model, node or edge aggregation, and tokenization varied for each of task. In our case, we want to explore the abilities of a Graph Token approach using a single model applied to different tasks.

After exploring several graph encoding methods proposed by the original paper, we opted to use the

Relational Transformer (**?**) (RT) as our main graph encoding layer. While applied in a very different setting than our paper, RT showed very strong performance compared to other graph neural network architectures at learning to solve graph reasoning tasks, and additionally demonstrated adaptability to OOD graph sizes. RT also has the benefit of applying attention while maintaining n^2 computational complexity for a graph with n nodes, unlike many other works which require e^2 over the number of edges e in a graph.

We re-implemented RT into `torch` modules, and construct our graph encoding model using 6 RT layers (explored 1 to 12 layers), node embedding sizes of 384 (explored embedding sizes from 64 to 768), hidden sizes of 64 (explored sizes of 32 to 128), 12 attention heads (explored number of attention heads from 2 to 12), and a dropout of 0.3.

Another important element of the Graph Token architecture is our choice of node features to serve as input to the graph encoder. We experimented with three approaches for node features: producing sentence embeddings of node labels, Laplacian position embeddings, and Sinusoidal position embeddings. We found that simply using Sinusoidal position embeddings was the most effective in our experiments.

Lastly, for our graph into to the graph encoder model, we designed our model such that we always input a fixed size graph, regardless of the task at hand, and similarly output a fixed number of graph tokens. In addition to using a fixed graph size enabling better parallelization during training, we found that the model struggled to effectively learn all of our tasks when we allowed the input and output graph sizes to vary. In our main experiments, we set a fixed graph size of 200 nodes, and output 400 total embeddings to serve as our graph tokens (200 node embeddings, and 200 edge embeddings, computed by aggregating the sum of the adjacency matrix output by our graph encoder model). Using a fixed graph size, in practice each problem only changes the adjacency matrix input to the graph encoder.

A high level overview of the graph encoder is shown in Figure 16. The graph encoder takes as input nodes and an adjacency matrix, and similarly outputs updated representations of nodes and an adjaency matrix – inner RT layers take the same input and outputs. A projection layer is used to resize the nodes and adjacency matrix at both the input and output of the encoder.

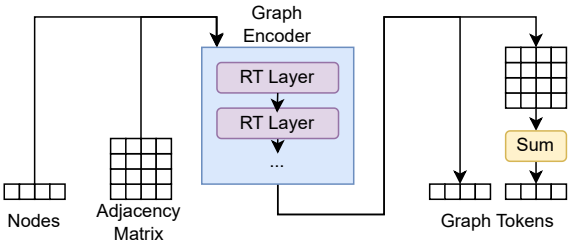

Figure 16: High level overview of the graph encoder.

For training, we used 40 epochs for Graph Tokens, again with early stopping based on validation loss, and used batch sizes of 32. We typically found that 20 epochs were not sufficient for training the Graph Token model, and early stopping often triggered around 30-35 epochs.

C.5 Graph Tokens + Text

Our Graph Tokens + Text method used the exact same training procedure and model architecture as the Graph Tokens method, except that the input question text also included the edge information (i.e., "The edges in G are: ..."). The choice of token count used for this model and P-Tuning make them affect the LLM in very similar ways, with the main difference being that P-Tuning inputs a static set of tokens for every question while Graph Tokens + Text outputs a different set of tokens for each question.

While the Graph Token + Text approach has an additional encoder, we find that the training time per epoch is nearly identical to that of our P-Tuning model. In practice, the computational cost for training and inference is dominated by the LLM. The Graph Token model without the additional text has the fastest training due to its reduction in input tokens.

D Data Statistics

We used 7 random graph generation algorithms and models for our training and test data generation, except for our OOD graph structures. When randomly generating our data, we also applied different weightings to how likely it was to select a given algorithm. The reason for this was that some graph structures are more interesting than others, and many graph structures were more likely to result in data which did not fit our desired distribution for certain tasks. Additionally, in the process of data generation, if a given answer already reached

our answer distribution constraint, we would not include the generated data in our dataset – this resulted in some algorithms producing more data than others.

Table 11 shows the presence of each graph type within our training set, both in terms of absolute count and percentage. Each training task has 3,000 samples each, for a total of 12,000 graphs. We can see that some graph types are much more likely to occur in our data than others – nearly 60% of our training data uses scale-free network graphs. This is caused in part due to some graph structures being much more likely to have certain properties which would lead to more similar answers. For example, Path graphs would always have a node degree of 1, and therefore would no longer be added to the dataset after enough training samples with an answer of 1 have been generated.

Graph Type	Count	Percentage
ER Graph	1,811	30.2%
BA Graph	2,060	34.3%
SBM Graph	1,688	14.1%
SFN Graph	7,132	59.4%
DAG	1,398	23.3%
Star Graph	2,428	20.2%
Path Graph	1,107	18.4%

Table 11: Distribution of graph types in training data.

Here we show some more detailed statistics surrounding our generated datasets. Table 12 shows the node counts in the data, and Table 13 shows the edge counts. Both are presented in terms of the first quartile (Q1), median, mean, third quartile (Q3), and max node or edge counts seen in each dataset.

Data Split	Q1	Median	Mean	Q3	Max
Train	8.0	12.0	12.2	16.0	20
Large Graphs	30.0	43.0	51.4	74.0	100
XL Graphs	145.0	150.0	149.8	155.0	160
Trees	38.0	61.0	60.9	84.0	99
Cycles	40.0	70.0	70.0	100.0	120
OOD Tasks	30.0	43.0	51.2	75.0	100
Edge Labeled	9.0	13.0	12.6	16.0	20

Table 12: Overview of node count statistics for training and test datasets.

Additionally, we present information surrounding trace lengths for the DFS task for each of our datasets in Table 14. Trace lengths are computed for reachability questions regardless of whether the nodes are reachable, as the trace length indicates how many steps the CoT needs to search before determining the correct solution.

Data Split	Q1	Median	Mean	Q3	Max
Train	11.0	20.0	35.7	45.0	300
Large Graphs	64.0	118.0	158.1	223.0	500
XL Graphs	152.0	221.0	208.7	243.0	500
Trees	37.0	60.0	59.9	83.0	98
Cycles	40.0	70.0	70.0	100.0	120
OOD Tasks	57.0	95.0	136.0	167.0	500
Edge Labeled	12.0	25.0	45.8	65.0	300

Table 13: Overview of edge count statistics for training and test datasets.

Data Split	Q1	Median	Mean	Q3	Max
Train	1.0	3.0	4.2	6.0	18
Large Graphs	5.0	11.0	14.6	19.0	77
XL Graphs	8.0	20.0	31.2	49.0	139
Trees	1.0	8.0	15.4	23.0	90
Cycles	20.0	35.0	35.0	50.0	60
Edge Labeled	1.0	3.0	4.2	6.0	17

Table 14: DFS task trace length statistics for each data split.

Several key observations may be made which can provide further insight into our experimental results surrounding DFS. In particular, the cycle graphs tend to have much larger traces on average due to the very deliberate manner in which they were constructed. This likely leads to exposing the models to more test cases with traces that were poorly represented during training. In contrast, even for our extra large graph sizes or test data which aimed to produce longer answers, the median trace lengths were only 20.0.

E Graph Tokens preliminary experiments

This appendix provides a rough overview of some initial experimentation we did to determine a suitable configuration for our Graph Tokens models in our main experiments.

Method	Node Count	Node Degree	DFS	BFS
Random	.100	.150	.500	.500
No CoT	.958	.364	.848	.866
With CoT	.956	.914	.986	.990

Table 15: Comparison of graph token with/without CoT, 5-20 node graphs

Table 15 shows performance differences when we are training the model to output CoT or no CoT, and using CoT shows a clear improvement in performance, especially for the Node Degree task (despite its simplicity). Like prior work, we found that the graph tokens alone struggled to learn the local task of node degree, and even including

CoT the performance of this task hovered near 35% accuracy when trained and tested alone. However, performing multi-task training – with four different tasks and different types of CoT – the model was able to also learn to solve the Node Degree task correctly. This suggest an important difference compared to prior work such as Perozzi et al. (2024), which indicated limitations of graph tokens with certain encoding strategies to effectively learn some tasks when trained on a single task and without producing a CoT as its output.

Method	Node Count	Node Degree	DFS	BFS
Random	.100	.150	.500	.500
2N Tokens	.992	.466	.862	.896
30 Tokens	.956	.914	.986	.990
200 Tokens	.996	.862	.984	.990

Table 16: Comparison of graph token with/without fixed token size, 5-20 node graphs

Table 16 indicates performance when the graph is trained and tested with a fixed number of nodes in the graph versus using a variable number of nodes. When not using fixed token size, the number of graph tokens is 2N where N is the number of nodes in the graph. Using fixed token size limits the extensibility of the method, but appears to help significantly with tasks like node degree (note that results in Table 15 were produced using a fixed token count of 30 as well). Using only 30 fixed tokens vs 200 seems to have some effect on performance, but it isn't strictly positive or negative. This indicates that extra unused tokens do not hinder the ability of the graph token approach – in practice it is likely that the extra tokens simply act in a same manner as tokens learned in the P-tuning approach.

When a fixed number of tokens are being used, the input to the graph encoder (and subsequent output/input to LLM) treats the graph as always having a fixed number of nodes, so different input graphs essentially just are changing the adjacency matrix.

Method	N. Count	N. Degree	DFS	BFS
Random	.100	.150	.500	.500
With Labels	.998	.926	.976	.992
Without Labels	.956	.914	.986	.990

Table 17: Comparison of including node label text in the prompt, 5-20 node graphs

Table 17 shows performance comparing when the model is trained/tested while including node labels in the prompt v.s. not including labels. When labels are not included, questions are all phrased to ask about nodes by ID (e.g. "what is the degree of node 7?"), and when including labels we use randomized node names. Including node labels in the prompt appears to somewhat improve performance – this also is useful to allow us to compare graph tokens against other approaches in a more comparable manner, and also allows for an alternative approach to introduce semantics into the graph nodes without having to devise a complex encoding mechanism. This setting uses significantly more tokens than when no labels associated with the ndoes are in the prompt, but it appears to somehow change what the graph encoder learns to encode and output as graph tokens.

Method	N. Count	N. Degree	DFS	BFS
Random	.100	.150	.500	.500
With Labels	.428	.410	.720	.710
Without Labels	.034	.420	.764	.672

Table 18: Comparison of including node label text in the prompt, on OOD graph sizes containing 20-30 nodes (trained on 5-20 node graphs)

Table 18 compares when we include labels in the prompt text vs not, on OOD graph sizes (trained on 5-20 node graphs, tested on 20-30 node graphs). Including labels in the prompt appears to make a significant difference in out of distribution performance for the node count task. As the base training data in this case only trains to output answers for node counts between 5 and 20, poor performance has been seen quite often. However, what the graph encoder learns to encode when labels are in the prompt somehow affects how well it generalizes to output numbers that were never seen during training.

Given the results of these explorations, the main experiments using graph tokens were conducted using a fixed node count, and node labels were included in the prompt.

Note that some of the final performance metrics seen by our model on 5-20 node graphs in the main experiments differ from those seen in these preliminary experiments due to the use of a different model (main experiments use Phi 4 mini, while these preliminary experiments used Granite3-2B-instruct) as well as minor differences in training data. In particular, our preliminary experiments shows that our single graph token model could reach over 90% accuracy on the Node Degree task, while our main experiments show graph tokens achieving only 25% accuracy even on in-distribution graph sizes. Interestingly, we often saw accuracy on the node degree

task cap out at $\tilde{3}0\%$ across a variety of different configurations. This further suggests how Graph Tokens can be quite brittle, as it was unclear exactly what elements of training procedure, hyperparameters, or prompt/CoT details lead to differences in performance.

F Detailed Result Tables for Phi4

The following tables show our detailed test results when using Phi 4 (14B parameters) on our main set of test tasks and graphs. For these experiments, we focus only on GRPO-A, GRPO-S, and SFT experiments, given that these models shows the best performance when using Phi 4 mini.

Table 19 shows results for in-distribution tasks over varying graph sizes. The 0-shot performance of the baseline model is significantly stronger than that of Phi 4 mini, which in turn leads to the relative performance gains being smaller. It is also worth noting that GRPO-S's extremely poor performance on extra large graphs is a result of the output never producing `<answer>` tags, leading to the lack of answer being treated as incorrect. Based on manually inspecting partial outputs, many predictions by the GRPO-S model appeared to be on a reasonable path to producing the correct answer, but for Phi 4 this variation of the reward function lead to excessively verbose outputs.

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
In Distribution (small) Graphs				
0-Shot Prompt	.998	.964	.906	.874
GRPO-A	1.00	.988	1.00	.998
GRPO-S	.994	.830	.754	.790
SFT	1.00	1.00	.998	1.00
Out of Distribution (larger) Graphs				
0-Shot Prompt	.970	.770	.686	.632
GRPO-A	**.974**	.958	.768	.808
GRPO-S	.914	.588	.200	.224
SFT	.680	**.998**	**.980**	**.992**
Out of Distribution (extra large) Graphs				
0-Shot Prompt	.974	.738	.620	.642
GRPO-A	.984	.942	.688	.746
GRPO-S	.000	.562	.156	.122
SFT	.468	.972	.964	.916

Table 19: Training Phi 4 on 5-20 node graphs and testing on graphs with 20-100 nodes (larger), and 140-160 nodes (extra large).

Similarly, Tables 20 and 21 show results for out of distribution graph structures on Trees and Cycle graphs. We once again see that GRPO-S did not successfully improve model performance here, while SFT and GRPO-A continue to show similar improvement to the Larger graph results. Of note is SFT's very strong performance on DFS and BFS, showing over 97% accuracy on the cycle graphs.

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
0-Shot Prompt	1.00	.910	.852	.802
GRPO-A	.942	.996	.820	.798
GRPO-S	.866	.356	.330	.364
SFT	.556	1.00	.996	1.00

Table 20: Test results over graphs shaped like trees with 20-100 nodes.

Method	DFS	BFS
Random	.500	.500
0-Shot Prompt	.473	.273
GRPO-A	.768	.743
GRPO-S	.132	.075
SFT	.975	.989

Table 21: Test results over graphs shaped like cycles with 20-120 nodes.

Table 22 show test results on OOD tasks, applying the trained Phi 4 models with no additional training and in a 0-shot fashion. We once again see that the baseline model has very strong performance, but GRPO-A and SFT demonstrate a large improvement for the shortest path task. However, we do once again observe some loss in performance, e.g. SFT leads to reduced performance on the cycle check task – however, unlike Phi 4 mini, SFT now has better performance than random chance.

Method	EE	SP	EC	CC
Random	.500	.150	.050	.500
0-Shot Prompt	.962	.244	**.216**	**.910**
GRPO-A	.794	.702	.084	.842
GRPO-S	.924	.246	.032	.388
SFT	**.974**	**.758**	.208	.756

Table 22: Test results on unseen tasks.

Lastly, Table 23 shows test results of Phi 4 on our edge labeled graphs. Here we are finally able to observe some cases where GRPO-S shows decent performance, thanks to test graphs being only 5-20 nodes. However, we observe that the strength of the baseline model without training is superior to many of the trained results for this test.

G Model Transfer Across LLMs

While SFT with LoRA convincingly showed the best performance across in-distribution tasks in generalizability, we were also interested in how an architecture like Graph Tokens might have ben-

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
Tasks only on target edge type				
0-Shot Prompt	-	**.964**	**.976**	**.922**
GRPO-A	-	.962	.702	.696
GRPO-S	-	.654	.752	.736
SFT	-	.962	.852	.820
Tasks on any edge type				
0-Shot Prompt	**.878**	**.846**	.788	.784
GRPO-A	.952	.516	.732	.748
GRPO-S	.962	.560	.804	.858
SFT	**.998**	.486	**.950**	**.938**

Table 23: Test results over graph with edge labels.

efits in that it is detached from the LLM architecture. Consider, for example, that existing work which combines image encoders with LLMs often start from large, pretrained image encoding models. The benefit of such an approach is that an encoding model trained separately can be plugged in together with an LLM. In our case, we did not have such a pretrained graph encoder, but the final result of our training is a graph encoding model which is suitable for making an LLM better at solving graph tasks (albeit with some caveats on its OOD performance).

Our exploration then aims to investigate whether our Graph Token model can be "portable" across different LLMs – i.e., train our Graph Token model on one LLM and apply it to another. If such a transfer of a learned Graph Token model can be successful, it would provide more merit to the approach: LoRA may still be the most effective model when applied to fully train a particular LLM, but Graph Tokens might still provide benefits in terms of training time or scaling.

We conduct a small set of experiments on two ideas. The first is to investigate whether we can train a Graph Token model on a small LLM and transfer it to a larger LLM, where both LLMs share the same model architecture. Here, we use the Granite 3.3 family of LLMs for our experiments, training our Graph Token model using the 2B[4] model and then trying to transfer it to the 8B[5] model. The experimental procedure is to first train our Graph Token on the Granite3.3 2B model, similarly to our other experiments. Then we would freeze the weights of our graph encoder, and train only the final projection layer togher with Granite3.3 8B. Our intuition here is that by training on a smaller model, we can cut down significantly on computational cost of training our Graph Token's encoder, and then continuing training on only the final projection layer can be completed with much less training.

Model	Method	Epochs	DFS	BFS
Granite3.3 2B	Train	40	.996	.980
Granite3.3 8B	Train	40	.986	.966
Granite3.3 8B	Transfer	1	.966	.998

Table 24: Transfer experiment of trained Graph Token model, where the graph encoder was either trained from scratch (first 2 rows) or trained on a 2 billion parameter model and transferred to a 8 billion parameter model with 1 additional epoch of training (last row).

Table 24 shows experimental results, where we compare model performance when we train a Graph Token model from scratch on Granite3.3 2B and Granite3.3 8B. Both models reach nearly identical performance when the graph encoder is fully trained. We then compare the performance of transferred our Graph Token model – trained for 40 epochs on Granite 2B – to the Granite 8B model, leaving the graph encoder weights frozen and training only the projection layer for 1 epoch. We can see that this simple transfer approach enables the Graph Token model to be applied to Granite 8B, achieving nearly identical performance without the need for extensive training. Even beyond just the difference in training epochs, Granite 8B also requires significantly more computational resources to train than Granite 2B, which makes the potential benefits of such a model transfer even better.

Our second experiment aims to investigate whether we can train a Graph Token model on one LLM and transfer to another LLM, where the two LLMs have different model architectures. This time, we perform our transfer experiment by training a Graph Token model using Granite3.3 2B and seeing if it can be transferred to the Phi4-Mini-Instruct model, which we use throughout our main experiments. These results can be seen in Table 25. We see that here, the transfer is not quite as successful as with the transfer among LLMs of the same model family. We do still observe that Phi4 can achieve 95% accuracy on the BFS task, but the DFS performance is clearly weaker. Transfer among different model architectures remains an interesting direction for future work.

[4]https://huggingface.co/ibm-granite/granite-3.3-2b-instruct

[5]https://huggingface.co/ibm-granite/granite-3.3-8b-instruct

Model	Method	Epochs	DFS	BFS
Granite3.3 2B	Train	40	.968	.986
Phi4 Mini	Train	40	.988	.990
Phi4 Mini	Transfer	1	.736	.958

Table 25: Transfer experiment of trained Graph Token model, trained on a 2 billion parameter Granite model and transferred to a 4 billion parameter Phi4 model.

H SFT Experiments with Larger Models

While our primary experiments focus on Phi4 mini, it is reasonable to wonder whether larger models, or different model architectures, may change the story. In Appendix F we present full evaluation results on Phi 4, which increases the parameter count from Phi4 mini's 4 billion to 14 billion, and we observe strong performance of SFT.

One apparent advantage of SFT was that it generalized well to larger graphs for tasks where the answer was in distribution – in particular, DFS and BFS. In this appendix we add some additional experiments to explore how far out models can generalize performance on these tasks, as well as explore how training with larger graphs might affect such performance.

For these experiments we only focus on training LoRA adapters using SFT, but now we perform training using graphs containing 20 to 100 nodes. From the Qwen3 family of models (**?**) we train a Qwen3-4B model and Qwen3-32B model, and additionally train Phi4 models (the full Phi4 model as well as Phi4 mini).

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
Phi4-mini (4B)	.758	1.00	.992	.994
Qwen3-4B	1.00	.996	.974	.966
Phi4 (14B)	1.00	1.00	.990	.992
Qwen3-32B	1.00	.998	.982	.966

Table 26: Results over graphs with 140-160 nodes, after training using graphs with 20-100 nodes.

In-distribution task performance remains strong across all models, but we do see some minor differences in performance on larger graphs. Table 26 shows evaluation results for our 4 in-distribution tasks over larger graphs than those seen during training, where each model shown in the table is using its trained LoRA adapter. The Qwen3 models both appear to not generalize as well as the Phi4 models on the DFS and BFS tasks – notably, simply using a larger model does not address this gap, with the Qwen3-32B model performing worse than Phi4-mini on these two tasks despite being significantly larger and newer. The node count task, on the other hand, shows how other models can demonstrate better generalization than Phi4-mini, but even Qwen3-4B is capable of reaching 100% accuracy on this task and therefore we do not see any clear benefit from model size alone.

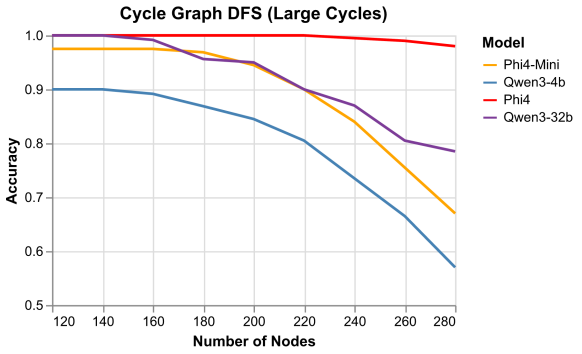

Figure 17: DFS accuracy v.s. graph size on cycle graphs.

We also observe similar patterns when exploring the generalization ability of the models to graphs with both OOD size and shape, as in our cycle graph experiments. Figure 17 shows the performance of respective models at performing BFS over cycle graphs. We once again can see that, regardless of model size, performance gradually decreases as the graph size and DFS traversal length become further out of the training distribution. All models show a similar pattern, where after some critical point the model starts to become unable to accuracy perform BFS, either by hallucinating edges or missing existing edges – even Phi4, which shows the strongest ability to generalize, can be seen starting to decrease in accuracy by the end of our test cases.

Method	EE	SP	EC	CC
Random	.500	.150	.050	.500
Phi4-mini (4B)	.870	.192	.122	.348
Qwen3-4B	.896	.236	.082	.416
Phi4 (14B)	.904	.630	.100	.434
Qwen3-32B	.994	.740	.142	.608

Table 27: Results on OOD tasks, using larger models.

Table 27 shows the performance of each models after training on our 4 training tasks and testing on out-of-distribution tasks in a 0-shot manner, similar as to results in Table 4 of our main experiments (however, in these experiments our training graphs also contain 20-100 nodes). Here, we see some slight improvement in generalization when using larger models as well as larger training graphs –

in particular, the shortest path (SP) task, which essentially just uses BFS and outputs a list rather than Yes/No, shows much stronger performance in the larger models. However, the models still struggle to generalize to the cycle check (CC) tasks. The edge existing (EE) task, which is very simple and quite similar to tasks that were seen during training, can be solved very well by the Qwen3-32B model. Curiously the Phi4 model actually appears to perform worse on OOD task in these experiments, where the models were trained using 20-100 node graphs, compared to the results in Table 22, where the model was trained on 5-20 node graphs – perhaps the fact that larger graphs were further out of distribution somehow helped the model to move out of its over-fit outputs from the BFS/DFS tasks.

Therefore, while increasing model does does show some improvement in generalization, the extent of this generalization is still quite unsatisfactory. Additionally, while larger models of the same architecture appear to generally improve performance, it does not appear to be the case that simply increasing size always improves performance when comparing different architectures; Phi4 (14B) shows much better generalization to larger graphs and longer paths for DFS compared to Qwen3-32B, while Qwen3-32B shows slightly better generalization to new tasks than Phi4.

I Experimentation using 20-100 Node Graphs and SFT-based Methods

This appendix contains a collection of experiments which were used during earlier stages of exploration into the problem of solving graph tasks using LLMs. Much of this exploration was focused on determining if LLMs could solve such tasks in the first place – much of the experiments therefore focus on SFT-based training methods. A key takeaway that we observed from these results was that training LLMs to solve graph tasks *and* remain generalizable was quite difficult.

The experiments in this section used Granite3.3 2B as the LLM of choice, given that it showed good performance while still being quite small and relatively easy to train. Additionally, training graphs used in these experiments contained 20-100 nodes, while our main paper experiments focus on training over 5-20 node graphs and generalizing to bigger graphs. Lastly, this appendix's experiments do not explore GRPO-style training. Note that the results for several tasks shown in this appendix may appear to differ from those shown in the main paper content, primarily due to these differences in LLM, train, and test setup.

I.1 Can LLMs Learn Graph Tasks?

LLMs are able to learn to solve fundamental graph tasks quite well using SFT, as can be seen in Table 28. In this table and subsequent results, Random indicates the accuracy of predicting the most common answer class in the test data and 3-Shot Prompt indicates results using the base LLM with a 3-shot prompting strategy. We observe that all tasks are learned very effectively by all models for in-distribution graph sizes, with the exception of Graph Tokens on the Node Degree task.

Method	NC	ND	DFS	BFS
Random	.100	.150	.500	.500
In Distribution (20-100 node) Graphs				
3-Shot Prompt	.492	.290	.382	.202
LoRA	**1.00**	**1.00**	**.982**	**.996**
P-Tuning	.998	.996	.978	.988
Graph Tokens	.990	.762	.958	.972
Graph Tokens + Text	**1.00**	.992	.978	.988
Out of Distribution (140-160 node) Graphs				
3-Shot Prompt	.366	.212	.340	.330
LoRA	**.758**	**1.00**	**.992**	**.994**
P-Tuning	.662	.948	.970	.958
Graph Tokens	.004	.158	.254	.270
Graph Tokens + Text	.478	.874	.852	.762

Table 28: Results for test cases compared by graph size.

I.2 Generalization of Graph Size and Structure

Broadly, we observe that using LoRA together with multi-task training and instruction tuning, LLMs can learn to solve graph tasks and generalize across graph sizes and structures. Other approaches show greater performance degradation when applied different structures, especially cycle graphs. We also find that using Graph Tokens alone is particularly brittle when applied out of the training distributions, or out of training tasks.

I.2.1 OOD Graph Size

Results for OOD graph sizes can be seen in the lower half of Table 28. Graph Tokens' performance significantly drops in this setting. We expect that a strong contributing factor to this is the use of a fixed node count for Graph Tokens during training, where nodes past the first 100 would never have edges during training. Graph Tokens + Text remains somewhat more robust, but we observe that

its performance is worse than P-Tuning. All models show some decrease in performance on the node count task as all numeric answers for NC would be completely unseen during training over the 20-100 node graphs.

I.2.2 OOD Lengths

For our OOD Lengths test set, we find that out of distribution answer lengths for BFS and DFS did not significantly affect outcomes of the model, with the exception of tests which investigated longer trace lengths. When applied to test cases with longer trace lengths, LoRA's accuracy remained above .990 for all tests, but the DFS performance of P-Tuning decreased from .978 to .830, Graph Tokens decreased from .958 to .870, and Graph Tokens + Text decreased from .978 to .778 – similar decreases were seen for BFS. A likely explanation for this decrease is that the search trace required to determine when nodes were *not* reachable tended to be quite short in our training data – this test introduces many cases where long traces are needed to determine that nodes are not reachable. Full result tables for this set of experiments can be found in Appendix 7.

I.2.3 OOD Graph Structures

Table 29 shows model performance for our two OOD graph structures, trees and cycle graphs. For the tree graphs, Graph Tokens once again shows some larger decrease in performance compared to the other methods. On the cycle graphs, LoRA continues to show very strong performance while our other models show a larger degree of degradation.

	tree				cycle	
Method	NC	ND	DFS	BFS	DFS	BFS
Random	.100	.150	.500	.500	.500	.500
LoRA	**1.00**	**1.00**	**1.00**	**1.00**	**.998**	**.998**
P-Tuning	.998	.982	.966	.996	.927	.882
G.Tok	.998	.414	.898	.810	.700	.786
G.Tok+Text	**1.00**	.986	.958	.976	.867	.791

Table 29: Results for OOD graph structures. G.Tok refers to experiments using Graph Tokens, and G.Tok+Text refers to Graph Tokens + Text experiments.

Figure 18 highlights how models struggle to generalize to cycle graphs for the DFS task as the graph size and path length increase. Despite all other models having lower performance, LoRA continues to show strong performance over the cycle graphs.

On the right side of Figure 18, we evaluate LoRA on much larger graphs, up to 280 nodes / 140-step DFS traces. Even for a 200 node graph – twice the size seen during training – LoRA shows strong performance. Eventually its accuracy drops closer to random, but our results show significant generalization beyond the training set both in terms of graph size and trace length, where 75% of the training samples had traces with only 19 steps or fewer.

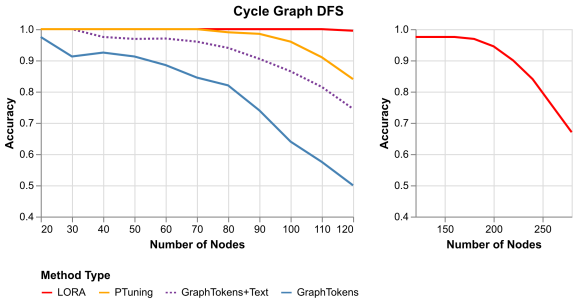

Figure 18: DFS accuracy v.s. graph size on cycle graphs.

I.2.4 Correlation Between Graph Properties and DFS

To inspect whether there are any patterns in the graphs which indicate difficulty of tasks for each trained model, we investigate the DFS task results on larger graphs. Table 30 shows Pearson correlation of some models, where we compute correlation of model accuracy on the DFS task compared to properties of the graph or question – Trace Len. indicates the length of the ground-truth DFS trace to correctly answer the question. We find that for our models which successfully learned to output clear traces (SFT and P-Tuning), trace length has a strong negative correlation – problems which require longer DFS traces are, naturally, harder to answer correctly. GRPO-A, on the other hand, mostly learned to output structural information (like `<think>` and `<answer>` tags) rather than specific steps, and thus its accuracy has little correlation with trace length – the LLM has likely learned to take "short-cuts" in solving the problem by looking directly at the graph.

Method	#Nodes	#Edges	Trace Len.
GRPO-A	-0.148^*	0.135^*	0.019
GRPO-S	0.011	-0.372^*	-0.261^*
SFT	-0.149^*	-0.261^*	-0.516^*
P-Tuning	-0.141^*	-0.260^*	-0.355^*

Table 30: Pearson Correlation between graph properties and DFS accuracy for large graphs. * indicates p-value ≤ 0.05.

I.3 Generalization to New Tasks

Broadly, all models struggle to directly solve new tasks out-of-the-box, but the underlying reasoning steps they output often appear to be on track (due to formatting issues, it is difficult to assess the accuracy of these programmatically). Models often can still out-perform random and fewshot prompting baselines.

Table 31 shows the results of our OOD tasks. We observe that our models all struggle to generalize to the cycle check task – a key reason for this is that they tend to correctly follow the steps to perform a BFS to find the cycle, but they fail to stop after reaching the origin node which would close the cycle. As a result, they often continue to produce reasoning steps and never reach an answer, leading to performance below chance.

Model	EE	SP	EC	CC
Random	.500	.150	.050	.500
3-Shot Prompt	.678	.288	.030	**.526**
LoRA	.870	.192	**.122**	.348
P-Tuning	.722	.060	.090	.384
Graph Tokens	.688	.098	.098	.354
Graph Tokens + Text	**.896**	**.398**	.090	.388

Table 31: Out of distribution task results. Besides 3-Shot Prompt, models are applied in a 0-shot manner after pretraining on our 4 training tasks.

Table 32 compares the performance of models after performing multi-task training using 30 examples of each task. With only 30 training samples, the models can quickly learn new tasks, and the fine tuned models show better learning than training models from scratch on the same number of examples. EE is an exception, where training from scratch seemed to be comparable to fine tuning, but this task is also the easiest. For the Graph Tokens, we found that 30 samples were not sufficient to train the encoder model from scratch, and fine tuning also was less effective than for LoRA.

I.4 Question Answering Tasks

While Graph Tokens shows some generalization to new graph tasks, we found that their performance collapsed when applied to question answering tasks with unfamiliar inputs. Additionally, Graph Tokens requires a properly formatted graph to serve as input for the encoding, which may not be easily producible for many question answering tasks. As a result, our analysis on these tasks are limited to the LoRA and P-Tuning models.

Model	EE	SP	EC	CC
Random	.500	.150	.050	.500
S LoRA	**.968**	.544	.172	.780
S P-Tuning	.756	.232	.086	.484
S Graph Tokens	.246	.102	.002	.492
S Graph Tokens + Text	.722	.202	.040	.288
FT LoRA	.966	**.720**	**.198**	**.994**
FT P-Tuning	.742	.222	.054	.436
FT Graph Tokens	.904	.250	.070	.694
FT Graph Token + Text	.878	.364	.108	.784

Table 32: Out of distribution task results after training the model on 30 task samples. FT indicates fine-tuning the models after training on our 4 training tasks, S indicates base model trained from scratch only using the new task samples.

I.4.1 ProntoQA

Here are a set of statements:
Every petun is not green. Petuns are gretons. Every greton is not smart. Gretons are not luminous.
Create a graph from these facts. Format it so it looks like this:
Below is a graph of nodes, where x → y, z means the graph has edges (x, y) and (x, z).
<graph>
petun → not green, greton
greton → not smart, not luminous
<graph>
Here are some new statements (PrOntoQA problem):
Every tumpus is not angry. Tumpuses are rompuses. Every numpus is not bright. Rompuses are not luminous... Stella is a yumpus. Create a graph with these facts, in the same format.

Figure 19: Prompt for graphs from PrOntoQA problems

We re-structured the PrOntoQA (Saparov and He, 2023) task into two tasks - 1. `id_graphs` where we manually parsed logical statements into graphs in the format used for training , and 2. `generated` where we prompted the base model to produce a graph that we then used with the trained models for reasoning (see Figure 19 for an example prompt used for graph generation). Crucially, the graph structure in `generated` is *different* from those used in training. Table 33 shows results on PrOntoQA, compared with the case when no training was involved. In a now familiar pattern, both LoRA and P-Tuning demonstrated generalization with LoRA showing much greater generalization than P-Tuning.

Model	Prompt Type	Accuracy
Base	original	.472
Base	id_graphs	.470
Base	generated	.546
LoRA	id_graphs	**.998**
LoRA	generated	.774
P-Tuning	id_graphs	.608
P-Tuning	generated	.580

Table 33: Results for the PrOntoQA task.

I.4.2 LC-QuAD 2.0

Table 34 shows results on LC-QuAD (Dubey et al., 2019), where we only take the subset of Boolean questions from the dataset and sample a 3-hop neighborhood around the main entities to serve as our input graph. In the prompt type, `triples` indicates that the KG subgraph was fed in as triplets, i.e. `(subj, pred, obj)`, into the question prompt. `id_triples` indicates that we converted the triplets to be somewhat similar to our training data, with nodes being given IDs and expressing `pred, obj` pairs as a single node label. We provide accuracy in terms of all boolean questions, simply as "Accuracy", as well as accuracy over 156 questions which did not ask questions about numeric values as "N.N. Accuracy" (non-numeric accuracy).

Model	Prompt Type	Accuracy	N.N. Accuracy
Random	-	.522	.616
Base	triples	.707	.519
Base	id_triples	.507	.263
LoRA	triples	.625	.462
LoRA	id_triples	.621	.641
S LoRA	id_triples	.703	.474
FT LoRA	id_triples	**.747**	**.660**
P-Tuning	triples	.291	.244
P-Tuning	id_triples	.404	.462
S P-Tuning	id_triples	.602	.423
FT P-Tuning	id_triples	.528	.449

Table 34: Results for the LC-QuAD task. FT indicates fine-tuning the models after training on our 4 training tasks, S indicates base model trained from scratch only using the new task samples.

We observe several key elements which appear to make our trained models struggle to generalize to KGQA. First, in addition to the KG size being OOD compared to our training graphs, introducing relation information as triplets is quite different from training where graphs had no edge labels. Second, many boolean questions for KGQA cannot be answered by just following paths or counting nodes, as was done in our training tasks. For example, questions may require checking multiple paths like "Did Pope Paul VI work in both Rome and Munich?", or comparing values like "Is the luminosity of the Alpha Andromedae less than 240?". Third, because our training tasks use integers to indicate node IDs, questions which include numeric values appear to confuse the LLM. To examine the impact of this last case, our "N.N. Accuracy" shows that when we disregard such questions, LoRA has better performance compared to the base model, showing some generalization.

We also explore whether fine tuning on 30 examples can enable better performance. Fine tuning does help to improve our models' ability to understand the KG structure, despite the fact that LC-QuAD is challenging due to the need to go well beyond the graph tasks it was trained on. Additionally, fine tuning our pretrained LoRA shows better performance than training a model from scratch.

I.5 Tradeoffs and Limitations from Initial Exploration

Graph Tokens with no added text can learn many tasks, but they are less effective than methods which rely on graph text and show poorer OOD generalization. Graph Tokens have a major advantage in input token count, especially for graphs with many edges. OOD tasks should be a future focus, as reduced input token count can improve the efficiency of LLMs.

P-Tuning demonstrates that LLMs are capable of using prompt changes to solve graph tasks. In many cases, we saw P-Tuning match or beat the Graph Tokens + Text approach, despite P-Tuning using a static set of tokens.

P-Tuning and Graph Tokens are both more portable across different LLMs. LoRA shows the strongest performance, but it is specialized to the LLM in our experiments; a Graph Token encoder can be trained with one LLM and applied to another, and P-Tuning can be adapted by training a projection layer to change the embedding size of the learned tokens.