GraphMind: Empowering LLMs for Interactive Graph Structure Reasoning with Semi-Formal Language

Anonymous Author(s)

Abstract

Large language models (LLMs) have demonstrated remarkable capabilities in understanding unstructured text and images. However, regarding to graph structure reasoning, LLMs suffer from a severe **interaction disorder** when directly verbalizing the graph using natural language (i. e. informal language): in the *input phase*, LLMs are sensitive to the arbitrary graph prompt and node/edge description order; in the *output phase*, LLMs show insufficient graph reasoning logic and graph structure hallucinations. To solve this problem, we propose an interactive graph structure reasoning framework using semi-formal language, named GraphMind, decoupling the external graph storage from internal structure reasoning. Specifically, we design a semi-formal grammar for LLMs with embedded graph functions, interacting with external table transformed from graph. Secondly, we employ a two-stage LLM instruction tuning to acquire the new grammar and call the graph functions accurately. Finally, we propose a multi-faceted scoring to align the structure reasoning trajectory with an ordered preference. Extensive experiments on 9 structure reasoning benchmarks demonstrate the superiority of proposed GraphMind, outperforming existing state-of-the-art baseline with an average 21.90% improvement. In particular, the performance improvements for Shortest Path, Maximum Triangle Sum, and Topological Sorting task reach 121.6%, 182.8%, and 332.9%, respectively. Additionally, GraphMind shows strong robustness on larger and denser graphs, and can be easily extended to text-attributed graph understanding, e.g., citation networks.

Keywords

Semi-formal language, graph structure reasoning, Graph LLM

ACM Reference Format:

Anonymous Author(s). 2026. GraphMind: Empowering LLMs for Interactive Graph Structure Reasoning with Semi-Formal Language. In . ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large-language models (LLMs) have demonstrated strong understanding capabilities across a variety of domains—including natural language processing [15, 34, 36, 40, 45, 46] and computer vision [11, 28, 29, 32, 35]. In contrast to unstructured text and images, graphs organize information by representing heterogeneous entities and relations. This structural property enables graphs to effectively model complex systems across diverse domains such as social networks [27, 47], transportation systems [25, 48], and knowledge graphs [37, 49]. However, the intrinsic complexity of graph topology, characterized by the unordered nature of nodes and edges, poses unique challenges to the sequential understanding of LLMs. To fully leverage the versatility of LLMs, a straightforward solution is to verbalize the graph with natural language, e.g.

> **The input of LLM:** "Given a directed graph, the nodes range from 0 to 3. Node 0 is connected to node 3. Node 1 is connected to node 2. Node 2 is connected to node 3....."

Existing works that follow this paradigm for graph structure reasoning either employ carefully handcrafted prompt engineering on pretrained LLMs [2, 21, 31, 41], or finetune the models on constructed graph corpus [4, 10, 19, 26]. However, directly verbalizing a graph as plain text encounters severe **interaction disorder** in both input and output phase of LLM processing. ❶ In the *input phase*, the unordered nature of nodes and edges in the graph conflicts with the sequential input of LLMs. LLMs are highly sensitive to variations in the order of node/edge description and node labeling (e.g., numbers or alphabet), leading to contradictory reasoning outcomes. ❷ In the *output phase*, LLMs may show structural hallucinations during reasoning, such as introducing nonexistent nodes/edges, overlooking existing ones, or producing flawed logical inferences, including connectivity, structural matching, and counting errors. Therefore, this suggests the following

Question: *How to mitigate the interaction disorder to ensure accurate graph structure input and logically coherent LLM output?*

To solve this problem, we propose to decouple the external graph storage from internal LLM structure reasoning with a semi-formal language, named GraphMind. By teaching LLM to learn a new semi-formal graph grammar, the input graph is transformed into an externally stored data structure; GraphMind interacts with it through graph function calling (i.e., formal language), while logically reasoning via a mix of natural language (i.e., informal language). Specifically, we first design a hybrid semi-formal language that embeds five distinct graph functions interacting with graph-transformed external table. Secondly, we propose a two-stage LLM instruction tuning to teach LLM understanding and correctly calling the graph functions on a comprehensive graph structure reasoning QA pairs across 9 different tasks. Finally, we define three evaluation metrics—Answer Correctness, Function Density, and Function Appropriateness—to score and rank multiple LLM outputs. These scores generate preference and dispreference pairs to align the reasoning trajectory with on ordered preference, which further refines the model's responses to be more coherent and functionally accurate.

The contributions of this paper are summarized as follows:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- We propose a novel semi-formal language reasoning framework to decouple graph data from the LLM's native processing, thereby substantially alleviating the interaction disorder in both LLM input and output phase.
- We design a composite set of preference evaluation metrics guiding LLM structure reasoning trajectory, enforcing greater logical coherence in the LLM's reasoning and alleviating the output disorder.
- GraphMind ranks 1st on the public benchmark over baselines, especially achieving improvements of 121.9% to 332.9% on shortest path, triangle sum, and topological sorting tasks. Meanwhile, GraphMind demonstrates satisfactory generalization on larger and denser graphs and easily extends to text-attributed graphs.

2 Related Work

2.1 Pure Graph Structure Reasoning with LLMs

Pure Graph Structure Reasoning with LLMs explores the capacity of LLMs to solve classical graph-theoretic problems (e.g., cycle detection, shortest path, Hamiltonian circuit) purely through their text-based understanding and generation capabilities, without relying on external algorithmic modules or symbolic solvers. As the boundaries of LLM capabilities are increasingly scrutinized, this line of research has attracted growing attention in both the NLP and GNN communities. Recent efforts investigating Pure Graph Structure Reasoning with LLMs [2, 4, 10, 26, 31, 41, 50] can be broadly categorized into two main directions.

Prompt-based methods leverage only prompt engineering with large language models (e.g., GPT-4) for graph reasoning. Notable examples include Talk Like a Graph [10], which applies nine prompting schemes—such as adjacency-based prompts ("In an undirected graph, (i,j) indicates a connection ...") and expert-style prompts ("You are a graph analyst. Nodes A–I are connected by edges ...")—and NLGraph [41], which employs Chain-of-Thought prompting, Self-Consistency sampling, and Least-to-Most prompting to elicit structured reasoning. A key drawback is that LLMs are highly sensitive to input formatting, often yielding incorrect or fabricated responses.

Finetuning-based methods fine-tune smaller LLMs using human-generated or LLM-generated graph reasoning data. For instance, GraphWiz [4] first creates large-scale GPT-4–generated datasets (up to 100-node graphs) and then applies Direct Preference Optimization [33] with simple reward functions for accuracy and brevity. Despite extensive fine-tuning, these models still hallucinate, as their reasoning paths lack sufficient constraints, resulting in inconsistent outputs even on similar problems.

2.2 Text-attributed Graph Understanding

Another growing body of work has recently explored the potential of large language models (LLMs) in the context of text-attributed graphs [3, 5–7, 17, 20, 22, 23, 42]. These approaches generally fall into two main categories: (1) encoding the graph data into textual representations using graph neural networks (GNNs) and then applying LLMs for reasoning and prediction (e.g., GIANT [7], Llaga [5]); and (2) directly processing graph data with LLMs, followed by projecting the results back into the graph space using GNNs for downstream tasks (e.g., TAPE [22]).

These existing strategies typically focus on aligning the semantic spaces of graph structures and natural language, aiming to leverage either LLMs or GNNs for downstream tasks once this alignment is achieved. In contrast, our approach bypasses this alignment stage entirely: GraphMind directly performs reasoning in a semi-formal, natural language-like semantic space, utilizing the generative capabilities of LLMs without requiring explicit mappings between graph and text representations. This allows for a more flexible, interpretable, and generalizable reasoning approach over rich graph-structured data.

3 Preliminary

3.1 Problem Formulation

Definition 1. Pure Graph. We consider a pure graph setting with graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the sets of vertices and edges respectively. In the pure graph, we focus on the structure analysis, and neither nodes nor edges possess any additional attributes (e.g., textual descriptions or semantic labels).

Definition 2. Graph Structure Reasoning. Given a pure graph $\mathcal{G}$, graph structure reasoning aims to analyze the graph topology, such as connectivity, cycle detection, shortest path, subgraph matching. Formally, this involves a computational process where, given a pure graph $\mathcal{G}$ and a query Q, an LLM $\mathcal{M}$ (with parameter Θ) generates an answer $\mathcal{A}$. The query Q can be expressed either as a natural language question or a logical expression, while the output $\mathcal{A}$ may be a vertex subset or a Boolean value. This reasoning process can be represented as $\mathcal{A} = \mathcal{M}(\mathcal{G}, Q; \Theta)$.

3.2 Semi-formal Languages

Semi-formal languages constitute hybrid systems integrating characteristics of both formal and informal (natural) languages. Formal languages are rigorously defined by fixed syntactic and semantic rule sets, exemplified by systems such as SQL, SPARQL. Informal languages exhibit no systematic grammatical constraints, like in everyday speech. Semi-formal languages occupy an intermediate position, retaining controllable organizational features without the absolute rigidity of formal systems. In this study, semi-formal language specifically denotes natural language augmented through embedded graph functions (e. g., FIND_NODE[node_id]). These primitives introduce deterministic grammar structures into inherently flexible expressions, thereby enabling machine-interpretable semantic parsing within context-dependent boundaries.

4 Methodology

In this section, we formally provide the detailed methodology of our proposed GraphMind. In Figure 1, we present the training and inference stage of GraphMind.

4.1 Semi-Formal Graph Grammar Design

In order to decouple external graph storage from the internal structure reasoning of LLMs, we design a novel semi-formal graph grammar. The core idea is to transform the graph into a table format and then design a series of graph functions that the LLMs can flexibly choose from to interact with the table. This approach eliminates the need for complex descriptions and difficult-to-trace reasoning logic when dealing with the graph.

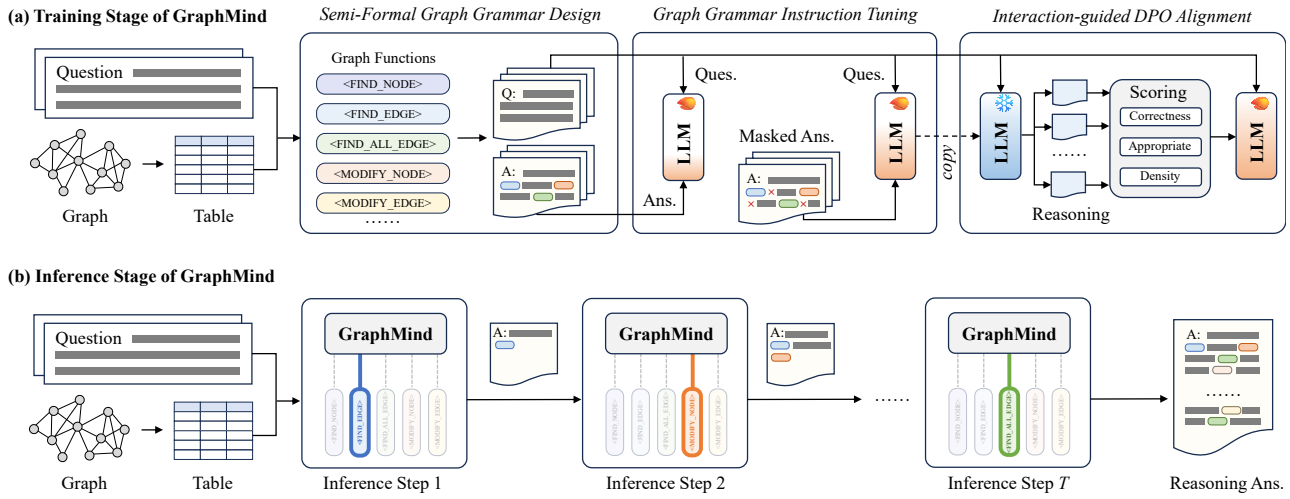
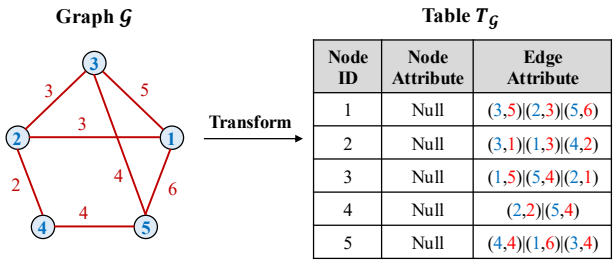


Figure 1: Training and Inference Framework of our proposed GraphMind.



Node ID	Node Attribute	Edge Attribute
1	Null	(3,5)|(2,3)|(5,6)
2	Null	(3,1)|(1,3)|(4,2)
3	Null	(1,5)|(5,4)|(2,1)
4	Null	(2,2)|(5,4)
5	Null	(4,4)|(1,6)|(3,4)

Figure 2: A toy example of graph and its corresponding table format. Red number is the weight of edge, Blue number is the Node ID. Edge attributes are jointed by '|'.

Graph2Table Transformation. Most existing works [4, 26] directly verbalize the graph using natural language. However, despite the permutation-invariant nature of graph nodes and edges, the variation in serialized textual input significantly impacts the output of the LLM [1]. To address this issue, we first propose the Graph2Table transformation, which converts the graph into a table and allows the LLM to access it via functional interfaces instead. Figure 2 provides a toy example for illustration, where each graph $\mathcal{G}$ is transformed to a table $T_{\mathcal{G}}$ with three columns. Each row in the table represents a node in the graph. The first column denotes the node identifier, the second column denotes the node attribute, and the third column lists edge pairs, each consisting of an adjacent node and the corresponding edge properties separated by '|'.

Semi-formal Language Design. To enable interaction with graphs in table format, we design a semi-formal language that combines both natural language and formal graph functions. In our work, we propose only 5 commonly used `GraphFunction` $\mathcal{F} = \{f_j\}_j$ covering various graph structure reasoning tasks as shown in Table 1:

- `FIND_NODE[node_id]`
- `FIND_ALL_EDGES[node_id]`
- `FIND_EDGE[node_id1, node_id2]`
- `MODIFY_NODE[node_id, node-attribute]`
- `MODIFY_EDGE[node_id1, node_id2, edge-attribute]`

Our `GraphFunction` design is relatively simple and exhibits low inter-function similarity. At the same time, it supports both query and modification operations on graph edges, as well as on nodes. These node and edge operations are sufficiently expressive to support the completion of most graph-related tasks. In practice, the `GraphFunction` can be further extended, adapting to different graph types and downstream tasks.

Graph Structure Reasoning QA Pairs Construction. To enable the LLMs to master the proposed semi-formal graph grammar, we construct more than 18,000 QA training pairs $\mathcal{D} = \{\mathcal{Q}, \mathcal{A}\}$ based on the latest GraphWiz [4] dataset. GraphWiz categorizes graph structure reasoning questions into three difficulty levels—Easy, Medium, and Hard—based on algorithmic complexity, covering 9 common graph computation problems. The Easy level includes four types of problems such as cycle detection and connectivity checking; the Medium level includes three types of problems such as shortest path and maximum flow; the Hard level includes Hamiltonian cycle and subgraph matching. The graph sizes in the GraphWiz dataset range from 2–100 nodes, which offers a more comprehensive scale compared to other similar datasets [12, 14, 26, 41, 50]. The specific transformation method consists of two steps. (1) Read a graph $\mathcal{G}$ from the GraphWiz dataset and convert $\mathcal{G}$ into the table format $T_{\mathcal{G}}$; (2) Generate the LLM answer using algorithmic procedures. Invoke the five associated functions in `GraphFunction`. Each time a function is called, include both the corresponding function call and its returned result in the generated answer. The detailed algorithm is shown in Algorithm 1. An illustrative example is provided in A simple case for QA pair construction 4.1. More details of QA pairs construct are shown in Appendix D.

A simple case for QA pair construction.

GraphWiz (Original QA pairs)
Question: Determine whether two nodes are connected in an undirected graph. In an undirected graph, (i,j) means that node i

Algorithm 1 Graph Reasoning QA Pairs Construction

Require: Graph $\mathcal{G}$, natural language questions Q_{NL} in GraphWiz.
Ensure: Semi-formal graph reasoning QA pairs $\mathcal{D} = \{Q, \mathcal{A}\}$.

```
% Step 1: Graph to Table Conversion
T_G ← Convert graph G to table format.
% Step 2: Question Processing
for each question q_i ∈ Q_NL do
    q_i ← Remove the graph description in q_i.
% Step 3: Semi-Formal Answer Generation
    Initialize corresponding answer a_i ← ∅.
    while Algorithm not Finished do
        if operation matches GraphFunction f_j then
            result ← Execute f_j on T_G.
            Update a_i with f_j and result.
        else
            Update a_i correlate to question.
        end if
    end while
    Append (q_i, a_i) in D.
end for
return D.
```

and node j are connected with an undirected edge. Given a graph and a pair of nodes, you need to output Yes or No, indicating whether the node i and node j are connected. The nodes are numbered from 0 to 9, and the edges are: (0, 4) (0, 8) (1, 6) (1, 3) (2, 4) (3, 8) (3, 4) (3, 9) (3, 7) (5, 8) (6, 7). Is node 0 and node 3 connected?
Answer: Node 0 is connected to node 4, node 4 is connected to node 3, so the answer is yes. ### Yes.

GraphMind-base (Ours)
Question: A undirected graph is represented as a table. Node range from 0 to 9. Check if node 0 and node 3 are connected in the given graph table. Answer by Yes or No.
Answer: We use BFS to traverse the graph. If we can reach node 3 from node 0, then they are connected. Traverse from node 0: `FIND_ALL_EDGES[0]` = (4,NUL)|(8,NUL). Traverse node 4 beacuse it's unvisited: `FIND_ALL_EDGES[4]` = (0,NUL)|(2,NUL)|(3,NUL). Node 3 is visited. After walking through the graph, we find that node 0 and node 3 are connected. So the answer is ### Yes.

4.2 Graph Grammar Instruction Tuning

To enable LLMs to learn and master our designed semi-formal language, our training consists of two stages. Stage 1 focuses on graph language acquisition, akin to learning a dialect. Stage 2 emphasizes grammar-centric refinement, serving to enhance the LLM's understanding and command of `GraphFunction`.

Graph Language Acquisition. LLMs are pretrained on vast amounts of unformalized linguistic data (i. e., natural language corpora), which endows them with strong performance in the informal semantic space $\mathcal{I}$. In order to transfer this linguistic competence into our deliberately designed semi-formal semantic space $\mathcal{S}$, we perform full paramete fine-tuning [9, 13] of the pretrained LLM using the bespoke QA pairs we have constructed. This targeted fine-tuning process enables the model to acquire and internalize the conventions and structures of our semi-formal language without altering its underlying architecture. In simple terms, this completes the pattern mapping from $\mathcal{I}$ to $\mathcal{S}$ ($\mathcal{I} \rightarrow \mathcal{S}$).

$$\mathcal{L}^{\text{full}}_{\text{acquisition}}(\Theta) = \sum_{(Q,\mathcal{A})\in\mathcal{D}} -\log P_{\Theta}(\mathcal{A} \mid T_{\mathcal{G}}, Q), \tag{1}$$

where $T_{\mathcal{G}}$ denotes the table transformed from the original graph data, Q and $\mathcal{A}$ denote the question and answer in the QA pair, $\mathcal{D}$ denotes the dataset, Θ denotes the full set of trainable parameters (i. e., the full model).

Grammar-Centric Refinement. During the Graph Language Acquisition stage, the LLM has become accustomed to the semi-formal language combining natural language and `GraphFunction`; however, it frequently misapplies function calls, resulting in incomplete responses and logical errors. To address this issue, we propose a grammar-centric refinement approach that enhances the LLM's understanding of function calls by masking the returned results of `GraphFunction` calling. The core idea is that in the GraphMind framework, the storage of graph structures and the reasoning process of the LLM are fully decoupled. While the results of function calls can assist in interpreting the function's behavior, they may also negatively impact the model's ability to generalize the semantics of the function itself. To mitigate this, we introduce a grammar-centric refinement strategy that masks function call outputs, thereby encouraging the model to focus on the grammar and structural aspects of the function rather than its specific outcomes (Figure 3 illustrates an example of this masking strategy). Based on this masking approach, we further perform LoRA-based fine-tuning on the LLM to adapt it to the grammar-centric reasoning paradigm. Formally, our training objective can be expressed as:

$$\mathcal{L}^{\text{LoRA}}_{\text{refinement}}(\Theta_{\text{LoRA}}) = \sum_{(Q,\mathcal{A})\in\mathcal{D}} -\log P_{\Theta_{\text{LoRA}}}(\mathcal{A} \mid T_{\mathcal{G}}, Q), \tag{2}$$

where Θ_{LoRA} denotes the tunable parameters introduced by LoRA.

4.3 Interaction-Guided DPO Alignment

Direct Preference Optimization (DPO)[33] is a reinforcement fine-tuning method that directly aligns model outputs with human preferences by optimizing a preference-based objective. Existing studies have demonstrated that DPO can significantly improve a model's understanding of linguistic patterns and enhance the logical consistency of its responses[8, 18, 43, 44]. To further enhance the LLM's capability in semi-formal language understanding and make its output process more reasonable, we propose an interaction-guided dpo alignment method. This approach aligns model outputs with three critical reasoning attributes: logical coherence, structural integrity, and factual accuracy. Unlike conventional DPO implementations that focus solely on final answers, our method supervises the entire reasoning trajectory through a multi-faceted scoring mechanism.

(1) **Answer Correctness** (S_{correct}): This metric evaluates the factual correctness of the final conclusions provided by the model.

Table 1: Detail description of proposed five `GraphFunction`.

`GraphFunction` Instruction	Description
`FIND_NODE[node_id]`	Retrieve the information associated with the specified node in the graph.
`FIND_ALL_EDGES[node_id]`	Retrieve all edges incident to the specified node.
`FIND_EDGE[node_id1, node_id2]`	Retrieve the edge information between two specified nodes in the graph.
`MODIFY_NODE[node_id, node-attribute]`	Modify the attribute value of the specified node.
`MODIFY_EDGE[node_id1, node_id2, edge-attribute]`	Modify the attribute value of the edge between the two specified nodes.

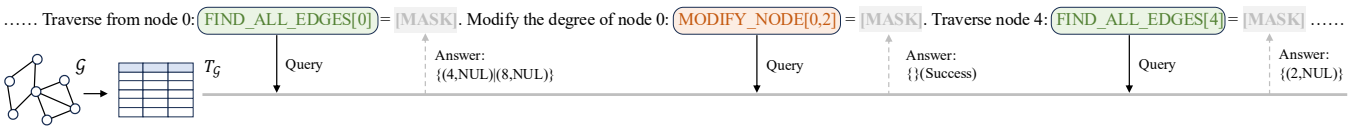


Figure 3: A case illustration of label masking used in grammar-centric refinement.

It is assessed as a binary score, indicating whether the answer is factually accurate or not.

(2) **Function Density** (S_{density}): This metric measures the deviation in the density of function calls from the ground truth. It reflects how the number of invoked functional steps in the model's response compares to that in the reference solution.

(3) **Function Appropriateness** ($S_{\text{appropriate}}$): This metric assesses the appropriateness of function usage in the model's response. It focuses on whether the selected functions are relevant, contextually appropriate, and proportionally used in accordance with the task requirements.

These metrics combine into a unified preference signal:

$$\text{score} = w_1 S_{\text{correct}} + w_2 S_{\text{density}} + w_3 S_{\text{appropriate}}, \tag{3}$$

with normalized factors ($\sum w_i = 1$) ensuring balanced optimization.

The correctness metric targets the factual validity of the model's final answer, enforcing strict binary evaluation as used in many reasoning and factual QA settings. The model is rewarded only when its conclusion aligns with verified ground truth. We define $S_{correct}$ as 1 when the final answer is factually correct, and 0 otherwise.

The density metric aims to ensure that the number of function calls made during the model's output generation is appropriate for producing a well-reasoned inference. It is formally defined as: $S_{\text{density}} = 1 - \alpha|D_{\text{answer}} - D_{\text{gt}}|$, where D_{answer} denotes the function call density in the model's output, and D_{gt} represents the function call density in the reference answer. The density D is computed as the ratio of the total number of function calls to the length of the output: $D = \frac{N_{\text{total}}}{\text{Length}}$. N_{total} is the number of function calls invoked during the reasoning process, and Length is the total number of tokens in the model's final answer. The hyperparameter α is used to map the absolute density difference to a score within the range $[0, 1]$, thereby penalizing deviations from the expected density.

The appropriateness metric quantifies the execution-level accuracy of invoked function calls, defined as the proportion of successfully executed functions out of all function calls made by the model. It directly reflects runtime reliability and correctness of usage: $S_{\text{appropriate}} = \frac{N_{\text{success}}}{N_{\text{total}}}$, where N_{success} is the number of successfully executed function calls, and N_{total} is the total number of function calls issued. A higher score indicates more reliable and contextually accurate use of functional components.

Preference pairs (y_p, y_{dp}), representing respectively the *preferred* and *dispreferred* responses for a given input x, are constructed based on model scores as follows: (1) Outputs with composite scores exceeding a predefined threshold τ are selected as preferred responses (y_p); (2) Dispreferred responses (y_{dp}) are sampled from outputs with scores below τ; (3) When thresholding is not applicable due to score distribution, direct pairwise ranking based on composite scores is used instead. This preference signal is then used to supervise learning via the DPO objective:

$$\mathcal{L}_{\text{DPO}} = -\mathbb{E}_{(q,y_p,y_{dp})}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_p|q)}{\pi_{\text{ref}}(y_p|q)} - \beta\log\frac{\pi_\theta(y_{dp}|q)}{\pi_{\text{ref}}(y_{dp}|q)}\right)\right], \tag{4}$$

where π_θ denotes the current policy, π_{ref} is a frozen reference policy, σ is sigmoid function, and β controls the sharpness of preference.

5 Experiments

In this section, we empirically evaluate the effectiveness and generalizability of GraphMind via extensive experiments by addressing the following research questions (**RQ**s):

- **RQ1:** How well does GraphMind perform on graph structure understanding tasks?
- **RQ2:** How robust is GraphMind when applied to larger and denser graphs?
- **RQ3:** How well does GraphMind deal with interactive disorder in both input and output phase?
- **RQ4:** What are the inference efficiency and model perplexity (PPL) of GraphMind?
- **RQ5:** How to extend GraphMind to handle graph with textual information, e.g., citation network?

5.1 Experiment Setting

Firstly, we present the experiment setting in detail. To support the reproducibility of the results, we have released our code[1]. All evaluated models are implemented on a server with 2 A6000 GPUs.

[1]https://anonymous.4open.science/r/GraphMind-00B8

Datasets. Our training and test sets are derived directly from the corresponding splits of the GraphWiz [4] dataset. In total, GraphWiz comprises 18,125 examples, and its test split is organized into nine distinct graph-structure understanding tasks, each containing 400 instances. For a detailed description please refer to Appendix A.

Baselines. For graph structure understanding tasks, we adopt four Large LLMs API–based baselines: GPT-4 in a zero-shot setting, GPT-3.5 in a two-shot setting, GPT-4 in a two-shot setting. We also include NLGraph [41], which directly leverages prompt engineering with GPT models to perform graph reasoning without any additional fine-tuning. In addition, we compare against four representative methods specifically designed for graph structure reasoning—Naive-SFT model, GraphWiz[4], GraphInstruct[26], and Talk like a Graph [10]—which fine-tune smaller LLMs using task-specific data. More detailed descriptions of each baseline configuration and prompt design are provided in Appendix B.

Evaluation Metrics. We evaluate model performance using answer accuracy, which is the proportion of instances where the LLM's response contains the ground-truth answer.

Implementation Details. We refer to our model without DPO alignment as GraphMind-base, and the one after DPO alignment as GraphMind-DPO. We adopt two widely used 7B-parameter language models—LLaMA-2-7B [39] and Mistral-7B [38]—as our backbones. All models are implemented in PyTorch [30] and optimized with AdamW [24]. In the Graph Language Acquisition stage, we perform full-model fine-tuning with a learning rate of 2×10^{-5} and a batch size of 2. During the Grammar-Centric Refinement stage, we apply LoRA fine-tuning with a learning rate of 1×10^{-5} and a LoRA rank of 32. Finally, in the Interaction-Guided DPO Alignment stage, we continue with a learning rate of 1×10^{-5} and set the DPO trade-off parameter β to 0.3.

5.2 Performance on Graph Structure Understanding Tasks (RQ1)

We evaluate GraphMind's ability to understand and reason over graph structures by comparing it with existing graph reasoning baselines. The experiments are repeated three times, and we report the averaged results in Table 2, Table 9 and Figure 4.

GraphMind consistently outperforms most baseline methods across tasks, and rank 1st on average across the 9 tasks. As shown in Table 2, our approach achieves a substantial 62.00% improvement over GPT-4. Moreover, relative to the current state-of-the-art baseline GraphWiz, we observe an overall enhancements of 29.10% and 21.90% on the standard GraphWiz and the GraphWiz-DPO variant, respectively. The performance of approaches relying solely on prompt engineering is unsatisfactory on graph-structure reasoning tasks, falling short of the results obtained by fine-tuning smaller LLM. This gap highlights the necessity of advanced models designs tailored specifically for graph-structure reasoning.

Notably, GraphMind achieves significant performance improvements on several specific tasks through semi-formal structural reasoning. For topology problems, GraphMind-base yields a 332.9% gain over the SOTA; after DPO fine-tuning, GraphMind-DPO surpasses GraphWiz-DPO on the shortest-path and triangle tasks by 121.6% and 182.8%, respectively. Figure 4 illustrates the performance of GraphMind-base and GraphMind-DPO across difficulty levels. On Easy problems, we achieve improvements of 39.4% and

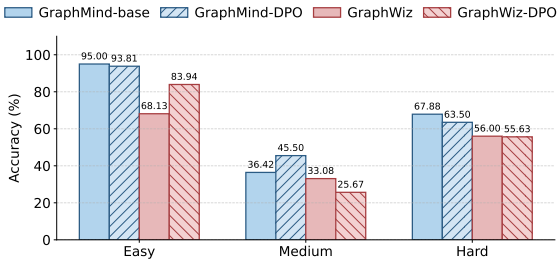


Figure 4: Performance comparison between GraphMind and GraphWiz over different task diffculties.

11.8% over the corresponding GraphWiz models; on Medium problems, gains of 10.1% and 77.2%; and on Hard problems, increases of 21.2% and 14.1%. Table 9 reports results using LLaMA-2-7B as the backbone. Here, GraphMind still outperforms the GraphWiz baseline by 23.4% on the standard fine-tuned model and by 10.2% on the DPO-fine-tuned model, demonstrating the robustness of our approach across different backbone architectures.

Admittedly, in the subgraph-matching task, GraphMind underperforms slightly relative to GraphWiz. We observe that, when LLMs are fine-tuned directly, they often produce outputs with flawed reasoning, revealing a clear hallucination issue. This problem is especially pronounced in GraphWiz after DPO training—its subgraph-matching performance drops significantly from 85.50% to 48.50%—further confirming that large language models struggle to produce reliable solutions for NP-hard problems requiring global topological understanding. This finding highlights the inherent limitations of current LLM-based graph-learning approaches when tackling complex graph-theoretic challenges.

5.3 Performance Robustness on Larger and Denser Graphs (RQ2)

Existing graph reasoning datasets mostly consist of small-scale graphs [12, 14, 26, 41, 50], with node counts not exceeding 40. GraphWiz was the first to extend graph size to 100 nodes; in our work, we further increase the scale to graphs containing 150–250 nodes to assess model performance on large graphs. Moreover, in all existing datasets, edge density (defined as $\frac{2|E|}{|V|(|V|-1)}$) is approximately 0.1. We elevate this density to 0.4–0.7 to evaluate LLMs' ability to perform graph reasoning on dense graphs.

For large graphs, we select the connectivity task, since as node count increases, model output disorder rises dramatically, making connectivity queries substantially more challenging than in small graphs. For dense graphs, we focus on the shortest-path problem, as the number of potential paths grows significantly with density, reducing the likelihood of random correct predictions.

Figure 5 compares the performance of GraphMind and GraphWiz on both large and dense graphs. On large graphs, GraphMind-base and GraphMind-DPO outperform their GraphWiz counterparts by 633.3% and 283.5%, respectively; on dense graphs, they achieve gains of 21.9% and 35.5%, respectively. These results demonstrate that GraphMind is more adaptable to a variety of graph

Table 2: Algorithm performance on different graph problems, where bold indicates the best performance and underlined indicates the second-best. The blue number represents the ranking of different methods based on average performance.

Algorithms	Easy				Medium			Hard		Average (Rank)
	cycle	connect	bipartite	topology	shortest	triangle	flow	hamilton	subgraph	
GPT-3.5 (2-shot)	51.25	43.75	70.75	4.50	3.50	17.25	8.50	54.25	43.00	32.97 (10)
GPT-4 (zero-shot)	38.75	17.00	65.25	5.00	9.25	5.75	3.25	59.25	45.50	27.67 (13)
GPT-4 (2-shot)	52.50	62.75	74.25	25.25	18.25	31.00	7.75	**75.75**	46.75	43.81 (6)
NLGraph	67.00	77.00	68.00	16.00	9.00	6.00	16.00	44.00	54.00	39.67 (7)
Naive SFT	73.75	83.50	78.50	1.00	23.00	47.00	28.75	31.75	41.25	46.56 (5)
GraphInstruct	68.00	61.50	61.00	1.50	7.50	0.00	7.00	36.00	53.00	32.83 (11)
GraphInstruct+	57.00	67.00	70.00	8.00	3.00	1.00	5.00	54.00	47.00	34.67 (8)
Talk Like a graph (a.)	55.00	52.00	45.00	4.00	30.00	1.00	19.00	25.00	44.00	30.56 (12)
Talk Like a graph (e.)	46.00	63.00	50.00	25.00	30.00	1.00	19.00	23.00	42.00	33.22 (9)
GraphWiz	92.00	89.50	72.00	19.00	31.25	38.75	29.25	26.50	**85.50**	53.75 (4)
GraphMind-base	**99.00**	99.75	**99.00**	82.25	**35.25**	60.00	14.00	74.25	61.50	69.44 (2)
GraphWiz-DPO	85.50	79.50	85.50	**85.25**	12.50	29.00	35.50	62.75	48.50	58.22 (3)
GraphMind-DPO	98.50	**100.00**	94.25	82.50	**35.25**	**64.75**	**36.50**	72.75	54.25	**70.97** (1)

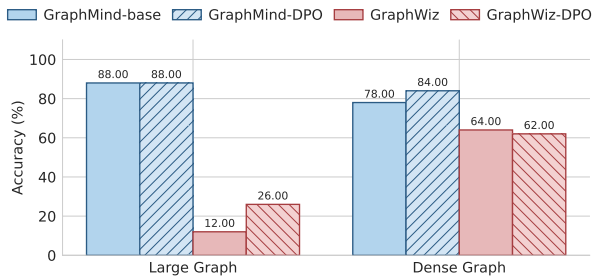


Figure 5: Performance comparison of GraphMind and GraphWiz under large graph and dense graph.

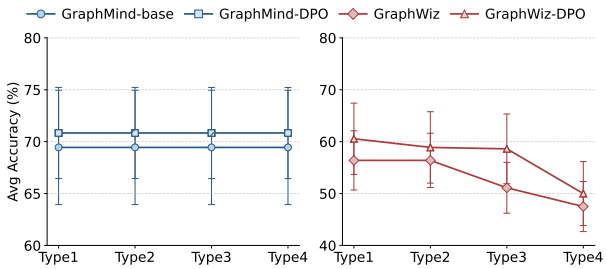


Figure 6: Model performance across different confusion types, showing average accuracy (%) with variance.

topologies. Meanwhile, the growth of network size poses significant challenges to existing models, and our proposed GraphMind has stronger robustness due to its decoupling of external graph storage and internal LLM reasoning.

5.4 GraphMind's Performance in Handling Interactive Disorder (RQ3)

To investigate model's ability to cope with interactive disorder, we conduct experiments separately on input and output phase.

For the input-disorder experiment, we rearranged the edges of our prebuilt graphs according to four strategies, with nodes first arranged in ascending order of their node IDs: (1) fully ordered nodes and edges, (2) edges unordered per node but nodes ordered, (3) nodes unordered but edges ordered per node, and (4) fully unordered nodes and edges. We then evaluated GraphMind and GraphWiz on a sampled test set under each arrangement. As the level of disorder increased, GraphWiz's accuracy fell from 56.38% to 47.50%, and GraphWiz-DPO's accuracy declined from 60.56% to 50.00%. In contrast, both GraphMind-base and GraphMind-DPO maintained constant accuracy across all four settings. This outcome demonstrates that our approach—decoupling graph construction from large-model reasoning—effectively eliminates the input-disorder problem in graph structure inference. The detail experiment data is shown in Appendix C Table 11.

For the output-disorder experiment, we compare GraphMind's function-calling accuracy with and without Grammar-refinement. As shown in Table 3, applying Grammar-refinement reduced the error rate for function usage to below 1% in most tasks. Specially in the flow task, the need to perform modification and lookup operations leads to the invocation of all five `GraphFunction`. Yet, the flow task unfortunately has fewest training samples in adopted GraphWiz dataset, which in turn results in comparatively low accuracy in function usage. Despite of this, our experiments still demonstrate that applying the grammar-centric refinement consistently and significantly reduces the error rate by nearly 50%. These results indicate that we markedly mitigates the output-disorder by guiding the model toward well-formed, consistent reasoning outputs.

5.5 Perplexity and Inference Efficiency (RQ4)

To obtain a direct comparison of inference time and perplexity (PPL) between GraphMind and alternative approaches, we evaluated both GraphMind and GraphWiz on our constructed test set and the corresponding GraphWiz test set. In the left panel of Figure 7, GraphMind-base achieves an average PPL of 1.12, while GraphMind-DPO attains 1.11—corresponding to reductions

Table 3: `GraphFunction` error Rates for Mistral-7B and Llama Models. w/o GR means LLM without grammar-refinement, w/ GR means LLM with grammar-refinement.

Task	Mistral-7B		Llama-7B	
	w/o GR	w/ GR	w/o GR	w/ GR
cycle	1.88%	0.87%	1.27%	0.38%
connect	0.87%	0.35%	0.72%	0.18%
topology	1.03%	0.65%	1.88%	0.78%
bipartite	1.15%	0.66%	1.40%	0.45%
shortest	1.61%	0.46%	1.01%	0.49%
triangle	1.71%	0.66%	1.68%	0.37%
flow	54.68%	28.94%	72.94%	29.77%
hamilton	1.58%	0.33%	1.34%	0.56%
subgraph	0.96%	0.21%	1.23%	0.76%

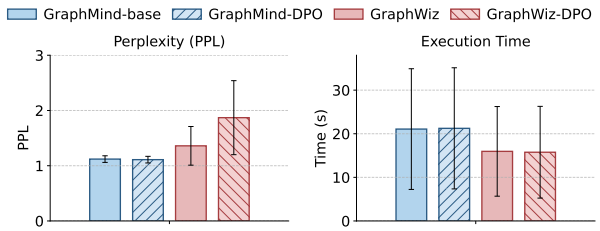


Figure 7: The figure illustrates PPL and inference time compare between GraphMind and GraphWiz.

of 17.6% and 40.6% compared to GraphWiz and GraphWiz-DPO, respectively. This substantial decrease in PPL can be attributed to the more rigorous inference logic attributed to our semi-formal language, combining `GraphFunction` and natural language. Our proposed grammar constrains the model's reasoning trajectory and makes it easier to predict each subsequent token , thereby mitigating interactive disorder. In the right panel of Figure 7, GraphMind-base exhibits an average inference speed of 21.07s/it and GraphMind-DPO 21.24s/it; the marginal slowdown relative to GraphWiz remains acceptable. Together, these results demonstrate that our approach not only enhances logical coherence during inference but also substantially mitigates hallucination without incurring prohibitive computational overhead. The detail experiment data is shown in Appendix C Table 7 and 8.

5.6 Extending GraphMind to Text-Attributed Graph Understanding (RQ5)

While GraphMind was originally designed for pure graph structure reasoning, it can be readily extended to handle text-attributed graphs such as citation networks. For node classification tasks, GraphMind leverages the `GraphFunction` to perform semantic-aware neighborhood exploration. The key insight is that effective node classification using LLMs depends on identifying semantically relevant nodes within a 2-hop neighborhood. As illustrated in Figure 8, the system employs a BFS-based approach that starts at the target node and dynamically evaluates each visited node: if the

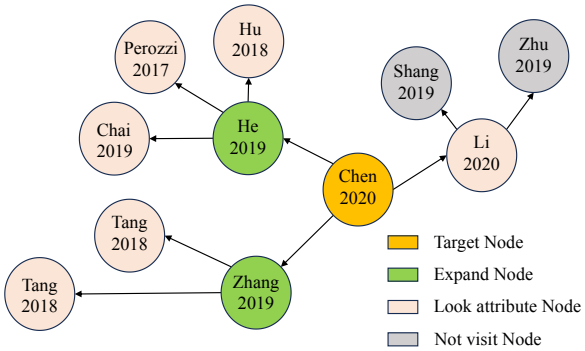


Figure 8: A 2-hop neighborhood of citation graph.

Table 4: Comparison of model performance on the ogbn_Arxiv dataset. Llaga and GraphMind were evaluated using 5 runs of random sampling (1000 samples per run).

Model	ogbn_Arxiv
TAPE [22]	0.7553 ± 0.0025
Llaga [5]	0.7491 ± 0.0101
GraphGPT	0.6874 ± 0.0104
GraphMind (Ours)	**0.7637** ± 0.0103

node's information suggests potential semantic relevance, its neighbors are explored; otherwise, the traversal terminates at that node. This selective expansion process continues until all relevant nodes within the 2-hop radius are identified, which are then processed by the LLM for final classification.

We adopt OGBN-arxiv [16] citation network and present the performance comparison with existing Graph LLMs methods in Table 4. We randomly select 1,000 samples as test set, and conduct five runs for evaluation. Our proposed GraphMind achieves an accuracy of 76.37%, surpassing other specific LLM-based node classification methods—TAPE [22] at 75.53% and Llaga [5] at 74.91%. These findings indicate that GraphMind shows promising capabilities to extend to text-attributed graphs. Meanwhile, by incorporating more appropriate GraphFunctions, we may achieve enhanced performance. An example of GraphMind's inference process on the OGBN-arxiv dataset is provided in Appendix D (see page 18).

6 Conclusion

In this paper, we propose GraphMind, a framework that enhances large language models (LLMs) in graph structure reasoning by utilizing semi-formal language. Our method addresses critical issues of interaction disorder during LLM input and output phases, leading to significant performance improvements across various benchmarks. We empirically validate the effectiveness and practicality of GraphMind in tasks of pure graph reasoning, processing large and dense graphs, inference speed, and handling graph data containing textual information. Future work will focus on expanding its capabilities to handle more diverse graph data types, such as dynamic and heterogeneous graphs.

References

[1] Yukun Cao, Shuo Han, Zengyi Gao, Zezhong Ding, Xike Xie, and S. Kevin Zhou. 2024. GraphInsight: Unlocking Insights in Large Language Models for Graph Structure Understanding. arXiv:2409.03258 [cs.CL] https://arxiv.org/abs/2409.03258

[2] Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. 2023. GraphLLM: Boosting Graph Reasoning Ability of Large Language Model. *CoRR* abs/2310.05845 (2023).

[3] Shantanu Chandra, Pushkar Mishra, Helen Yannakoudakis, Madhav Nimishakavi, Marzieh Saeidi, and Ekaterina Shutova. 2020. Graph-based Modeling of Online Communities for Fake News Detection. *CoRR* abs/2008.06274 (2020).

[4] Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. 2024. 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 2024, Barcelona, Spain, August 25-29, 2024*. ACM, 353–364.

[5] Runjin Chen, Tong Zhao, Ajay Kumar Jaiswal, Neil Shah, and Zhangyang Wang. 2024. LLaGA: Large Language and Graph Assistant. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

[6] Zhikai Chen, Haitao Mao, Hongzhi Wen, Haoyu Han, Wei Jin, Haiyang Zhang, Hui Liu, and Jiliang Tang. 2024. Label-free Node Classification on Graphs with Large Language Models (LLMs). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

[7] Eli Chien, Wei-Cheng Chang, Cho-Jui Hsieh, Hsiang-Fu Yu, Jiong Zhang, Olgica Milenkovic, and Inderjit S. Dhillon. 2022. Node Feature Extraction by Self-Supervised Multi-scale Neighborhood Prediction. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

[8] Sayak Ray Chowdhury, Anush Kini, and Nagarajan Natarajan. 2024. Provably Robust DPO: Aligning Language Models with Noisy Feedback. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. doi:10.18653/v1/N19-1423

[10] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a Graph: Encoding Graphs for Large Language Models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.

[11] Hao Fei, Shengqiong Wu, Hanwang Zhang, Tat-Seng Chua, and Shuicheng Yan. 2024. Vitron: A Unified Pixel-level Vision LLM for Understanding, Generating, Segmenting, Editing. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[12] Jiayan Guo, Lun Du, and Hengyu Liu. 2023. GPT4Graph: Can Large Language Models Understand Graph Structured Data ? An Empirical Evaluation and Benchmarking. *CoRR* abs/2305.15066 (2023). arXiv:2305.15066 doi:10.48550/ARXIV.2305.15066

[13] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. arXiv:2403.14608 [cs.LG] https://arxiv.org/abs/2403.14608

[14] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.).

[15] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. 2022. An empirical analysis of compute-optimal large language model training. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *Advances in Neural Information Processing Systems (NeurIPS)* 33 (2020), 22118–22133. https://arxiv.org/abs/2005.00687

[17] Yuntong Hu, Zheng Zhang, and Liang Zhao. 2023. Beyond Text: A Deep Dive into Large Language Models' Ability on Understanding Graph Data. *CoRR* abs/2310.04944 (2023).

[18] Hamish Ivison, Yizhong Wang, Jiacheng Liu, Zeqiu Wu, Valentina Pyatkin, Nathan Lambert, Noah A. Smith, Yejin Choi, and Hanna Hajishirzi. 2024. Unpacking DPO and PPO: Disentangling Best Practices for Learning from Preference Feedback. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[19] Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A General Framework for Large Language Model to Reason over Structured Data. arXiv:2305.09645 [cs.CL] https://arxiv.org/abs/2305.09645

[20] Yuhan Li, Zhixun Li, Peisong Wang, Jia Li, Xiangguo Sun, Hong Cheng, and Jeffrey Xu Yu. 2024. A Survey of Graph Meets Large Language Model: Progress and Future Directions. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 8123–8131.

[21] Chang Liu and Bo Wu. 2023. Evaluating Large Language Models on Graphs: Performance Insights and Comparative Analysis. arXiv:2308.11224 [cs.AI] https://arxiv.org/abs/2308.11224

[22] Lin Liu, Lingxi Xie, Xiaopeng Zhang, Shanxin Yuan, Xiangyu Chen, Wengang Zhou, Houqiang Li, and Qi Tian. 2022. TAPE: Task-Agnostic Prior Embedding for Image Restoration. In *Computer Vision - ECCV 2022 - 17th European Conference, Tel Aviv, Israel, October 23-27, 2022, Proceedings, Part XVIII (Lecture Notes in Computer Science, Vol. 13678)*. Springer, 447–464.

[23] Pengfei Liu, Yiming Ren, Jun Tao, and Zhixiang Ren. 2024. GIT-Mol: A multi-modal large language model for molecular science with graph, image, and text. *Comput. Biol. Medicine* 171 (2024), 108073.

[24] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=Bkg6RiCqY7

[25] Bin Lu, Xiaoying Gan, Weinan Zhang, Huaxiu Yao, Luoyi Fu, and Xinbing Wang. 2022. Spatio-Temporal Graph Few-Shot Learning with Cross-City Knowledge Transfer. In *KDD '22: The 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 14 - 18, 2022*. ACM, 1162–1172.

[26] Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, and Xing Xie. 2024. GraphInstruct: Empowering Large Language Models with Graph Understanding and Reasoning Capability. *CoRR* abs/2403.04483 (2024).

[27] Zizhuo Meng, Ke Wan, Yadong Huang, Zhidong Li, Yang Wang, and Feng Zhou. 2024. Interpretable Transformer Hawkes Processes: Unveiling Complex Interactions in Social Networks. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*. ACM, 2200–2211.

[28] Muhammad Ferjad Naeem, Muhammad Gul Zain Ali Khan, Yongqin Xian, Muhammad Zeshan Afzal, Didier Stricker, Luc Van Gool, and Federico Tombari. 2023. I2MVFormer: Large Language Model Generated Multi-View Document Supervision for Zero-Shot Image Classification. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*. IEEE, 15169–15179.

[29] Muhammad Ferjad Naeem, Yongqin Xian, Luc Van Gool, and Federico Tombari. 2022. I2DFormer: Learning Image to Document Attention for Zero-Shot Image Classification. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035.

[31] Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Seyed Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. 2024. Let Your Graph Do the Talking: Encoding Structured Data for LLMs. *CoRR* abs/2402.05862 (2024).

[32] Shraman Pramanick, Guangxing Han, Rui Hou, Sayan Nag, Ser-Nam Lim, Nicolas Ballas, Qifan Wang, Rama Chellappa, and Amjad Almahairi. 2024. Jack of All Tasks, Master of Many: Designing General-purpose Coarse-to-Fine Vision-Language Model. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*. IEEE, 14076–14088.

[33] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[34] Yi Ren and Danica J. Sutherland. 2025. Learning Dynamics of LLM Finetuning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

[35] Zhongwei Ren, Zhicheng Huang, Yunchao Wei, Yao Zhao, Dongmei Fu, Jiashi Feng, and Xiaojie Jin. 2024. PixelLM: Pixel Reasoning with Large Multimodal Model. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*. IEEE, 26364–26373.

[36] Yuda Song, Hanlin Zhang, Carson Eisenach, Sham M. Kakade, Dean P. Foster, and Udaya Ghai. 2025. Mind the Gap: Examining the Self-Improvement Capabilities of Large Language Models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

[37] Gu Tang, Xiaoying Gan, Jinghe Wang, Bin Lu, Lyuwen Wu, Luoyi Fu, and Chenghu Zhou. 2024. EditKG: Editing Knowledge Graph for Recommendation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2024, Washington DC, USA, July 14-18, 2024*. ACM, 112–122.

[38] Mistral AI team. 2023. Mistral 7B. https://mistral.ai/news/announcing-mistral-7b/

[39] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *CoRR* abs/2307.09288 (2023).

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 5998–6008.

[41] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can Language Models Solve Graph Problems in Natural Language?. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

[42] Haishuai Wang, Yang Gao, Xin Zheng, Peng Zhang, Hongyang Chen, Jiajun Bu, and Philip S. Yu. 2024. Graph Neural Architecture Search with GPT-4. arXiv:2310.01436 [cs.LG] https://arxiv.org/abs/2310.01436

[43] Junkang Wu, Yuexiang Xie, Zhengyi Yang, Jiancan Wu, Jinyang Gao, Bolin Ding, Xiang Wang, and Xiangnan He. 2024. β-DPO: Direct Preference Optimization with Dynamic β. In *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*.

[44] Shusheng Xu, Wei Fu, Jiaxuan Gao, Wenjie Ye, Weilin Liu, Zhiyu Mei, Guangju Wang, Chao Yu, and Yi Wu. 2024. Is DPO Superior to PPO for LLM Alignment? A Comprehensive Study. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net.

[45] Shicheng Xu, Liang Pang, Mo Yu, Fandong Meng, Huawei Shen, Xueqi Cheng, and Jie Zhou. 2024. Unsupervised Information Refinement Training of Large Language Models for Retrieval-Augmented Generation. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*. Association for Computational Linguistics, 133–145.

[46] Jui-Nan Yen, Si Si, Zhao Meng, Felix X. Yu, Sai Surya Duvvuri, Inderjit S. Dhillon, Cho-Jui Hsieh, and Sanjiv Kumar. 2025. LoRA Done RITE: Robust Invariant Transformation Equilibration for LoRA Optimization. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

[47] Shiqi Zhang, Yiqian Huang, Jiachen Sun, Wenqing Lin, Xiaokui Xiao, and Bo Tang. 2023. Capacity Constrained Influence Maximization in Social Networks. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*. ACM, 3376–3385.

[48] Weijia Zhang, Jindong Han, Zhao Xu, Hang Ni, Hao Liu, and Hui Xiong. 2024. Urban Foundation Models: A Survey. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*. ACM, 6633–6643.

[49] Yutong Zhang, Lixing Chen, Shenghong Li, Nan Cao, Yang Shi, Jiaxin Ding, Zhe Qu, Pan Zhou, and Yang Bai. 2025. Way to Specialist: Closing Loop Between Specialized LLM and Evolving Domain Knowledge Graph. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.1, KDD 2025, Toronto, ON, Canada, August 3-7, 2025*. ACM, 1996–2007.

[50] Zeyang Zhang, Xin Wang, Ziwei Zhang, Haoyang Li, Yijian Qin, and Wenwu Zhu. 2024. LLM4DyG: Can Large Language Models Solve Spatial-Temporal Problems on Dynamic Graphs?. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2024, Barcelona, Spain, August 25-29, 2024*. ACM, 4350–4361.

GenAI Usage Disclosure

Generative AI tools were used only for light editing and formatting assistance. No LLM-generated scientific content is included, except model outputs that are part of the experimental analysis.

A GraphWiz Dataset

In this section, we provide a comprehensive overview of the dataset employed in our study. As shown in Table 5, the dataset encompasses nine distinct reasoning tasks, each characterized by a unique combination of structural complexity and semantic challenges. For each task, we outline its specific objective, the approximate size of the underlying graph structures (in terms of nodes and edges), and a qualitative assessment of its difficulty level, which reflects the depth of reasoning required. To further facilitate understanding of the dataset's composition, Table 6 presents detailed statistics of the training and test splits. This includes the total number of graph-question pairs, the cumulative number of nodes across all graphs, and the distribution of reasoning paths that illustrate the nature of the inferential processes involved. Together, these tables offer a foundational perspective on the dataset's scale, diversity, and the reasoning complexity it poses for downstream tasks.

B Baselines

In this section, we introduce the baselines we employ in detail, namely NAIVE-SFT, NLGraph, GraphInstruct and Talk-Like-a-Graph.

- **NAIVE-SFT**: This baseline method simply concatenates each question with its corresponding answer and fine-tunes the language model on these question–answer pairs, without incorporating any structural graph information.
- **NLGraph** [41]: This approach enhances the model's ability to perform graph-based reasoning by leveraging prompt-based strategies such as chain-of-thought and self-consistency, enabling more explicit intermediate reasoning steps.
- **GraphInstruct** [26]: To mitigate the effects of input disorder, GraphInstruct randomly masks portions of the input graph during training. However, since it still directly feeds raw graph structures into the language model, its effectiveness on complex graph reasoning tasks remains limited.
- **Talk-Like-a-Graph** [10]: This method investigates alternative prompt formulations, including adjacency list representations and interpersonal relationship analogies, to express graph structures in a more natural language format. In addition, it incorporates expert-designed graph prompts to further improve the model's structural comprehension and reasoning ability.
- **GraphWiz** [4]: GraphWiz fine-tunes language models using a large collection of high-quality graph reasoning processes generated by GPT-4. It then performs Direct Preference Optimization (DPO) fine-tuning using a preference dataset constructed based on the correctness and length of the generated answers, thereby aligning model outputs with desired reasoning quality.

Table 5: Overview of nine tasks in our GraphMind benchmark with problem definition, time complexity of representative algorithms, graph type (weighted/directed), node size range, and task difficulty. $|\mathcal{V}|$ and $|\mathcal{E}|$ indicate the number of nodes and edges in the graph.

Problem	Definition	Time Complexity	Weighted?	Directed?	Node Range	Difficulty
Cycle Detection	Detect if a given graph $\mathcal{G}$ contains any cycles.	$O(\|\mathcal{V}\| + \|\mathcal{E}\|)$	×	×	[2,100]	Easy
Connectivity	Assess if two nodes u and v in $\mathcal{G}$ are connected.	$O(\|\mathcal{V}\| + \|\mathcal{E}\|)$	×	×	[2,100]	Easy
Bipartite Graph Check	Judge if $\mathcal{G}$ is bipartite.	$O(\|\mathcal{V}\| + \|\mathcal{E}\|)$	×	✓	[2,100]	Easy
Topological Sort	Find a topological ordering in a DAG $\mathcal{G}$.	$O(\|\mathcal{V}\| + \|\mathcal{E}\|)$	×	✓	[2,50]	Easy
Shortest Path	Compute shortest path between u and v in $\mathcal{G}$.	$O(\|\mathcal{E}\| + \|\mathcal{V}\| \log \|\mathcal{V}\|)$	✓	×	[2,100]	Medium
Maximum Triangle Sum	Find max sum of weights for any connected triplet.	$O(\|\mathcal{V}\|^3)$	✓	×	[2,25]	Medium
Maximum Flow	Calculate max flow from s to t in $\mathcal{G}$.	$O(\|\mathcal{V}\|^2\sqrt{\|\mathcal{E}\|})$	✓	✓	[2,50]	Medium
Hamilton Path	Determine if $\mathcal{G}$ has a Hamiltonian path.	NP-Complete	×	×	[2,50]	Hard
Subgraph Matching	Verify if $\mathcal{G}$ contains subgraph isomorphic to $\mathcal{G}'$.	NP-Complete	×	✓	[2,30]	Hard

Table 6: Statistics of our GraphMind corpus, including total samples ($\mathcal{G}$-$\mathcal{Q}$), nodes ($\mathcal{V}$) and reasoning paths $\mathcal{R}$.

		Easy				Medium			Hard		Sum.
Tasks		cycle	connect	bipartite	topology	shortest	triangle	flow	hamilton	Subgraph	
Train	Total $\mathcal{G}$-$\mathcal{Q}$	3,717	2,687	2,013	902	1,392	2,756	405	2,097	1,435	17,158
	Total $\mathcal{V}$	84,980	79,853	58,860	10,146	23,204	14,714	4,333	33,284	7,847	315,051
	Total $\mathcal{R}$	13,122	10,001	9,324	4,481	5,859	13,483	747	8,454	6,274	72,785
Test	Total $\mathcal{G}$-$\mathcal{Q}$	400	400	400	400	400	400	400	400	400	3,600
	Total $\mathcal{V}$	19,570	19,500	19,515	9,449	19,449	4,990	10,024	9,732	6,594	118,823

C Additional Experiment Results

In this section, we present the complete set of numerical results corresponding to the figures discussed in the main text, offering deeper insights into model performance and efficiency.

C.1 Perplexity and Inference Time

Table 8 and Table 7 report the detailed perplexity (PPL) and inference time measurements, respectively, for both GraphMind and GraphWiz. These results complement the visual summary in Figure 7 and allow for a more granular comparison of model fluency and computational efficiency. As shown in Table 8, GraphMind consistently achieves lower PPL scores than GraphWiz across all nine core tasks, with notably more stable performance. In addition, Table 7 indicates that GraphMind maintains acceptable inference times for all nine tasks, underscoring its practical efficiency.

C.2 Robustness to Input Order Variations

Table 11 presents the accuracy results of GraphMind and GraphWiz under different input orders of graph descriptions. This analysis corresponds to the trends shown in Figure 6 and sheds light on each model's robustness to graph input permutations. As observed in Table 11, the performance of GraphWiz deteriorates significantly as the degree of disorder in the graph structure descriptions increases. In contrast, GraphMind remains largely unaffected by such variations, demonstrating substantially stronger robustness to input perturbations.

C.3 Function Usage Accuracy

Table 3 provides the function usage accuracy of GraphMind under two settings: with and without grammar-centric refinement. These results, associated with Figure 7, demonstrate the effectiveness of the refinement strategy in improving the precision of function invocation during reasoning.

D Prompt Design of GraphMind

We introduce a semi-formal language framework designed to encompass nine distinct types of graph-related questions, each illustrated with representative question–answer (QA) pairs. Detailed

examples for each question type are provided in the corresponding boxes: bipartite graphs (page 13), subgraph matching (page 17), cycle detection (page 15), shortest path queries (page 16), graph connectivity (page 15), flow networks (page 15), maximum triangle sum (page 18), Hamiltonian cycles (page 16), and graph topology (page 17).

E Comparison of Different Methods

Tables 12,13, and14 present a comparative analysis of the reasoning processes of our method and the baselines GraphWiz and Talk-like-a-graph on the Easy, Medium, and Hard subsets, respectively.

F Examples of Disordered Output from LLMs

In this section, we reveal a critical phenomenon in large language models (LLMs): their reasoning about graph-structured data is highly sensitive to input permutations. Even minor variations in the ordering of nodes or edges can trigger inconsistent or contradictory outputs. An illustrative example from GPT-4 is provided in page 19.

G Limitations

Although GraphMind demonstrates strong performance in graph data processing, it still has several limitations and areas for future improvement.

(1) **Limitations in `GraphFunction` Design**: While our design meets the needs of most graph data processing tasks, there remain scenarios where the current set of functions is insufficient. In the future, more functions could be introduced or more generalizable ones could be designed to address a broader range of use cases. However, the design of `GraphFunction` must be handled with care—overly complex or highly similar functions may confuse LLMs during processing, ultimately affecting performance. Therefore, it is essential to strike a balance between function diversity and complexity during design.

(2) **Diversity of Graph Data**: GraphMind currently focuses primarily on structured graph data. Its capabilities remain limited when dealing with unstructured graph data or other types of graph data, such as dynamic graphs or heterogeneous graphs. Future work may involve extending GraphMind's capabilities to support a wider variety of graph data types.

(3) **Limitations of LLMs**: Due to resource constraints, we currently utilize Llama-2-7B and Mistral-7B as base models for our experiments. Future work may consider leveraging larger-scale models or more advanced architectures to further enhance GraphMind's performance. Additionally, the training and inference speed of GraphMind is constrained by the nature of LLMs. Optimizing the training and inference pipeline could be an effective way to improve overall efficiency.

Table 7: Average execution time (in seconds) of each model variant on 9 benchmark tasks. Standard models (GraphMind, GraphWiz) are shown above, while their DPO-enhanced variants are shown below.

Model Variant	cycle	connect	bipartite	topology	shortest	triangle	flow	hamilton	subgraph	Average
GraphMind	4.75	4.03	18.05	49.50	33.56	10.21	4.33	28.03	37.20	21.07
GraphWiz	4.03	3.31	14.22	37.67	27.05	7.63	4.35	16.52	28.91	15.96
GraphMind-DPO	4.75	4.03	18.01	50.18	33.61	10.89	4.33	28.21	37.12	21.24
GraphWiz-DPO	4.05	3.38	13.99	37.34	27.03	7.79	3.91	16.02	28.35	15.76

Table 8: Perplexity (PPL) scores of each model variant across 9 benchmark graph reasoning tasks. Lower values indicate better performance. Standard models (GraphMind, GraphWiz) are shown above, while their DPO-enhanced variants are shown below.

Model Variant	cycle	connect	bipartite	topology	shortest	triangle	flow	hamilton	subgraph	Average
GraphMind	1.15	1.15	1.15	1.09	1.08	1.17	1.21	1.02	1.05	1.12
GraphWiz	1.31	1.30	2.25	1.74	1.48	1.27	1.35	1.35	1.19	1.36
GraphMind-DPO	1.15	1.14	1.15	1.08	1.07	1.15	1.20	1.02	1.03	1.11
GraphWiz-DPO	1.50	1.44	2.87	4.17	1.63	1.37	1.49	2.12	1.27	1.87

Table 9: Performance comparison on LLaMA 2-7B backbone. The performance indicators are consistent with Table 2.

Algorithms	Easy			Medium			Hard			Average (Rank)
	cycle	connect	bipartite	topology	shortest	triangle	flow	hamilton	subgraph	
Naive SFT	73.75	83.50	41.25	4.00	9.50	30.00	16.50	69.00	75.45	44.81 (6)
GraphInstruct	64.50	62.00	41.00	0.50	7.00	0.00	5.5	33.00	51.00	29.39 (9)
GraphInstruct+	34.00	54.00	47.00	7.00	1.00	13.00	17.00	40.00	42.00	32.78 (8)
Talk Like a graph (a.)	80.00	93.00	61.00	2.00	30.00	1.00	44.00	55.00	45.00	45.67 (5)
Talk Like a graph (e.)	66.00	69.00	43.00	13.00	13.00	2.00	22.00	43.00	53.00	36.00 (7)
GraphWiz	91.50	87.00	74.00	18.00	28.00	38.25	24.50	52.25	**82.25**	55.08 (4)
GraphMind-base	**98.50**	**100.00**	**98.00**	79.75	**32.75**	46.50	15.00	66.25	75.00	67.97 (2)
GraphWiz-DPO	89.00	82.50	84.75	46.75	24.00	**52.75**	43.50	**81.50**	77.25	65.00 (3)
GraphMind-DPO	98.00	99.50	89.00	**81.25**	32.25	**52.75**	**46.50**	77.75	67.50	**71.61** (1)

Table 10: Performance comparison (%) between standard and DPO-enhanced models on different graph scales. GraphMind shows consistent performance on large graphs, while its DPO variant improves small graph accuracy by 7.69%. GraphWiz demonstrates better performance on small graphs but struggles with large-scale tasks.

Model Variant	Large Graph Accuracy	Dense Graph Accuracy
GraphMind	88.00	78.00
GraphMind-DPO	88.00	84.00
GraphWiz	12.00	64.00
GraphWiz-DPO	26.00	62.00

Bipartite

Question:A directed graph is represented as a table,node range from 0 to 4. Check if the given graph is bipartite. Answer by Yes or No. **Answer**: Convert the directed graph to an undirected graph by ignoring edge directions, then proceed with BFS traversal. If we can color the graph with two colors without adjacent nodes sharing the same color, then it is bipartite. Color the start node 0 with color 0: "'MODIFY_NODE[0,0]"' . Start travers from node 0,find the color of node 0: "'FIND_NODE[0]"' = 0,find the node connected to node 0:"'FIND_ALL_EDGES[0]"'=(4,NUL)|(2,NUL),Check the color of node that is visited:No conflict, so color the node not visited with color 1: "'MODIFY_NODE[4,1]"' , "'MODIFY_NODE[2,1]"' . Traverse node 4 beacuse it's unvisited,find the color of node 4:"'FIND_NODE[4]"'=1,find the node connected to node 4:"'FIND_ALL_EDGES[4]"'=(0,NUL)|(2,NUL),Check the color of node that is visited: "'FIND_NODE[0]"' =0,"' FIND_NODE[2]"' =1. Conflict found between node 2 and node 4, so the graph is not bipartite. So the answer is### No.

Table 11: Accuracy Comparison with different degree of confusion

Task	Model	Fully ordered	Edges unordered per node, nodes ordered	Sequential nodes, ordered edges per node	Fully unordered
Cycle	**GraphMind-base**	97.5	97.5	97.5	97.5
	GraphWiz	82.5	85.0	62.5	60.0
	GraphMind-DPO	95.0	95.0	95.0	95.0
	GraphWiz DPO	95.0	92.5	90.0	90.0
Connectivity	**GraphMind-base**	97.5	97.5	97.5	97.5
	GraphWiz	92.5	97.5	82.5	75.0
	GraphMind-DPO	100.0	100.0	100.0	100.0
	GraphWiz DPO	97.5	90.0	87.5	82.5
Hamilton	**GraphMind-base**	75.0	75.0	75.0	75.0
	GraphWiz	62.5	57.5	57.5	57.5
	GraphMind-DPO	72.5	72.5	72.5	72.5
	GraphWiz DPO	82.5	80.0	77.5	57.5
Substructure	**GraphMind-base**	65.0	65.0	65.0	65.0
	GraphWiz	80.0	70.0	70.0	72.5
	GraphMind-DPO	55.0	55.0	55.0	55.0
	GraphWiz DPO	82.5	87.5	85.0	65.0
Bipartite	**GraphMind-base**	97.5	97.5	97.5	97.5
	GraphWiz	85.0	77.5	85.0	70.0
	GraphMind-DPO	92.5	92.5	92.5	92.5
	GraphWiz DPO	92.5	92.5	97.5	82.5
Flow	**GraphMind-base**	15.0	15.0	15.0	15.0
	GraphWiz	25.0	30.0	20.0	20.0
	GraphMind-DPO	40.0	40.0	40.0	40.0
	GraphWiz DPO	32.5	35.0	27.5	27.5
Shortest	**GraphMind-base**	35.0	35.0	35.0	35.0
	GraphWiz	22.5	22.5	20.0	5.0
	GraphMind-DPO	37.5	37.5	37.5	37.5
	GraphWiz DPO	25.0	17.5	27.5	10.0
Triangle	**GraphMind-base**	60.0	60.0	60.0	60.0
	GraphWiz	40.0	42.5	35.0	42.5
	GraphMind-DPO	65.0	65.0	65.0	65.0
	GraphWiz DPO	35.0	32.5	32.5	30.0
Topology	**GraphMind-base**	82.5	82.5	82.5	82.5
	GraphWiz	17.5	25.0	27.5	25.0
	GraphMind-DPO	80.0	80.0	80.0	80.0
	GraphWiz DPO	2.5	2.5	2.5	5.0

Connectivity

Question: A undirected graph is represented as a table. Node range from 0 to 13. Check if node 11 and node 7 are connected in the given graph table. Answer by Yes or No. **Answer**: We use BFS to traverse the graph. If we can reach node 7 from node 11, then they are connected. Traverse from node 11: "'FIND_ALL_EDGES[11]"' =(2,NUL)|(3,NUL)|(4,NUL)|(8,NUL)|(9,NUL). Traverse node 2 beacuse it's unvisited: "'FIND_ALL_EDGES[2]"' =(5,NUL)|(11,NUL)|(13,NUL)|(6,NUL). Traverse node 3 beacuse it's unvisited:"'FIND_ALL_EDGES[3]"' =(4,NUL)|(11,NUL). Traverse node 4 beacuse it's unvisited:"'FIND_ALL_EDGES[4]"' =(11,NUL)|(9,NUL)|(8,NUL)|(1,NUL)|(3,NUL). Traverse node 8 beacuse it's unvisited:"'FIND_ALL_EDGES[8]"' =(11,NUL)|(4,NUL)|(6,NUL). Traverse node 9 beacuse it's unvisited:"'FIND_ALL_EDGES[9]"' =(11,NUL)|(10,NUL)|(4,NUL)|(5,NUL). Traverse node 5 beacuse it's unvisited:"'FIND_ALL_EDGES[5]"' =(9,NUL)|(7,NUL)|(1,NUL)|(2,NUL). Node 7 is visited. After walking through the graph, we find that node 11 and node 7 are connected. So the answer is ### Yes.

Cycle

Question:A undirected graph is represented as a table. Node range from 0 to 4. Check if there is a cycle in the given graph table. Answer by Yes or No.
Answer: We use DFS to traverse the graph. If we can reach the visited node from any other node except next node, then there is a cycle. Traverse from node 0:"'FIND_ALL_EDGES[0]"'=(3,NUL)|(4,NUL)|(1,NUL)|(2,NUL),all nodes except the parent node -1 are unvisited. Proceed with traversal,traverse node 3:"'FIND_ALL_EDGES[3]"'=(4,NUL)|(0,NUL)|(1,NUL)|(2,NUL). All nodes except the parent node 0 are unvisited. Proceed with traversal,traverse node 4:"'FIND_ALL_EDGES[4]"'=(0,NUL)|(1,NUL)|(2,NUL)|(3,NUL). Node 0 is visited. After walking through the graph, we find that there is a cycle in the graph. So the answer is ### Yes.

Flow

Question:A directed graph is represented as a table,node range from 0 to 4. Find the maximum flow from node 0 to node 4 in the given graph table. Answer by a integer.
Answer: We use Edmonds-Karp algorithm to find the maximum flow from node 0 to node 4. Step 1, we need to add a reverse edge with 0 capacity for each edge in the graph. Find the edge of node 0:"'FIND_ALL_EDGES[0]"'=(1,6),Add reverse edge for edge with non-zero capacity:"'MODIFY_EDGE[0,1,0]"'. Find the edge of node 1:"'FIND_ALL_EDGES[1]"'=(2,2)|(0,0),Add reverse edge for edge with non-zero capacity:"'MODIFY_EDGE[1,2,0]"'. Find the edge of node 2:"'FIND_ALL_EDGES[2]"'=(4,6)|(1,0),Add reverse edge for edge with non-zero capacity:"'MODIFY_EDGE[2,4,0]"'. Find the edge of node 3:"'FIND_ALL_EDGES[3]"'=(4,4),Add reverse edge for edge with non-zero capacity:"'MODIFY_EDGE[3,4,0]"'. Find the edge of node 4:"'FIND_ALL_EDGES[4]"'=(2,0)|(3,0),Add reverse edge for edge with non-zero capacity:. Step 2, we need to find the augmenting path from node 0 to node 4. At first, the flow is 0. Find the augmenting path 1:['0', '1', '2', '4'],find the capacity of the path:"'FIND_EDGE[0,1]"'=6;"'FIND_EDGE[1,2]"'=2;"'FIND_EDGE[2,4]"'=6,so the capacity of the path is the minimum one:2. Add the flow to the path:"'MODIFY_EDGE[0,1,6-2]"';"'MODIFY_EDGE[1,0,0+2]"';"'MODIFY_EDGE[1,2,2-2]"';"'MODIFY_EDGE[2,1,0+2]"';"'MODIFY_EDGE[2,4,6-2]"';"'MODIFY_EDGE[4,2,0+2]"'. Now the flow is:0+2=2. No more augmenting path. The maximum flow is ### 2.

Table 12: Comparison of Talk-like-a-Graph, GraphWiz, and GraphMind Methods on Easy task

	Talk like a graph	**GraphWiz**	**GraphMind (Ours)**
(Easy) Connectivity	*Q:* In an undirected graph, `(i,j)` means that node i and node j are connected. G describes a graph among nodes 0, 1, 2, 3. The edges: `(0,2)` `(0,1)` `(1,3)`. Determine whether node 0 and 1 are connected. *A:* Yes, node 0 and node 1 are connected directly by an edge.	*Q:* Determine whether two nodes are connected. In an undirected graph, `(i,j)` means connection. Given nodes [0–3] and edges `(0,2)` `(0,1)` `(1,3)`. Is there a path between node 0 and 1? *A:* Node 0 and node 1 are directly connected by an edge, so there is a path between them. We can follow the path: [0->1], so the answer is yes. ### Yes.	*Q:* An undirected graph represented as table (nodes 0–4). Check connectivity between node 0 and 1. Answer "Yes" or "No". *A:* Use BFS from node 0: "'FIND_ALL_EDGES[0]"' = `(1,NULL)|(2,NULL)`, node 1 visited. **### Yes**

Hamilton Cycle

Question:A undirected graph is represented as a table. Node range from 0 to 4. Determine whether the graph has a Hamiltonian cycle. Answer by a Yes or No.

Answer: We will use the following steps to determine whether the graph has a Hamiltonian cycle. Step 1:count the degree of each node. We will attempt to find a Hamiltonian path starting from node 2 up to three times. Attempt 1:Start from node 2. Find the edge of node 2 in the graph:"'FIND_ALL_EDGES[2]"'=(0,NUL). Unvisited nodes are: ['0']. Choose a random node from the unvisited nodes:['0']. Find the edge of node 0 in the graph:"'FIND_ALL_EDGES[0]"'=(1,NUL)|(3,NUL)|(4,NUL)|(2,NUL). Unvisited nodes are: ['1', '3', '4', '2']. Choose a random node from the unvisited nodes:['1', '3', '4', '2']. Find the edge of node 2 in the graph:"'FIND_ALL_EDGES[2]"'=(0,NUL). No way to go,Attempt 1 failed. Attempt 2:Start from node 2. Find the edge of node 2 in the graph:"'FIND_ALL_EDGES[2]"'=(0,NUL). Unvisited nodes are: ['0']. Choose a random node from the unvisited nodes:['0']. Find the edge of node 0 in the graph:"'FIND_ALL_EDGES[0]"'=(1,NUL)|(3,NUL)|(4,NUL)|(2,NUL). Unvisited nodes are: ['1', '3', '4', '2']. Choose a random node from the unvisited nodes:['1', '3', '4', '2']. Find the edge of node 1 in the graph:"'FIND_ALL_EDGES[1]"'=(3,NUL)|(0,NUL). Unvisited nodes are: ['3']. Choose a random node from the unvisited nodes:['3']. Find the edge of node 3 in the graph:"'FIND_ALL_EDGES[3]"'=(4,NUL)|(0,NUL)|(1,NUL). Unvisited nodes are: ['4']. Choose a random node from the unvisited nodes:['4']. Find the edge of node 2 and node 4 in the graph:"'FIND_EDGE[2,4]"'=None. The first node and the last node are not connected,Attempt 2 failed. Attempt 3:Start from node 2. Find the edge of node 2 in the graph:"'FIND_ALL_EDGES[2]"'=(0,NUL). Unvisited nodes are: ['0']. Choose a random node from the unvisited nodes:['0']. Find the edge of node 0 in the graph:"'FIND_ALL_EDGES[0]"'=(1,NUL)|(3,NUL)|(4,NUL)|(2,NUL). Unvisited nodes are: ['1', '3', '4', '2']. Choose a random node from the unvisited nodes:['1', '3', '4', '2']. Find the edge of node 3 in the graph:"'FIND_ALL_EDGES[3]"'=(4,NUL)|(0,NUL)|(1,NUL). Unvisited nodes are: ['4', '1']. Choose a random node from the unvisited nodes:['4', '1']. Find the edge of node 1 in the graph:"'FIND_ALL_EDGES[1]"'=(3,NUL)|(0,NUL). No way to go,Attempt 3 failed. All attempts failed, so the answer is ### No.

Shortest Path

Question:A undirected graph is represented as a table. Node range from 0 to 4. Find the shortest path from node 1 to node 2 in the given graph table. Answer by a integer.

Answer: We use Dijkstra's algorithm to find the shortest path from node 1 to node 2. Initialize the distance of 1 to 0 and all other nodes to infinity:"'MODIFY_NODE[1,0]"'. Now the distance is:[('1', 0)]. Node 1 has the smallest distance 0. Traverse node 1:"'FIND_ALL_EDGES[1]"'=(4,1). Now update the distance all the nodes:"'FIND_NODE[4]"'=NUL,0+1<NUL,update node 4:"'MODIFY_NODE[4,1]"';Now the distance well know is:[('4', 1)]. Node 4 has the smallest distance 1. Traverse node 4:"'FIND_ALL_EDGES[4]"'=(1,1)|(3,9). Now update the distance all the nodes:"'FIND_NODE[3]"'=NUL,1+9<NUL,update node 3:"'MODIFY_NODE[3,10]"';Now the distance well know is:[('3', 10)]. Node 3 has the smallest distance 10. Traverse node 3:"'FIND_ALL_EDGES[3]"'=(4,9)|(2,6). Now update the distance all the nodes:"'FIND_NODE[2]"'=NUL,10+6<NUL,update node 2:"'MODIFY_NODE[2,16]"';Now the distance well know is:[('2', 16)]. Node 2 has the smallest distance 16. So the answer is ### 16.

Subgraph Matching

Question:Two directed graphs are represented as a table. Graph_1 nodes are range from 0 to 5 and Graph_2 nodes are range from a to e. Determine whether the second graph is a substructure of the first graph. Answer by a Yes or No.

Answer: Treat all edges as bidirectional, ignoring their original directionality. We will use the following steps to determine whether the second graph is a substructure of the first graph. Step 1:count the degree of each node. Find the edge of node 0 in the Graph_1:"'FIND_ALL_EDGES[0]"'=(5,NUL),the degree of node 0 is 1,modify node 0:"'MODIFY_NODE[('0', 1)]"'. Find the edge of node 1 in the Graph_1:"'FIND_ALL_EDGES[1]"'=(4,NUL),the degree of node 1 is 1,modify node 1:"'MODIFY_NODE[('1', 1)]"'. Find the edge of node 2 in the Graph_1:"'FIND_ALL_EDGES[2]"'=(5,NUL),the degree of node 2 is 1,modify node 2:"'MODIFY_NODE[('2', 1)]"'. Find the edge of node 3 in the Graph_1: "'FIND_ALL_EDGES[3]"' =(5,NUL),the degree of node 3 is 1,modify node 3:"'MODIFY_NODE[('3', 1)]"'. Find the edge of node 4 in the Graph_1:"'FIND_ALL_EDGES[4]"'=(1,NUL),the degree of node 4 is 1,modify node 4:"'MODIFY_NODE[('4', 1)]"'. Find the edge of node 5 in the Graph_1:"'FIND_ALL_EDGES[5]"'=(0,NUL)|(2,NUL)|(3,NUL),the degree of node 5 is 3,modify node 5:"'MODIFY_NODE[('5', 3)]"'. Find the edge of node a in the Graph_2:"'FIND_ALL_EDGES[a]"'=(c,NUL)|(b,NUL),the degree of node a is 2,modify node a:"'MODIFY_NODE[('a', 2)]"'. Find the edge of node b in the Graph_2:"'FIND_ALL_EDGES[b]"'=(d,NUL)|(e,NUL)|(a,NUL),the degree of node b is 3,modify node b:"'MODIFY_NODE[('b', 3)]"'. Find the edge of node c in the Graph_2:"'FIND_ALL_EDGES[c]"'=(a,NUL),the degree of node c is 1,modify node c:"'MODIFY_NODE[('c', 1)]"'. Find the edge of node d in the Graph_2:"'FIND_ALL_EDGES[d]"'=(b,NUL),the degree of node d is 1,modify node d:"'MODIFY_NODE[('d', 1)]"'. Find the edge of node e in the Graph_2:"'FIND_ALL_EDGES[e]"'=(b,NUL),the degree of node e is 1,modify node e:"'MODIFY_NODE[('e', 1)]"'. Step2:Sort the degree of node in Graph_1 and Graph_2. Find the degree of node in Graph_1:"'FIND_NODE[0]"'=1;"'FIND_NODE[1]"'=1;"'FIND_NODE[2]"'=1;"'FIND_NODE[3]"'=1;"'FIND_NODE[4]"'=1; "'FIND_NODE[5]"' = 3. Sort the degree of node in Graph_1:[(5,3),(4,1),(3,1),(2,1),(1,1),(0,1)]. Find the degree of node in Graph_2:"'FIND_NODE[a]"'=2;"'FIND_NODE[b]"'=3;"'FIND_NODE[c]"'=1;"'FIND_NODE[d]"'=1;"'FIND_NODE[e]"'=1. Sort the degree of node in Graph_2:[(b,3),(a,2),(e,1),(d,1),(c,1)]. Step3:Compare the degree of node in Graph_1 and Graph_2.3>=3,the degree of node 5 is greater than or equal to the degree of node b,continue.1<2,the degree of node 4 is less than the degree of node a. So the answer is ### No.

Topology Sort

Question:A directed graph is represented as a table. Node range from 0 to 7. Output the topological sorting of the given graph table. Answer by a list of nodes.

Answer: We use Kahn's algorithm to find the topological sorting of the graph. Step 1:Initialize a queue with all nodes that have in-degree 0. Find the degree of each node:"'FIND_NODE[0]"'=3,"'FIND_NODE[1]"'=1,"'FIND_NODE[2]"'=4,"'FIND_NODE[3]"'=3,"'FIND_NODE[4]"' = 2,"'FIND_NODE[5]"'=3,"'FIND_NODE[6]"'=0, "'FIND_NODE[7]"' = 3. So the zero degree nodes are:['6']. Step 2:Use kahn's algorithm to find the topological sorting. The zero degree node not explored is:['6']. Choose node 6 to explore:"'FIND_ALL_EDGES[6]"'=(1,NUL)|(2,NUL)|(7,NUL)|(4,NUL)|(5,NUL). Decerease the in-degree of node connect to 6:"'FIND_NODE[1]"'=1,"'MODIFY_NODE[1,0]"' ;"'FIND_NODE[2]"'=4,"'MODIFY_NODE[2,3]"' ; "'FIND_NODE[7]"'=3,"'MODIFY_NODE[7,2]"' ; "'FIND_NODE[4]"'=2,"'MODIFY_NODE[4,1]"' ;"'FIND_NODE[5]"'=3, "'MODIFY_NODE[5,2]"' ;The zero degree node not explored is:['1']. Choose node 1 to explore:"'FIND_ALL_EDGES[1]"'=(4,NUL)|(3,NUL)|(7,NUL). Decerease the in-degree of node connect to 1:"'FIND_NODE[4]"' =1, "'MODIFY_NODE[4,0]"' ; "'FIND_NODE[3]"'=3,"'MODIFY_NODE[3,2]"';"'FIND_NODE[7]"'=2, "'MODIFY_NODE[7,1]"' ;The zero degree node not explored is:['4']. Choose node 4 to explore:"'FIND_ALL_EDGES[4]"'=(7,NUL)|(2,NUL)|(5,NUL)|(3,NUL)|(0,NUL). Decerease the in-degree of node connect to 4:"'FIND_NODE[7]"'=1,"'MODIFY_NODE[7,0]"';"'FIND_NODE[2]"'=3,"'MODIFY_NODE[2,2]"';"'FIND_NODE[5]"'=2, "'MODIFY_NODE[5,1]"' ; "'FIND_NODE[3]"' =2, "'MODIFY_NODE[3,1]"' ; "'FIND_NODE[0]"'=3,"'MODIFY_NODE[0,2]"';The zero degree node not explored is:['7']. Choose node 7 to explore:"'FIND_ALL_EDGES[7]"'=(3,NUL)|(0,NUL)|(2,NUL). Decerease the in-degree of node connect to 7:"'FIND_NODE[3]"'=1,"'MODIFY_NODE[3,0]"';"'FIND_NODE[0]"'=2,"'MODIFY_NODE[0,1]"';"'FIND_NODE[2]"'=2, "'MODIFY_NODE[2,1]"' ;The zero degree node not explored is:['3']. Choose node 3 to explore:"'FIND_ALL_EDGES[3]"'=(5,NUL). Decerease the in-degree of node connect to 3:"'FIND_NODE[5]"'=1,"'MODIFY_NODE[5,0]"';The zero degree node not explored is:['5']. Choose node 5 to explore:"'FIND_ALL_EDGES[5]"'=(2,NUL). Decerease the in-degree of node connect to 5:"'FIND_NODE[2]"'=1,"'MODIFY_NODE[2,0]"';The zero degree node not explored is:['2']. Choose node 2 to explore:"'FIND_ALL_EDGES[2]"'=(0,NUL). Decerease the in-degree of node connect to 2:"'FIND_NODE[0]"'=1,"'MODIFY_NODE[0,0]"';The zero degree node not explored is:['0']. Choose node 0 to explore:"'FIND_ALL_EDGES[0]"'=. Decerease the in-degree of node connect to 0:So the topological sorting is:### [6,1,4,7,3,5,2,0].

Triangle Max Sum

Question:A directed graph is represented as a table,node range from 0 to 3. Find the maximum sum of the weights of three interconnected nodes in the given graph table. Answer by a integer.
Answer: We use the triangle detection algorithm to find the maximum sum of the weights of three interconnected nodes in the given graph table. Initialize the maximum sum to 0. Find the edge of node 0:"'FIND_ALL_EDGES[0]"'=no edge,Because node 0 has no edge continue. Find the edge of node 1:"'FIND_ALL_EDGES[1]"'=(2,NUL)|(3,NUL),Traverse the node connected to node 1,find the node connected to node 2:"'FIND_ALL_EDGES[2]"'=(3,NUL)|(1,NUL),node 1 and node 2 are connected to node 3. Find the weight of node 1, node 2 and node 3:"'FIND_NODE[1]"'=2;"'FIND_NODE[2]"'=5;"'FIND_NODE[3]"'=9,so the sum of the weights is 2+5+9=16. Because 0 < 16, update the maximum sum to 16. The maximum sum of the weights of three interconnected nodes is ### 16.

A example of inference of ogbn-Arxiv

Question:Predict the label of the paper in Arxiv with the node id 26524. The label is one of the following: cs. AI(Artificial Intelligence);cs. AR(Hardware Architecture);.....
Answer:To predict the label of the node 26524, we need to BFS search 2-hop neighbors. Initial BFS queue: [('26524', 0)]Find the neighbors of node 26524:"'FIND_ALL_EDGES[26524]"'=[('99288', 'NUL')]. Find all the neigbors information. Find the similarity of node 26524 and node 99288:"'FIND_SIMILARITY[26524,99288]"'=0.75,0.75>0.5,node 99288 is related to node 26524 put into the BFS queue.Current BFS queue: [('99288', 1)],Find the neighbors of node 99288:"'FIND_ALL_EDGES[99288]"'=[('6150', 'NUL')]. Find all the neigbors information. Find the similarity of node 26524 and node 6150:"'FIND_SIMILARITY[99288,6150]"'=0.63,0.63>0.5,node 6150 is related to node 26524 put into the BFS queue.Current BFS queue: [('6150', 2)],After BFS search, we find the related nodes:[99288,6150],The details of target node 26524 is "'FIND_NODE[26524]"'=<Title>bounds and inequalities relating h index g index e index and generalized impact factor an improvement over existing models<KeyWords>index index generalized impact,citation record price medalists,rousseau power law model,The details of related nodes is:"'FIND_NODE[99288]"'=<Title>analysis of generalized impact factor and indices of journals<KeyWords>impact factor index,citation data,quality research published journals"'FIND_NODE[6150]"'=<Title>weighted indices for evaluating the quality of research with multiple authorship<KeyWords>indices weighted citation cuts,contributions individual authors,measure quality research. Node 26524's title and keywords focus on h-index, g-index, e-index, and generalized impact factor, indicating a strong connection to bibliometric indices. Its 2-hop neighbor 99288 also discusses generalized impact factor and indices, with similar keywords like impact factor and citation data. The other neighbor 6150 deals with weighted indices and quality research metrics. Both neighbors share thematic overlap with 26524, emphasizing citation analysis and research evaluation. Given the consistent focus on bibliometrics and digital tools for measuring research impact, the most plausible type for node 26524 is cs.DL (Digital Libraries), as this field encompasses such analytical methods. Answer:### cs.DL(Digital Libraries)

Disordered example of GPT-o1

Order 1

Question:Please find the shortest path between node 0 and node 29 in the following graph. The edges are:(19, 37)|(3, 5)|(0, 26)|(24, 38)|(8, 22)|(6, 21)|(5, 45)|(3, 41)|(39, 48)|(2, 35)|(21, 22)|(33, 48)|(31, 38)|(16, 32)|(1, 42)|(17, 40)|(13, 20)|(0, 30)|(40, 48)|(30, 42)|(5, 16)|(6, 8)|(16, 18)|(17, 49)|(28, 46)|(16, 48)|(4, 33)|(23, 47)|(20, 34)|(22, 40)|(31, 37)|(12, 18)|(35, 39)|(14, 33)|(29, 46)|(38, 49)|(0, 10)|(3, 49)|(29, 33)|(9, 27)|(25, 38)|(3, 47)|(26, 49)|(21, 30)|(33, 34)|(27, 28)|(10, 38)|(2, 34)|(9, 45)|(9, 41)|(7, 22)|(0, 47)|(14, 41)|(16, 31)|(3, 9)|(19, 23)|(1, 36)|(8, 25)|(10, 11)|(4, 23)|(1, 17)|(24, 31)|(26, 34)|(24, 44)|(16, 21)|(21, 42)|(42, 48)|(14, 17)|(1, 13)|(16, 25)|(34, 45)|(26, 32)|(9, 18)|(24, 43)|(4, 20)|(16, 44)|(11, 43)|(4, 14)|(20, 23)|(9, 26)|(14, 21)|(26, 30)|(25, 27)|(9, 19)|(28, 48)|(8, 32)|(15, 27)|(11, 27)|(7, 42)|(17, 21)|(8, 41)|(14, 44)|(24, 37)|(17, 27)|(15, 18)|(24, 47)|(4, 37)|(18, 23)|(32, 47)|(0, 13)|(1, 14)|(20, 41)|(8, 24)|(18, 47)|(16, 24)|(0, 21)|(17, 37)|(3, 43)|(32, 46)|(36, 44)|(16, 17)|(11, 25)|(7, 11)|(20, 43)|(6, 16)|(4, 8)|(10, 23)|(45, 48)|(25, 49)|(25, 47)|(15, 48)|(3, 32)|(10, 43)|(17, 48)|(25, 42)|(23, 24)|(1, 18)

Answer:Graph Edges (Revisited) Node 0 is connected to: 26, 30, 10, 47, 21, 13. Node 29 is connected to: 46, 33. Other relevant connections from nodes near 0 and 29: Node 30 connects to 42. Node 10 connects to 38, 23. Node 26 connects to 49, 34, 32. Node 21 connects to 42. Node 13 connects to 20. Steps for the Shortest Path: Starting from node 0: From node 0, you can go to node 26, node 30, node 10, node 47, node 21, or node 13. Let's start by exploring node 26 (as it seems to have relevant connections). From node 26: Node 26 connects to nodes: 49, 34, 32. Node 32 connects to 47. We'll explore node 32 next. From node 32: Node 32 connects to 47. From node 47, we can go to node 29. Shortest Path: 0 → 26 → 32 → 47 → 29 So, the shortest path from node 0 to node 29 is: 0 → 26 → 32 → 47 → 29.

Order 2

Question:Please find the shortest path between node 0 and node 29 in the following graph. The edges are:(26, 30)|(1, 42)|(24, 38)|(15, 18)|(38, 49)|(12, 18)|(20, 34)|(0, 30)|(17, 48)|(16, 32)|(15, 27)|(25, 49)|(6, 16)|(16, 25)|(33, 48)|(16, 44)|(3, 9)|(8, 22)|(25, 42)|(7, 11)|(23, 47)|(22, 40)|(16, 21)|(4, 14)|(32, 46)|(26, 32)|(16, 48)|(14, 44)|(9, 19)|(17, 21)|(20, 43)|(16, 31)|(15, 48)|(8, 25)|(11, 25)|(10, 43)|(19, 37)|(5, 16)|(16, 17)|(21, 30)|(0, 13)|(30, 42)|(17, 37)|(11, 27)|(45, 48)|(3, 43)|(9, 45)|(31, 38)|(27, 28)|(33, 34)|(14, 33)|(6, 21)|(10, 11)|(1, 13)|(17, 40)|(14, 21)|(21, 22)|(3, 49)|(10, 38)|(28, 46)|(39, 48)|(24, 37)|(0, 21)|(4, 37)|(21, 42)|(7, 42)|(23, 24)|(19, 23)|(5, 45)|(14, 17)|(25, 47)|(16, 24)|(25, 38)|(24, 47)|(17, 49)|(34, 45)|(14, 41)|(16, 18)|(25, 27)|(26, 49)|(1, 36)|(18, 47)|(32, 47)|(3, 47)|(36, 44)|(29, 33)|(24, 31)|(0, 47)|(29, 46)|(9, 18)|(9, 27)|(4, 23)|(17, 27)|(24, 43)|(1, 17)|(18, 23)|(4, 33)|(20, 23)|(10, 23)|(3, 5)|(3, 41)|(28, 48)|(0, 10)|(9, 26)|(42, 48)|(0, 26)|(4, 20)|(40, 48)|(4, 8)|(3, 32)|(24, 44)|(7, 22)|(35, 39)|(8, 32)|(31, 37)|(6, 8)|(2, 34)|(1, 18)|(8, 24)|(20, 41)|(26, 34)|(13, 20)|(9, 41)|(11, 43)|(1, 14)|(8, 41)|(2, 35)

Answer:To find the shortest path between node 0 and node 29 in this graph, we can use the Breadth-First Search (BFS) algorithm, which is ideal for finding the shortest path in an unweighted graph like this one. Here's how we can do it step by step:Step 1: Graph RepresentationThe edges provided represent an undirected graph, meaning each edge connects two nodes, and we can traverse in both directions. We first need to convert the list of edges into an adjacency list. Each node will have a list of its neighboring nodes. Step 2: BFS AlgorithmStart from node 0. Explore all neighbors of the current node. If we reach node 29, we stop and backtrack to find the shortest path. Use a queue to manage the nodes to explore and keep track of visited nodes to avoid revisiting them. Step 3: BFS to Find the Shortest PathNow we perform BFS to find the shortest path from node 0 to node 29. Starting from node 0, we explore the neighbors, track visited nodes, and keep a queue to explore new nodes. Here's the shortest path from node 0 to node 29:Start at node 0. Neighbors of 0: [30, 13, 21, 10, 47, 26]Explore node 30 → Neighbors: [26, 32, 42] → We don't see node 29 yet. Explore node 26 → Neighbors: [30, 32, 49, 34] → No node 29 yet. Explore node 10 → Neighbors: [43, 11, 23, 38] → No node 29 yet. Finally, explore node 47 → Neighbors: [23, 24, 25, 29] → Node 29 is reached. The shortest path is:0 → 47 → 29Thus, the shortest path from node 0 to node 29 is:0 → 47 → 29

Table 13: Comparison of Talk-like-a-Graph, GraphWiz, and GraphMind Methods on Medium task

	Talk like a graph	GraphWiz	GraphMind (Ours)
(Medium) Triangle	*Q:* In an undirected graph, `[i,k]` means node *i* has weight *k*, `(i,j)` means connection. Weights: `[0,3] [1,1] [2,2] [3,1]`, edges: `(0,3) (0,2) (2,3)`. Calculate max sum weight of three nodes. *A:* The maximum sum of the weights of any three nodes is 6 (from nodes 0, 1, 2 or 0, 2, 3).	*Q:* Find max sum weight of three interconnected nodes. Weights: `[0,3] [1,1] [2,2] [3,1]`, edges: `(0,3) (0,2) (2,3)`. What is the max sum? *A:* The graph consists of nodes 0 to 3, each with respective weights of 3, 1, 2, and 1. Examining the edges, we see that Nodes 0, 2, and 3 are the only trio of interconnected nodes, connected by the edges (0, 3), (0, 2), and (2, 3). The sum of their weights is «3 (Node 0) + 2 (Node 2) + 1 (Node 3) = 6». There are no other groups of three interconnected nodes in this graph. Therefore, the maximum sum of the weights of three interconnected nodes in this graph is 6. ### 6.	*Q:* Directed graph table (nodes 0–3). Find max sum weight of three interconnected nodes. Answer integer. *A:* We use the triangle detection algorithm to find the maximum sum of the weights of three interconnected nodes in the given graph table. Initialize the maximum sum to 0. Find the edge of node 0:"'FIND_ALL_EDGES[0]"' =(3,NUL)|(2,NUL),Traverse the node connected to node 0,find the node connected to node 3:"'FIND_ALL_EDGES[3]"' =(0,NUL)|(2,NUL),node 0 and node 3 are connected to node 2. Find the weight of node 0, node 3 and node 2:"'FIND_NODE[0]"' =3;"'FIND_NODE[3]"' =1;"'FIND_NODE[2]"' =2,so the sum of the weights is 3+1+2=6. Because 0 < 6, update the maximum sum to 6. The maximum sum of the weights of three interconnected nodes is ### **6**.

Table 14: Comparison of Talk-like-a-Graph, GraphWiz, and GraphMind Methods on hard task

	Talk like a graph	GraphWiz	GraphMind (Ours)
(Difficult) Hamilton cycle	*Q:* In an undirected graph, (i,j) means that node i and node j are connected with an undirected edge. G describes a graph among nodes 0, 1, 2. The edges in G are: (0, 1) (1, 2). Is there a Hamiltonian cycle in this graph? *A:* No, this graph doesn't have a Hamiltonian cycle. Here's why: The graph is just a straight line: 0 — 1 — 2. To have a Hamiltonian cycle, you'd need to visit all three nodes in a loop—starting and ending at the same node without retracing any edges or revisiting nodes (except the starting one at the end). But in this case:If you go 0 → 1 → 2, you can't get back to 0 because there's no edge between 2 and 0. Any other path either misses a node or has to backtrack, which isn't allowed. So, no, there's no way to form a Hamiltonian cycle here.	*Q:* Determine whether or not there is a Hamiltonian cycle in an undirected graph. In an undirected graph, (i,j) means that node i and node j are connected with an undirected edge. Given a graph, you need to output Yes or No, indicating whether there is a Hamiltonian path in the graph. Q: The nodes are numbered from 0 to 2, and the edges are: (0, 1) (1, 2). Is there a Hamiltonian cycle in this graph? *A:* To determine if a Hamiltonian cycle exists in an undirected graph, we need to check if there's a cycle that visits each node exactly once. In this graph, there are only three nodes and two edges, forming a triangle. To visit all three nodes, we can start from any node and follow the edges in either direction, as long as we do not repeat any node. For example, one possible Hamiltonian path is: [0, 1, 2]. Another possible Hamiltonian path is: [2, 1, 0]. Therefore, there is a Hamiltonian path in this graph. ### Yes, [0, 1, 2] or [2, 1, 0].	*Q:* A undirected graph is represented as a table. Node range from 0 to 2.<Q>:Determine whether the graph has a Hamiltonian cycle. Answer by a Yes or No. *A:* We will use the following steps to determine whether the graph has a Hamiltonian cycle. Step 1:count the degree of each node. Find the edge of node 0 in the graph:"'FIND_ALL_EDGES[0]"' =(1,NUL). The degree of node 0 is 1,modify node 0:"'MODIFY_NODE[0,1]"'. Find the edge of node 1 in the graph:"'FIND_ALL_EDGES[1]"' =(2,NUL)|(0,NUL). The degree of node 1 is 2,modify node 1:"'MODIFY_NODE[1,2]"' . Find the edge of node 2 in the graph:"'FIND_ALL_EDGES[2]"' =(1,NUL). The degree of node 2 is 1,modify node 2:"'MODIFY_NODE[2,1]"'. Step 2:Sort the degree of node. Find the degree of node:"'FIND_NODE[0]"' =1;"'FIND_NODE[1]"' =2;"'FIND_NODE[2]"'=1. Sort the degree of node:[1,1,2]. Step 3:Check the smallest degree of node. The smallest degree of node is 1(node 0). The smallest degree of node is less than 1, we need to walk from node 0. We will attempt to find a Hamiltonian path starting from node 0 . Attempt : Start from node 0. Find the edge of node 0 in the graph:"'FIND_ALL_EDGES[0]"' =(1,NUL). Unvisited nodes are: ['1']. Choose a random node from the unvisited nodes:['1']. Find the edge of node 1 in the graph:"'FIND_ALL_EDGES[1]"' =(2,NUL)|(0,NUL). Unvisited nodes are: ['2', '0']. Choose a random node from the unvisited nodes:['2', '0']. Find the edge of node 0 and node 2 in the graph:"'FIND_EDGE[0,2]"'=edge not found. The first node and the last node are not connected,Attempt failed. Answer is ###**No**.