Can Large Language Models Analyze Graphs like Professionals? A Benchmark, Datasets and Models

Xin Li[1]*, **Weize Chen**[2*], **Qizhi Chu**[1], **Haopeng Li**[3], **Zhaojun Sun**[3], **Ran Li**[2], **Chen Qian**[2]
Yiwei Wei[4], **Zhiyuan Liu**[2,5], **Chuan Shi**[1], **Maosong Sun**[2,5], **Cheng Yang**[1†]

[1] School of Computer Science, Beijing University of Posts and Telecommunications,
[2] Department of Computer Science and Technology, Tsinghua University,
[3] School of Artificial Intelligence, Beijing University of Posts and Telecommunications,
[4] College of Petroleum Engineering, China University of Petroleum (Beijing) at Karamay,
[5] Institute for Artificial Intelligence, Tsinghua University

lixin4sky@bupt.edu.cn, chenwz21@mails.tsinghua.edu.cn, yangcheng@bupt.edu.cn

Abstract

The need to analyze graphs is ubiquitous across various fields, from social networks to biological research and recommendation systems. Therefore, enabling the ability of large language models (LLMs) to process graphs is an important step toward more advanced general intelligence. However, current LLM benchmarks on graph analysis require models to directly reason over the prompts describing graph topology, and are thus limited to small graphs with only a few dozens of nodes. In contrast, human experts typically write programs based on popular libraries for task solving, and can thus handle graphs with different scales. To this end, a question naturally arises: *can LLMs analyze graphs like professionals*? In this paper, we introduce ProGraph, a manually crafted benchmark containing 3 categories of graph tasks. The benchmark expects solutions based on programming instead of directly reasoning over raw inputs. Our findings reveal that the performance of current LLMs is unsatisfactory, with the best model achieving only 36% accuracy. To bridge this gap, we propose LLM4Graph datasets, which include crawled documents and auto-generated codes based on 6 widely used graph libraries. By augmenting closed-source LLMs with document retrieval and fine-tuning open-source ones on the codes, we show 11-32% absolute improvements in their accuracies. Our results underscore that the capabilities of LLMs in handling structured data are still under-explored, and show the effectiveness of LLM4Graph in enhancing LLMs' proficiency of graph analysis. The benchmark, datasets and enhanced open-source models are available at https://github.com/BUPT-GAMMA/ProGraph.

1 Introduction

Background. Large language models (LLMs) [6, 52, 3] are parameter-rich neural networks trained on a vast amount of text data to understand and generate human language. LLMs can not only handle classical natural language processing tasks like translation, but also benefit task solving in various domains such as code generation [41], logical reasoning [39], and mathematical calculation [43].

Previous LLM Benchmarks for Graph Analysis. Recently, many researchers have proposed extending LLMs to scenarios that require graph understanding and analysis [25, 48]. As graph is

*Equal Contribution.
†Corresponding author.

38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks.

 https://doi.org/10.52202/079017-4478

Table 1: Comparisons among different graph analysis benchmarks for LLMs.

Aspects	ProGraph (this work)	NLGraph ([55])	LLM4DyG ([64])	GraphTMI ([15])	GraphInstruct ([33])	GPT4Graph ([20])	GraphWiz ([9])
Basic Graph Theory	✓	✓	✓	✓	✓	✓	✓
Graph Statistical Learning	✓	✗	✗	✓	✗	✓	✗
Graph Embedding	✓	✓	✗	✗	✗	✗	✗
Access to External APIs	✓	✗	✗	✗	✗	✗	✗
Real-world Context	✓	✗	✗	✗	✗	✗	✗
Scalability	up to 10^6	up to 10^1	up to 10^1	up to 10^2	up to 10^1	up to 10^1	up to 10^2

a very commonly used data structure in real-world services (*e.g.,* social networks [50, 35, 54] and urban computing [30, 66, 62]), enabling the ability of LLMs to process graphs is an important step toward more advanced general intelligence. To this end, several benchmarks have been developed to evaluate such ability. For example, NLGraph [55] and GraphInstruct [33] investigate whether LLMs can understand and compute basic graph properties, such as counting node degrees or finding the shortest path for a node pair. GraphTMI [15] and GPT4Graph [20] also consider typical learning tasks such as node classification. LLM4DyG [64] further extends the tasks to dynamic graphs.

Limitations of Existing Benchmarks. However, from the perspective of practicality, we argue that previous benchmarks have three major drawbacks. Firstly, the problems in these work require LLMs to read through the adjacency lists of graphs from prompts before answering specific questions. Consequently, the graph sizes in their benchmarks are rather small (typically with a few dozens of nodes), due to the length limitation of LLMs. Being able to compute the shortest path on a small graph does not mean that the same can be done on a real graph with millions of nodes. Secondly, the desired solving process in these work requires step-by-step reasoning fully based on LLMs. But even with the help of Chain-of-Thought (CoT) [58, 14], the reasoning depths of current LLMs are still shallow [29, 27]. Consequently, LLMs might be able to count triangles one by one in a small graph with 10 nodes, and will inevitably fail for large graphs. Thirdly, the problem descriptions in these work are abstract and monotonous in form, lacking context from real-world application scenarios.

Inspirations from Human Experts. Consider the scenario that a human expert is asked to find the shortest path between two nodes in a million-scale graph, she will probably write a few lines of Python codes based on NetworkX[23], instead of directly reasoning over the raw inputs. To this end, a question naturally arises: *can LLMs analyze graphs like professionals*? Fortunately, most popular LLMs have shown the ability to write codes and utilize various application programming inferfaces (APIs), making it possible to analyze graphs via API calling as human experts will do. Compared with direct reasoning in previous benchmarks, generating a few lines of codes requires much shallower reasoning depths for LLMs, but can solve more complex problems.

Benchmark. In this paper, we propose ProGraph benchmark to evaluate the capability of LLMs in leveraging external APIs for graph analysis. The benchmark includes 512 problems with hand-crafted questions and answers (QA pairs). The problems cover three categories of tasks: basic graph theory, graph statistical learning, and graph embedding, and can be solved based on six popular Python libraries. In the questions, graphs can be either described by natural language or stored in files, and thus can scale to 10^6 in our benchmark. To improve the diversity of problem descriptions and align with real-world scenarios, the questions are rephrased in a role-play manner based on GPT-4 turbo [37]. In the answers, we provide reference code, key APIs and execution results. We also design an automated evaluation process that aligns well with human judgement.

Datasets and Models. To facilitate LLMs to solve these problems via programming, we construct the LLM4Graph datasets with both document and code data. The document dataset contains API information crawled from the official documents of the six Python libraries. The code dataset includes 29,260 QA pairs automatically generated by back-instructing [56] GPT-4 turbo. To enable CoT learning, we further introduce the thought on the document information of relevant APIs to the answers of the code dataset as prefixes. To demonstrate the value of our datasets, we enhance closed-source LLMs by extracting relevant information from the document dataset as RAG (retrieval-augmented generation), and improve open-source ones by instruction tuning over the code dataset. Besides the datasets, the improved open-source LLMs are also released for future researches.

Key Results. The accuracies of closed-source models (Claude, GPT and Gemini) on ProGraph are 25-36%, and can be improved to 37-46% with RAG using LLM4Graph as the retrieval pool. The accuracies of open-source models (Llama3 [1] and Deepseek Coder [19]) are only 12-24%, but can be

improved to 45-47% through instruction-tuning on LLM4Graph. These results show that ProGraph is challenging for current LLMs, and LLM4Graph can significantly enhance their performance.

Contributions. (1) To the best of our knowledge, we are the first work exploring the ability of LLMs to analyze graphs with external APIs. The utilization of external APIs is practical and powerful in real-world scenarios. (2) To evaluate such ability, we propose a novel and challenging ProGraph benchmark with hand-crafted QA pairs covering three categories of tasks, *i.e.,* basic graph theory, graph statistical learning, and graph embedding. (3) We develop LLM4Graph datasets containing both crawled document data and auto-generated code data. Experimental results demonstrate that our datasets can substantially enhance the performance of both closed-source and open-source LLMs. The improved open-source models are released together with the datasets for future researches.

2 Related Work

LLM for Graphs. Recent efforts leverage the strong generalization ability of LLMs for graph understanding and analysis. Benchmarks like NLGraph [55] and GraphInstruct [33] evaluate LLMs' graph reasoning abilities, finding that while LLMs have some capabilities, they are not very strong. GraphTMI [15] assesses LLMs' performance on tasks like node classification and link prediction using different input formats. GPT4Graph [20] evaluates LLMs' understanding of graph structure data in various formats, and LLM4DyG [64] studies LLMs in dynamic graphs, introducing techniques to improve performance. GraphWiz [9] employs two approaches to optimize reasoning paths for solving basic graph problems. Another approach combines LLMs with graph neural networks (GNNs) to enhance learning tasks, leveraging text attribute processing or direct graph task handling through techniques like prompt learning and instruction tuning [25, 17, 10, 59, 48, 24, 49, 63, 7]. However, these works are often specialized for classification tasks and do not handle complex graph tasks.

LLM Benchmarks. LLMs perform strongly in text processing, mathematical computation, and code generation. Text processing benchmarks evaluate capabilities in machine translation, summarization, and comprehension [2, 34, 61, 32, 16, 26]. Mathematical computation benchmarks assess understanding of numbers and elementary math concepts [12, 51]. Specialized field benchmarks, such as those in biology, chemistry, and finance, evaluate LLMs' expertise in specific domains [47, 21, 60]. Code generation benchmarks like HumanEval and MBPP focus on function-oriented tasks [8, 4], while our proposed ProGraph considers task-oriented code generation.

Enhancement Techniques for LLMs. There are many techniques to enhance the performance of LLMs [38, 57]. Among these, two important techniques are Chain-of-Thought (CoT) [58] and retrieval-augmented generation (RAG) [22]. CoT allows the model to mimic human thinking by reasoning step-by-step rather than directly providing an answer, significantly enhancing the logical analysis and reasoning capabilities of LLMs. RAG reduces the LLMs' hallucinations by allowing them to access relevant information before generating answers, improving LLMs' accuracy and reliability across various tasks.

3 Benchmark

To evaluate the ability of LLMs in graph analysis, we introduce the ProGraph benchmark, featuring 512 problems in three categories. These hand-crafted problems include questions and answers with two difficulty levels. To enhance diversity and realism, we leverage GPT-4 turbo to rephrase the questions in a role-playing manner, followed by manual verification for correctness. Finally, given the high consistency of answer judgments between humans and GPT-4o, we employ GPT-4o to automate the evaluation. We compare the proposed benchmark with previous ones in Table 1, present more discussions about related work in Appendix 2, and show a benchmark example in Appendix E.

3.1 Tasks

The proposed ProGraph benchmark considers three categories of tasks:

Category 1: Basic Graph Theory. Graph theory primarily studies the fundamental concepts of graphs, including types, properties, classical algorithms and many other basic operations. For example, some problems in this category aim to check whether a graph is acyclic, compute the centrality of nodes, or find the maximum cardinality matching.

 https://doi.org/10.52202/079017-4478

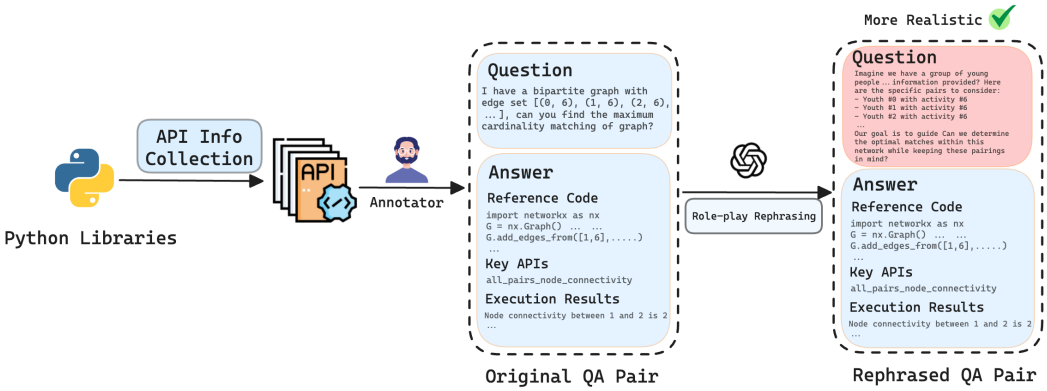


Figure 1: **The pipeline of ProGraph benchmark construction.** We first collect the API documents of 6 frequently-used graph reasoning Python libraries. Then human annotators read the API information, use random graph generators to generate graph data, write questions based on API information, and give the results and the corresponding python codes, forming the Original QA pairs. Finally, GPT-4 rephrases abstract questions and integrates them into real-world scenarios.

Category 2: Graph Statistical Learning. Graph statistical learning utilizes a probabilistic model to extract useful information from nodes, edges, or the entire topology for various tasks. In this work, we mainly focus on graph clustering and sampling techniques. For example, detecting communities in a graph with Louvain algorithm [5], or sampling a subgraph based on random walk.

Category 3: Graph Embedding. Graph embedding technique aims to learn real-valued vectors for nodes in a graph, where similar nodes are expected to have similar vectors. The learned vectors can be used as features to enhance the performance of downstream tasks. For example, a typical problem in this category will require the learned embeddings of DeepWalk algorithm [40] for a given graph.

Table 2: Statistics of ProGraph.

	Question Type				Answer Difficulty	
	True/False	Calculation	Drawing	Hybrid	Easy	Hard
Basic Graph Theory	32	240	25	15	257	55
Graph Statistical Learning	7	115	7	25	43	111
Graph Embedding	0	30	0	16	0	46
Total	39	385	32	56	300	212

3.2 Human Annotation

To create high-quality problems for benchmarking, we invite human annotators to develop questions and answers based on the following guidelines, and the annotation manual is in Appendix I.

Python Libraries. In this work, we consider six popular libraries to support the above three task categories, *i.e.,* NetworkX [23] and igraph [13] for basic graph theory, CDlib [44], graspologic [11] and Little Ball of Fur [46] for graph statistical learning, and Karate Club [45] for graph embedding.

Question Design. First of all, human annotators need to either manually design or use some random graph generators to obtain a graph. Then, based on the API documents of the six libraries, the annotators are asked to design questions with one of the four types: true/false, calculation, drawing, and hybrid questions. Here hybrid questions are composed of two or more of the first three types. Here we present a calculation question from the basic graph theory category as an example:

Question
I have a graph with an edge set [(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)], can you help me compute node connectivity for all pairs of nodes and print the node connectivity for each pair?

Reference Code
** Python Code **
Key APIs
all_pairs_node_connectivity
Execution Results
Node connectivity between 1 and 2 is 2
...

Answer Construction. Based on the proposed question, human annotators need to further provide the code, the number of involved APIs, and the execution result. Based on the number of APIs, we categorize the problems into two difficulty levels: easy level involves one API, while the hard level involves multiple APIs. Here's an example of the collected data for the above question:

3.3 Role-Play Rephrasing

To make questions more realistic, we rephrase them in a role-play manner. First, GPT-4 turbo generates hundreds of job titles and descriptions. Then, we randomly select a job for a given problem. Using GPT-4 turbo, we mimic the profession's characteristics, knowledge, and style to integrate the question into a real-world context. We manually review the modified questions to ensure they maintain the same semantics and graph structures as the original ones. A complete example of rephrasing is in Appendix B, and here we present the rephrased question of the above problem:

Rephrased Question
We're examining a simplified model of an ecosystem where [...], we've mapped out a series of interactions as follows: [(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)]. [...] Can we analyze our network to reveal the minimum number of species that would need to be removed to disrupt the direct connection between any two species in this web? [...]

3.4 Automated Evaluation

Metrics. To evaluate the ability of LLMs to solve these problems, we consider two metrics: pass rate and accuracy. Pass rate measures the ratio of executable code generated by an LLM, while accuracy measures the ratio of correct answers from the executable code. Accuracy is always no higher than the pass rate and is considered the more important metric as it evaluates final performance.

Process. Evaluating diverse answer formats with rule-based matching is challenging, and human evaluation is too labor-intensive. Thus, we automate evaluation using GPT-4o. First, we extract code snippets from LLM-generated answers using regular expressions. GPT-4o is then asked to check the correctness given the execution result. For problems with certain answers, such as true/false or calculation questions, GPT-4o assigns 1 point if the execution result matches the reference code's result, and 0 otherwise. For other problems, like drawing questions, GPT-4o matches key API usage: if the generated code contains m out of n key APIs, the accuracy point is m/n.

Rationale. To validate GPT-based evaluation, we measure its stability (self-consistency) and alignment with human judgments (human-consistency). Higher stability means judgment scores are consistent across multiple evaluations, while higher human alignment indicates better quality. We use the agreement metric [65] to assess these consistencies. For n evaluations of the same answer, we take the highest number of evaluations m that received the same score and divide it by n to get the consistency. Self-consistency is the agreement among three GPT-4o evaluations, and human-consistency is the agreement between one GPT-4o evaluation and a manual evaluation. We evaluate all 512 problems with answers from *Claude 3 Opus RAG 7*, the best-performing closed-source model which will be introduced in Section 4.2, and present the results in Table 3, showing high self-consistency and human-consistency.

Table 3: Self-Consistency (SC) and Human-Consistency (HC) of automated evaluation.

	SC (%)	**HC** (%)
Category 1	98.9	97.3
Category 2	98.9	96.4
Category 3	98.6	97.8
Overall	98.9	97.5

4 Datasets and Models

To enhance the ability of LLMs to solve graph problems with Python APIs, we construct LLM4Graph datasets. Based on the LLM4Graph, we enhance both closed-source and open-source models.

4.1 Datasets

Document dataset. The document dataset is crawled from the official documents of the corresponding Python libraries. These documents can be directly used to enhance closed-source models via RAG.

 https://doi.org/10.52202/079017-4478

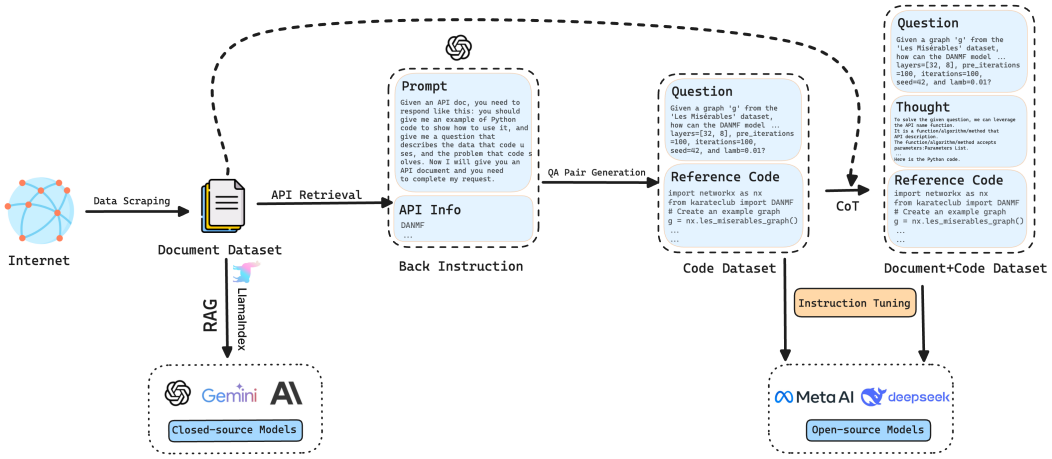


Figure 2: **The pipeline of LLM4Graph dataset construction and corresponding model enhancement.** We build the LLM4Graph dataset to improve the capabilities of LLMs in solving graph reasoning tasks through API calls. For the document dataset, we collect API documentation from the Internet. For code dataset, we automatically generate questions and corresponding codes with GPT-4 and API documentation. And for "doc+code" dataset, inspired by CoT reasoning, we add thought for each question, and combine the API documentation and the code dataset for construction.

We also use these documents for generating code datasets. Specifically, each API corresponds to a specific entry of the document dataset, including the description, parameter list, return value list, code examples, and other contents. An example of API entry is shown in Appendix C.

Code datasets. We construct two code datasets in the form of QA pairs. The questions in both datasets are the same, but the answers differ. In the simpler dataset, each answer only contains Python code. Inspired by Chain of Thought (CoT) [58], each answer in the more complex dataset additionally includes relevant APIs and their documents as prefixes. This modification can facilitate open-source models to utilize document information more effectively. We name the above code datasets as Code (QA) and Doc+Code (QA), respectively. Unlike the hand-crafted benchmark, problems in the code datasets are automatically generated and each contains only one key API.

Specifically, we first randomly select an API from the six Python libraries, and then employ GPT-4 turbo to generate example code for the API along with a corresponding question via back instruction [56]. In the Code (QA) dataset, each answer only contains the Python code in the generated json. In the Doc+Code (QA) dataset, we design a template to additionally incorporate the API document information into each answer. This allows a CoT process that first selects the possibly needed APIs, then provides the corresponding API information, and finally writes code to solve the problem. Besides, the questions in both code datasets need to undergo the role-play processing in Section 3.3 for diverse problem descriptions. The prompt for back instruction and an example of the Doc+Code dataset can be found at Appendix D and F:

Table 4: Statistics of LLM4Graph datasets.

	Document	**Code (QA)**	**Doc+Code (QA)**
Category 1	1,115	23,324	23,324
Category 2	253	5,136	5,136
Category 3	45	800	800
Total	1,413	29,260	29,260

4.2 Models

We use the above datasets to improve various LLMs in handling graph analysis tasks. For closed-source models, we enhance them by retrieving relevant information from the document dataset as RAG. For open-source models, we fine-tune them using code datasets as instruction tuning.

Table 5: Performance (%) of different models on ProGraph.

	Basic Graph Theory		Graph Statistical Learning		Graph Embedding		Overall	
Model	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy
Claude 3 Haiku	52.9	31.6	23.4	9.7	32.6	2.9	42.2	22.4
Claude 3 Sonnet	57.1	33.2	15.6	4.6	10.9	0.0	40.4	21.6
Claude 3 Opus	69.2	47.3	31.2	15.1	**47.8**	**14.5**	55.9	34.7
GPT-3.5	64.1	35.1	24.7	8.4	15.2	1.1	47.9	24.0
GPT-4 turbo	**72.4**	42.1	39.0	14.8	41.3	12.0	59.6	31.2
GPT-4o	69.9	**48.1**	**48.7**	**21.4**	32.6	5.8	**60.2**	**36.3**
Gemini 1.0 Pro	48.7	27.7	9.1	1.7	19.6	3.3	34.2	17.7
Gemini 1.5 Pro	59.6	37.2	21.4	6.6	13.0	1.8	43.9	24.8
Llama 3	36.5	17.3	12.3	3.8	15.2	0.4	27.3	11.7
Deepseek Coder	56.1	33.8	30.5	9.8	30.4	7.6	46.1	24.2

Closed-source Models. Due to the high difficulty of our ProGraph benchmark, mainstream LLMs (including Claude, GPT and Gemini) are not particularly strong in directly solving these problems. Therefore, before closed-source LLMs answer these questions, we retrieve the document information of top-K relevant APIs based on Llamaindex [31], allowing the models to learn through the context and enhance their performance. We explore $K = 3, 5, 7$ to investigate the impact of RAG, and the models will be given the corresponding suffix as No RAG, RAG 3, RAG 5 or RAG 7.

Open-source Models. We use the two code datasets (*i.e.,* Code and Doc+Code) to fine-tune Llama-3-8B-Instruct [1] and Deepseek-Coder-7B-instruct [19], and consider the following model variants: (1) Code-only: LLMs are fine-tuned with the Code dataset. (2) Code + RAG 3/5/7: Code-only models are further equipped with RAG as closed-source ones. (3) Doc+Code: LLMs are fine-tuned with the Doc+Code dataset, and conduct inference by a corresponding two-step CoT reasoning process. In the first step, the model generates potential APIs based on the question. In the second step, we retrieve the API information and then provide the question along with the API information back to the model. The model then completes the remaining reasoning by writing code to solve the problem. The third group of models can maximize the use of document information to enhance the performance of open-source models, and significantly narrow the performance gap with closed-source large models.

5 Experiment

In this section, we present a comprehensive evaluation of our proposed benchmark, ProGraph, and the accompanying dataset, LLM4Graph. Our experiments assess the performance of both closed-source and open-source LLMs on graph analysis tasks. We demonstrate the limitations of current LLMs in handling structured graph data, and showcase the potential improvements achievable through the use of our datasets via RAG or instruction tuning. More results are shown in Appendix G and H.

5.1 Experiment Settings and Baselines

In this work, we fine-tune two open-source models, Llama-3-8B-Instruct [1] and Deepseek-Coder-7B-Instruct-v1.5 [18], using the LLM4Graph dataset. For all models, we use alignment-handbook framework [53], set the learning rate, epochs, and max length as 2e-4, 4, and 4096, running on NVIDIA 8*A100 GPUs (40GB). The training batch size is set to 1 and evaluation batch size is set to 4. For testing, we set temperature as 0 and maximum output tokens as 4096, ensuring the stable and reasonable generation.

As our baselines, we employ models without RAG, including the GPT series (GPT-3.5, GPT-4 turbo, GPT-4o) [36], Claude 3 series (Haiku, Sonnet, Opus) [3], and Gemini Pro (1.0 and 1.5) [42], along with non-fine-tuned open-source smaller models Llama-3-8B-Instruct [1] and Deepseek-Coder-7B-Instruct-v1.5 [18]. The detailed results are presented in Table 5.

5.2 Benchmarking LLMs on ProGraph

We evaluate a variety of mainstream closed-source and open-source LLMs on the ProGraph, including GPT [36], Claude [3], Gemini [42], Llama 3 [1] and Deepseek Coder [18].

https://doi.org/10.52202/079017-4478

The results, presented in Table 5, indicate that none of the tested LLMs could effectively solve the problems in ProGraph. While GPT-4 turbo and its variant GPT-4o demonstrate relatively higher performance with an overall accuracy of 31.2% and 36.3% respectively, they still fall short in delivering satisfying accuracy across different categories. For instance, GPT-4 turbo achieves an accuracy of 42.1% in Category 1 but only 14.8% and 12.0% in Categories 2 and 3, respectively. Similar trends are observed in other models. These results highlight the challenges faced by current LLMs in addressing structured graph data as human experts. This necessitates the development of specialized datasets and fine-tuning approaches to bridge this performance gap.

5.3 Enhancing Closed-Source LLMs with Document Dataset

To investigate the potential of enhancing LLMs' performance on graph analysis tasks, we utilize RAG techniques to extract relevant API information from the documents in LLM4Graph. This augmented context is then provided to the LLMs to assist in generating more accurate and effective solutions.

Figure 3 shows the performance gains for four top closed-source LLMs: GPT-4 turbo, GPT-4o, Claude 3 Opus, and Gemini 1.5 Pro. All models show significant boosts in pass rate and accuracy, with more than a 5% improvement in accuracy. Gemini 1.5 Pro and Claude 3 Opus achieve over 10% accuracy improvement with five retrieved APIs. However, performance improvements plateau with additional API information, which may be attributed to the saturation of relevant information, where additional API details no longer contribute to further understanding and instead introduce redundancy or noise. This observation aligns with the findings of previous studies on RAG [28].

The effectiveness of the LLM4Graph in enhancing LLM capabilities on graph analysis tasks is evident from these results. By incorporating a RAG mechanism with the LLM4Graph, we demonstrate that it is possible to substantially improve the performance of closed-source LLMs without the need for extensive fine-tuning or architectural modifications. This approach offers a practical and efficient solution for adapting existing LLMs to handle structured graph data more effectively.

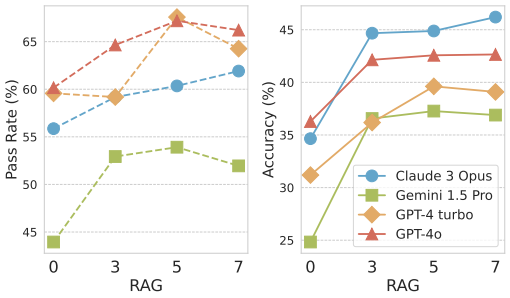


Figure 3: The pass rate (left) and accuracy (right) of closed-source models with RAG.

5.4 Enhancing Open-Source LLMs with Code Dataset and Doc+Code Dataset

We investigate the potential of enhancing open-source LLMs' performance on graph analysis tasks by fine-tuning them using LLM4Graph and augmenting the models with RAG. Experiments are conducted on a general-purpose model, Llama-3-8B-Instruct [1], and a model specifically designed for code generation, Deepseek-Coder-7B-Instruct-v1.5 [18]. The results, presented in Figure 4, demonstrate that our LLM4Graph can significantly improve the performance of different types of open-source small models. The accuracy of both models, after being fine-tuned merely on the code data within LLM4Graph, substantially surpasses the best result from closed-source models without RAG. The Doc+Code setting further enhances the models' performance, leading to comparable or even better accuracy than the best result from closed-source model with RAG.

However, augmenting the open-source models fine-tuned on the code with RAG mechanism does not further improve the performance, and even leads to degraded performance. We hypothesize that this discrepancy may be attributed to the limited ability of these small models to process long text, hindering their utilization of the document information. The additional information provided by RAG may introduce confusion in understanding the problem statement and arriving at the correct solution. Overall, our proposed Doc+Code setting proves to be an effective means of integrating document information from LLM4Graph, significantly enhancing the accuracy of open-source models. We show that LLM4Graph can serve as effective data to enhance the model's code-generation capabilities and lead to better utilization of various graph libraries.

https://doi.org/10.52202/079017-4478

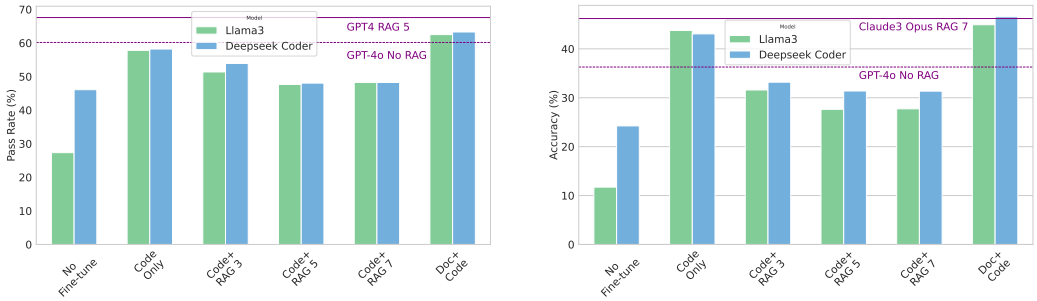
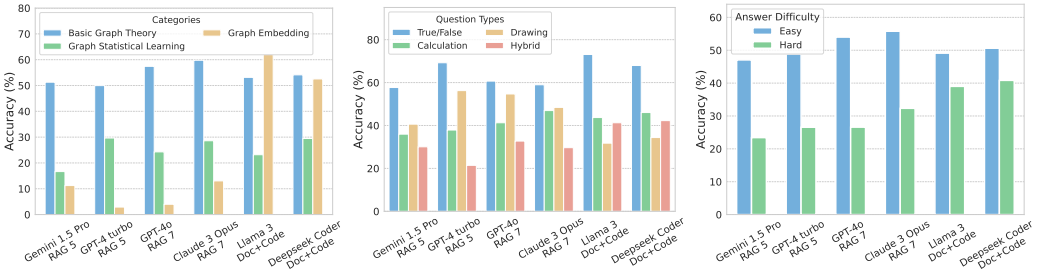


Figure 4: The pass rate (left) and accuracy (right) of open-source models with instruction tuning.



(a) Task Category (b) answer difficulty (c) question type

Figure 5: The performance of six best-performing models on different groupings of ProGraph.

6 Analysis

In this section, we present a comprehensive analysis of the performance of different models on the ProGraph. By grouping the benchmark based on categories, answer difficulties, and question type, we aim to provide a granular exploration of the strengths and limitations of these models in handling graph analysis tasks. We also present the types of compilation errors made by different models.

6.1 Performance Analysis on Different Benchmark Groupings

To gain a deeper understanding of the capabilities of LLMs and fine-tuned smaller models presented in Section 5.4, we analyze their performances on ProGraph from three different perspectives: task category, answer difficulty and question type.

Task Category. We analyze the model performance based on different categories in the ProGraph, as shown in Figure 5a. Mainstream LLMs and fine-tuned smaller models exhibit similar performance on graph theory and graph statistical learning tasks. However, a significant disparity is observed in their performance on graph embedding tasks, where fine-tuned smaller models substantially outperform RAG-enhanced large models. This observation suggests that not all graph analysis tasks can be easily handled by closed-source LLMs without further fine-tuning. More complex and challenging tasks still require fine-tuning for effective learning.

Answer Difficulty. We further examine the model performance based on different answer difficulties, *i.e.,* true/false, calculation, drawing, and hybrid. In Figure 5b, we plot the performance of different models on these answer difficulties separately. Mainstream LLMs excel in true/false and drawing types but struggle with calculation and hybrid ones. Fine-tuned smaller models demonstrate improvements across various answer difficulties, especially on the complex hybrid type, indicating the effectiveness of our proposed enhancement strategies.

Question Type. Lastly, we divide the ProGraph into two levels of difficulty: easy (involving only one API) and hard (involving multiple APIs). As shown in Figure 5c, mainstream closed-source large models perform well on easy-level problems, with accuracies generally approaching or exceeding 50%. However, their performance significantly deteriorates on hard-level ones, with the highest accuracy reaching only 32.5%. This observation suggests that mainstream LLMs have limitations when the number of required APIs increases. In contrast, our fine-tuned models demonstrate significantly higher accuracy on hard-level problems, approaching or exceeding 40%, yielding approximately an

https://doi.org/10.52202/079017-4478

improvement of 8% compared to the best closed-source LLM. Note that LLM4Graph only contains data involving one API. Still, the models fine-tuned on LLM4Graph show strong generalizability on problems requiring multiple APIs.

6.2 Compilation Error Analysis

To gain insights into the types of compilation errors made by different models, we conduct an error analysis on the best-performing closed-source models (GPT-4 turbo RAG 5 and Claude 3 Opus RAG 7) and fine-tuned open-source small models (DeepSeek Coder Doc+Code and Llama 3 Doc+Code). As shown in Figure 6, we categorize the errors into ten distinct types to identify patterns and differences in the error distributions among these models.

Our analysis reveals that closed-source models exhibit a low similarity in their error cases, suggesting that they possess varying coding capabilities. For instance, GPT-4 turbo often makes SyntaxError, but rarely ImportError, which is contrary to Claude 3 Opus. The fine-tuned open-source small models exhibit a high similarity in their error distributions, with AttributeError being the most dominant.

The error analysis also highlights some common challenges faced by all models, such as AttributeError and TypeError, suggesting that models may have difficulty in memorizing and understanding the attribute of class objects from various python libraries, and the type of returned results from different functions. Interestingly, the fine-tuned models have a notably lower percentage of SyntaxError compared to the closed-source models, indicating that further fine-tuning on the LLM4Graph helps the models learn better code syntax and structure.

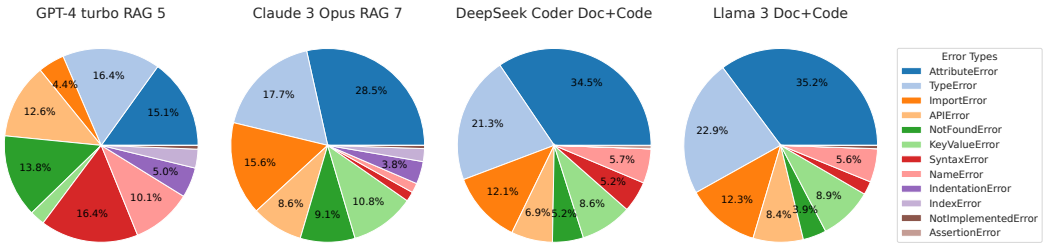


Figure 6: Compilation error statistics for four best-performing models.

7 Conclusion

In this paper, we introduce ProGraph, a novel and challenging benchmark for evaluating LLMs in graph analysis using external APIs. Current LLMs achieve only 36% accuracy, revealing their limitations. To bridge this gap, we further construct LLM4Graph, a dataset with crawled documents and auto-generated codes based on popular graph libraries. The datasets can help improve the accuracy of both closed-source and open-source LLMs by 11-32% through RAG and instruction tuning. Our work highlights the potential of enhancing LLMs with our LLM4Graph, offering valuable resources for advancing LLM capabilities in structured data analysis. Discussions about limitations and boarder impacts can be found in Appendix A.

8 Acknowledgements

This work is supported by the National Natural Science Foundation of China (No.62192784, 62236004), the National Key R&D Program of China (No.2022ZD0116312), Young Elite Scientists Sponsorship Program (No.2023QNRC001) by CAST, and Tsinghua University Initiative Scientific Research Program.

References

[1] AI@Meta. Llama 3 model card. 2024. URL `https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md`.

[2] Chantal Amrhein, Nikita Moghe, and Liane Guillou. ACES: Translation accuracy challenge sets for evaluating machine translation metrics. In *Seventh Conference on Machine Translation*

(WMT22), Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

[3] Anthropic. Announcing the claude 3 family, 2024. URL `https://www.anthropic.com/news/claude-3-family`.

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[5] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008:P10008, 2008.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[7] Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. Graphllm: Boosting graph reasoning ability of large language model, 2023.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

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

[10] Zhikai Chen, Haitao Mao, Hongzhi Wen, Haoyu Han, Wei Jin, Haiyang Zhang, Hui Liu, and Jiliang Tang. Label-free node classification on graphs with large language models (LLMs). In *The Twelfth International Conference on Learning Representations*, 2024.

[11] Jaewon Chung, Benjamin D. Pedigo, Eric W. Bridgeford, Bijan K. Varjavand, Hayden S. Helm, and Joshua T. Vogelstein. Graspy: Graph statistics in python. *Journal of Machine Learning Research*, 20(158):1–7, 2019.

[12] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems, 2021.

[13] Gábor Csárdi and Tamás Nepusz. The igraph software package for complex network research. 2006.

[14] Chengwei Dai, Kun Li, Wei Zhou, and Songlin Hu. Beyond imitation: Learning key reasoning steps from dual chain-of-thoughts in reasoning distillation, 2024.

[15] Debarati Das, Ishaan Gupta, Jaideep Srivastava, and Dongyeop Kang. Which modality should i use – text, motif, or image? : Understanding graphs with large language models, 2024.

 https://doi.org/10.52202/079017-4478

[16] Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *North American Chapter of the Association for Computational Linguistics*, 2019.

[17] Keyu Duan, Qian Liu, Tat-Seng Chua, Shuicheng Yan, Wei Tsang Ooi, Qizhe Xie, and Junxian He. Simteg: A frustratingly simple approach improves textual graph learning, 2023.

[18] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[19] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

[20] Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. Gpt4graph: Can large language models understand graph structured data ? an empirical evaluation and benchmarking, 2023.

[21] Taicheng Guo, Kehan Guo, Bozhao Nan, Zhengwen Liang, Zhichun Guo, N. Chawla, O. Wiest, and Xiangliang Zhang. What can large language models do in chemistry? a comprehensive benchmark on eight tasks. In *Neural Information Processing Systems*, 2023.

[22] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[23] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008. URL `http://conference.scipy.org/proceedings/SciPy2008/paper_2/`.

[24] Yufei He and Bryan Hooi. Unigraph: Learning a cross-domain graph foundation model from natural language, 2024.

[25] Xuanwen Huang, Kaiqiao Han, Dezheng Bao, Quanjin Tao, Zhisheng Zhang, Yang Yang, and Qi Zhu. Prompt-based node feature extractor for few-shot learning on text-attributed graphs, 2023.

[26] Mandar Joshi, Eunsol Choi, Daniel S. Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

[27] Ting En Lam, Yuhan Chen, Elston Tan, Eric Peh, Ruirui Chen, Paritosh Parmar, and Basura Fernando. Causalchaos! dataset for comprehensive causal action question answering over longer causal chains grounded in dynamic visual scenes, 2024.

[28] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html`.

[29] Jiachun Li, Pengfei Cao, Yubo Chen, Kang Liu, and Jun Zhao. Towards faithful chain-of-thought: Large language models are bridging reasoners, 2024.

[30] Zhonghang Li, Lianghao Xia, Jiabin Tang, Yong Xu, Lei Shi, Long Xia, Dawei Yin, and Chao Huang. Urbangpt: Spatio-temporal large language models, 2024.

[31] Jerry Liu. LlamaIndex, 11 2022. URL `https://github.com/jerryjliu/llama_index`.

[32] Zheheng Luo, Qianqian Xie, and Sophia Ananiadou. Factual consistency evaluation of summarisation in the era of large language models, 2024.

[33] Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, and Xing Xie. Graphinstruct: Empowering large language models with graph understanding and reasoning capability, 2024.

[34] Nikita Moghe, Arnisa Fazla, Chantal Amrhein, Tom Kocmi, Mark Steedman, Alexandra Birch, Rico Sennrich, and Liane Guillou. Machine translation meta evaluation through translation accuracy challenge sets, 2024.

[35] Seth A. Myers, Aneesh Sharma, Pankaj Gupta, and Jimmy Lin. Information network or social network? the structure of the twitter follow graph. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, page 493–498. Association for Computing Machinery, 2014. ISBN 9781450327459.

[36] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[37] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, and ... Gpt-4 technical report, 2024.

[38] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[39] Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-LM: Empowering large language models with symbolic solvers for faithful logical reasoning. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3806–3824, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.248. URL `https://aclanthology.org/2023.findings-emnlp.248`.

[40] Bryan Perozzi, Rami Al-Rfou, and Steven S. Skiena. Deepwalk: online learning of social representations. *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014.

[41] Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

[42] Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy P. Lillicrap, Jean-Baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, Ioannis Antonoglou, Rohan Anil, Sebastian Borgeaud, Andrew M. Dai, Katie Millican, Ethan Dyer, Mia Glaese, Thibault Sottiaux, Benjamin Lee, Fabio Viola, Malcolm Reynolds, Yuanzhong Xu, James Molloy, Jilin Chen, Michael Isard, Paul Barham, Tom Hennigan, Ross McIlroy, Melvin Johnson, Johan Schalkwyk, Eli Collins, Eliza Rutherford, Erica Moreira, Kareem Ayoub, Megha Goel, Clemens Meyer, Gregory Thornton, Zhen Yang, Henryk Michalewski, Zaheer Abbas, Nathan Schucher, Ankesh Anand, Richard Ives, James Keeling, Karel Lenc, Salem Haykal, Siamak Shakeri, Pranav Shyam, Aakanksha Chowdhery, Roman Ring, Stephen Spencer, Eren Sezener, and et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

[43] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, Alhussein Fawzi, Josh Grochow, Andrea Lodi, Jean-Baptiste Mouret, Talia Ringer, and Tao Yu. Mathematical discoveries from program search with large language models. *Nature*, 625:468 – 475, 2023.

 https://doi.org/10.52202/079017-4478

[44] Giulio Rossetti, Letizia Milli, and Rémy Cazabet. Cdlib: a python library to extract, compare and evaluate communities from complex networks. *Applied Network Science*, 4, 2019.

[45] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 3125–3132. ACM, 2020.

[46] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. Little Ball of Fur: A Python Library for Graph Sampling. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*, page 3133–3140. ACM, 2020.

[47] Varuni Sarwal, Viorel Munteanu, Timur Suhodolschi, Dumitru Ciorba, Eleazar Eskin, Wei Wang, and Serghei Mangul. Biollmbench: A comprehensive benchmarking of large language models in bioinformatics. *bioRxiv*, 2023.

[48] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. Graphgpt: Graph instruction tuning for large language models, 2024.

[49] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Long Xia, Dawei Yin, and Chao Huang. Higpt: Heterogeneous graph language model, 2024.

[50] Lei Tang and Huan Liu. Graph mining applications to social network analysis. *Managing and mining graph data*, pages 487–513, 2010.

[51] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science, 2022.

[52] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M. Dai, Anja Hauth, Katie Millican, David Silver, Melvin Johnson, and ... Gemini: A family of highly capable multimodal models, 2024.

[53] Lewis Tunstall, Edward Beeching, Nathan Lambert, Nazneen Rajani, Shengyi Huang, Kashif Rasul, Alvaro Bartolome, Alexander M. Rush, and Thomas Wolf. The Alignment Handbook. URL `https://github.com/huggingface/alignment-handbook`.

[54] Fernanda B Viégas and Judith Donath. Social network visualization: Can we go beyond the graph. In *Workshop on social networks, CSCW*, volume 4, pages 6–10, 2004.

[55] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can language models solve graph problems in natural language? In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 30840–30861. Curran Associates, Inc., 2023. URL `https://proceedings.neurips.cc/paper_files/paper/2023/file/622afc4edf2824a1b6aaf5afe153fa93-Paper-Conference.pdf`.

[56] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *Annual Meeting of the Association for Computational Linguistics*, 2022.

[57] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions, 2023.

[58] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

[59] Lianghao Xia, Ben Kao, and Chao Huang. Opengraph: Towards open graph foundation models, 2024.

[60] Qianqian Xie, Weiguang Han, Zhengyu Chen, Ruoyu Xiang, Xiao Zhang, Yueru He, Mengxi Xiao, Dong Li, Yongfu Dai, Duanyu Feng, Yijing Xu, Haoqiang Kang, Ziyan Kuang, Chenhan Yuan, Kailai Yang, Zheheng Luo, Tianlin Zhang, Zhiwei Liu, Guojun Xiong, Zhiyang Deng, Yuechen Jiang, Zhiyuan Yao, Haohang Li, Yangyang Yu, Gang Hu, Jiajia Huang, Xiao-Yang Liu, Alejandro Lopez-Lira, Benyou Wang, Yanzhao Lai, Hao Wang, Min Peng, Sophia Ananiadou, and Jimin Huang. The finben: An holistic financial benchmark for large language models, 2024.

[61] Liang Xu, Anqi Li, Lei Zhu, Hang Xue, Changtai Zhu, Kangkang Zhao, Haonan He, Xuanwei Zhang, Qiyue Kang, and Zhenzhong Lan. Superclue: A comprehensive chinese large language model benchmark, 2023.

[62] Yibo Yan, Haomin Wen, Siru Zhong, Wei Chen, Haodong Chen, Qingsong Wen, Roger Zimmermann, and Yuxuan Liang. Urbanclip: Learning text-enhanced urban region profiling with contrastive language-image pretraining from the web, 2024.

[63] Mengmei Zhang, Mingwei Sun, Peng Wang, Shen Fan, Yanhu Mo, Xiaoxiao Xu, Hong Liu, Cheng Yang, and Chuan Shi. Graphtranslator: Aligning graph model to large language model for open-ended tasks. In *Proceedings of the ACM on Web Conference 2024*, WWW '24, page 1003–1014, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400701719.

[64] Zeyang Zhang, Xin Wang, Ziwei Zhang, Haoyang Li, Yijian Qin, and Wenwu Zhu. Llm4dyg: Can large language models solve spatial-temporal problems on dynamic graphs?, 2024.

[65] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 46595–46623. Curran Associates, Inc., 2023. URL `https://proceedings.neurips.cc/paper_files/paper/2023/file/91f18a1287b398d378ef22505bf41832-Paper-Datasets_and_Benchmarks.pdf`.

[66] Xingchen Zou, Yibo Yan, Xixuan Hao, Yuehong Hu, Haomin Wen, Erdong Liu, Junbo Zhang, Yong Li, Tianrui Li, Yu Zheng, and Yuxuan Liang. Deep learning for cross-domain data fusion in urban computing: Taxonomy, advances, and outlook, 2024.

 https://doi.org/10.52202/079017-4478

A Limitation and Boarder Impact

Limitation. Due to the version iterations of GPT, the results based on automated evaluation might show slight differences compared to those presented in this paper. The proposed ProGraph benchmark contains 512 hand-crafted problems, which can only reflect a core subset of graph analysis tasks. Note that our dataset includes nearly 30,000 auto-generated instances. Thus, in future work, we can expand the benchmark by manually selecting high-quality cases from the dataset. Admittedly, the benchmark still presents certain differences from real-world scenarios, *e.g.,* we did not consider a multi-turn task-solving process.

Boarder Impact. The inherent complexity of graph data in real-world scenarios demands professional coding capabilities, making it a compelling testbed for evaluating the structured data handling abilities of LLMs. We believe this work can shed light on further researches toward more powerful general intelligence with structured data understanding.

https://doi.org/10.52202/079017-4478

B Role-play Rephrasing Example

Original Question
I have a graph with an edge set [(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)], can you help me compute node connectivity for all pairs of nodes and print the node connectivity for each pair?
Role-Play Rephrasing Prompt
The above question is too abstract. Please role-play based on the following professional information, mimic their everyday language and tone, and give a background from real life (more than 4 sentences). Restate the problem, ensuring that the semantics remain unchanged. Notes: If the original question has a file, you must give the file name in your new question.
Professional Information
Ecologist: Studies the relationships between organisms and their environments, including ecosystems, habitats, and biodiversity.
Rephrased Question
Imagine we're examining a simplified model of an ecosystem where the nodes represent different species, and the edges indicate direct ecological interactions between them, such as predator-prey or symbiotic relationships. For our current study, we've mapped out a series of interactions as follows: [(1, 2), (1, 3), (2, 3), (2, 4), (3, 5), (4, 5)].
To fully understand the resilience of our ecosystem, we need to determine the species connectivity, which reflects how each pair of species is interconnected through these interactions. Can we analyze our network to reveal the minimum number of species that would need to be removed to disrupt the direct connection between any two species in this web? I'd like to have a measurable indication of node connectivity for each unique pair of species within our ecosystem based on the interactions detailed above. This will provide us with valuable insights into the robustness of their relationships and, by extension, the stability of our ecosystem.

C Document Dataset Example

Section ID triadic_census
Description
Determines the triadic census of a directed graph. The triadic census is a count of how many ...
Field List
Parameters: G : digraph; nodelist : list
Returns: census : dict
Raises: ValueError
Rubrics
Notes: This algorithm has complexity O(m) where m is the number of edges in the graph ...
References: [1]Vladimir Batagelj and Andrej Mrvar, A subquadratic triad census algorithm ...
Examples:

```
G=nx.DiGraph([(1,2),(2,3),(3,1),(3,4),(4,1),(4,2)])
...
```

D Prompt for GPT Back-Instruction

Back Instruction Prompt
Given an API doc, you need to respond like this: you should give me an example of Python code to show how to use it, and give me a question that describes the data that code uses, and the problem that code solves. Now I will give you an API document and you need to complete my request.
** A complete API documentation entry **
Generate a JSON object like this: {API: , example Python code: , question: , }

https://doi.org/10.52202/079017-4478

E ProGraph Benchmark Example

Table 6: Benchmark Example

Conponent	Example

Annotated Question	Given a graph with edge set [(1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (5, 7), (6, 7)], can you Color a graph using largest_first coloring strategy of greedy graph coloring?
	Notes: You need to print the result like this.
	`for node, color in coloring.items():`
	`  print(f"Node {node}: Color {color}")`

Rephrased Question	As a Project Coordinator, you're tasked with the organization of a network of collaborative tasks, each represented by connections between different points, or 'nodes,' in a particular sequence. This sequence is comprised of pairs indicating task interdependencies: [(1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (5, 7), (6, 7)]. The challenge lies in assigning 'colors,' or categories, to these tasks in a manner that adjacent tasks (those directly connected) are not categorized the same this ensures a clear division of responsibilities and resources.
	Your goal is to utilize the "largest_first" strategy within the greedy coloring algorithm for optimal organization. This strategic approach selects nodes based on their descending degree, that is, beginning with the node with the most connections. By categorizing in this method, you're likely to minimize the total number of categories used.
	Upon completion of this process, we need to visualize our task categorization. Could you please format the output as a sequence of print statements in Python, displaying the nodes alongside their assigned categories? The output should follow this structure for clarity:
	`for node, color in coloring.items():`
	`  print(f"Node {node}: Color {color}")`
	This simplification will effectively translate the abstract graph coloring problem into a practical scenario suitable for organizing our network of tasks.

Answer	Node 4: Color 0 Node 5: Color 1 Node 1: Color 0 Node 2: Color 1 Node 3: Color 1 Node 6: Color 0 Node 7: Color 2

Reference Code	`import networkx as nx`
	`from networkx.algorithms.coloring import greedy_color`
	`# Create a sample graph`
	`G = nx.Graph()`
	`edges = [(1, 2), (1, 3), (2, 4), (3, 4),`
	`(4, 5), (5, 6), (5, 7), (6, 7)]`
	`G.add_edges_from(edges)`
	`# Apply the largest first coloring strategy`
	`coloring = greedy_color(G, strategy="largest_first")`
	`for node, color in coloring.items():`
	`  print(f"Node {node}: Color {color}")`

Key APIs	greedy_color

F Doc+Code Dataset Example

Question
Given the karate club graph provided by NetworkX, how do we find its communities using the greedy modularity maximization method and then print out the sorted list of nodes for each community?
Answer
To solve the given question, we can leverage greedy_modularity_communities function.
It is a function that "Find communities in G using greedy modularity maximization. This function uses Clauset-Newman-Moore greedy modularity maximization [2] to find the community partition with the largest modularity. Greedy modularity maximization begins with each node in its own community and repeatedly joins the pair of communities that lead to the largest modularity until no further increase in modularity is possible (a maximum). Two keyword arguments adjust the stopping condition. cutoff is a lower limit on the number of communities so you can stop the process before reaching a maximum (used to save computation time). best_n is an upper limit on the number of communities so you can make the process continue until at most n communities remain even if the maximum modularity occurs for more. To obtain exactly n communities, set both cutoff and best_n to n. This function maximizes the generalized modularity, where resolution is the resolution parameter, often expressed as (gamma). See modularity() .".

The function accepts parameters:
"Parameters:": {
"G : NetworkX graph": "",
"weight : string or None, optional (default=None)": "The name of an edge attribute that holds the numerical value used as a weight. If None, then each edge has weight 1. The degree is the sum of the edge weights adjacent to the node.",
"resolution : float, optional (default=1)": "If resolution is less than 1, modularity favors larger communities. Greater than 1 favors smaller communities.",
"cutoff : int, optional (default=1)": "A minimum number of communities below which the merging process stops. The process stops at this number of communities even if modularity is not maximized. The goal is to let the user stop the process early. The process stops before the cutoff if it finds a maximum of modularity.",
"best_n : int or None, optional (default=None)": "A maximum number of communities above which the merging process will not stop. This forces community merging to continue after modularity starts to decrease untilbest_ncommunities remain. If None, force it to continue beyond a maximum."}

The function returns: "Returns:": {
"communities: list": "A list of frozensets of nodes, one for each community. Sorted by length with largest communities first." }
The function's path is "networkx.community.greedy_modularity_communities".

By using the info above, we can write a Python code to solve this problem.
Here is the Python code.

```
import networkx as nx
# Create a graph
G = nx.karate_club_graph()
# Find communities in the graph
communities = nx.community.greedy_modularity_communities(G,
weight='weight', resolution=1, cutoff=1, best_n=None)
# Print the sorted list of nodes in each community
for community in communities:
    print(sorted(community))
```


https://doi.org/10.52202/079017-4478

G Experimental Results of All Models

Table 7: Performance (%) of closed-source models regarding different task categories.

Model		Graph Theory		Graph Statistical Learning		Graph Embedding		Overall	
		Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy
Claude 3 Haiku	No RAG	52.9	31.6	23.4	9.7	32.6	2.9	42.2	22.4
	RAG 3	68.9	47.7	22.1	11.4	23.9	1.1	50.8	32.6
	RAG 5	63.5	44.4	29.9	16.4	15.2	2.5	49.0	32.2
	RAG 7	65.4	51.0	25.3	15.2	17.4	6.5	49.0	36.2
Claude 3 Sonnet	No RAG	57.1	33.2	15.6	4.6	10.9	0.0	40.4	21.6
	RAG 3	63.5	45.8	13.6	7.6	19.6	5.8	44.5	30.7
	RAG 5	63.5	45.5	16.2	9.7	21.7	4.7	45.9	31.1
	RAG 7	66.4	50.0	25.3	12.3	21.7	4.3	50.0	34.6
Claude 3 Opus	No RAG	69.2	47.3	31.2	15.1	47.8	14.5	55.7	34.7
	RAG 3	74.4	59.4	39.6	28.3	21.7	0.0	59.2	44.7
	RAG 5	73.4	56.4	39.6	28.8	41.3	**20.7**	60.4	44.9
	RAG 7	75.6	**59.8**	42.9	28.6	32.6	13.0	61.9	**46.2**
GPT-3.5	No RAG	64.1	35.1	24.7	8.4	15.2	1.1	47.9	24.0
	RAG 3	67.0	44.3	32.5	12.0	41.3	5.1	54.3	31.1
	RAG 5	64.4	45.2	33.1	16.2	43.5	5.4	53.1	32.9
	RAG 7	64.7	45.8	33.8	15.9	37.0	3.3	52.9	33.0
GPT-4 turbo	No RAG	72.4	42.1	39.0	14.8	41.3	12.0	59.6	31.2
	RAG 3	74.7	48.5	40.3	21.4	17.4	2.2	59.2	36.2
	RAG 5	75.6	50.0	**56.5**	**29.7**	50.0	2.9	**67.6**	39.6
	RAG 7	74.7	51.3	46.8	23.8	52.2	7.6	64.3	39.1
GPT-4o	No RAG	69.9	48.1	48.7	21.4	32.6	5.8	60.2	36.3
	RAG 3	73.7	55.5	51.3	24.7	47.8	9.8	64.7	42.1
	RAG 5	74.7	54.8	55.2	27.8	**56.5**	9.1	67.2	42.6
	RAG 7	**76.9**	57.4	48.1	24.3	54.4	4.0	66.2	42.7
Gemini 1.0 Pro	No RAG	48.7	27.7	9.1	1.7	19.6	3.3	34.2	17.7
	RAG 3	61.5	47.4	16.2	7.5	15.2	2.2	43.8	31.3
	RAG 5	62.2	44.4	15.6	6.8	13.0	0.0	43.8	29.1
	RAG 7	64.4	45.3	15.6	5.8	19.6	0.0	45.7	29.4
Gemini 1.5 Pro	No RAG	59.6	37.2	21.4	6.6	13.0	1.8	44.0	24.8
	RAG 3	70.2	52.0	24.7	12.0	30.4	13.8	52.9	36.6
	RAG 5	71.2	51.3	29.2	16.7	19.6	11.2	53.9	37.3
	RAG 7	70.5	51.9	23.4	15.3	21.7	7.3	52.0	36.9

Table 8: Performance (%) of open-source models regarding different task categories.

Model		Graph Theory		Graph Statistical Learning		Graph Embedding		Overall	
		Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy
Llama 3	No Fine-tune	36.5	17.3	12.3	3.8	15.2	0.4	27.3	11.7
	Code Only	61.2	46.7	**49.4**	**36.6**	63.0	47.5	57.8	43.8
	Code+RAG 3	51.6	30.1	47.4	30.9	63.0	44.2	51.4	31.6
	Code+RAG 5	47.8	25.2	44.8	28.9	56.5	40.2	47.7	27.6
	Code+RAG 7	47.1	25.1	46.8	30.6	60.9	36.2	48.2	27.7
	Doc+Code	69.6	53.2	42.9	23.2	**80.4**	**62.0**	62.5	44.9
Deepseek Coder	No Fine-tune	56.1	33.8	30.5	9.9	30.4	7.6	46.1	24.2
	Code Only	62.5	46.9	47.4	33.4	65.2	49.6	58.2	43.1
	Code+RAG 3	59.0	35.9	46.8	28.8	43.5	29.4	53.9	33.2
	Code+RAG 5	52.9	32.0	43.5	32.3	30.4	24.3	48.1	31.4
	Code+RAG 7	51.6	33.6	**49.4**	31.4	21.7	15.9	48.2	31.4
	Doc+Code	**71.2**	**54.1**	46.1	29.8	67.4	52.5	**63.3**	**46.6**

Table 9: Performance (%) of closed-source models regarding different answer difficulties.

Model		True/False		Drawing		Calculation		Hybrid	
		Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy
Claude 3 Haiku	No RAG	71.8	53.4	43.4	20.2	34.4	27.1	41.1	13.5
	RAG 3	71.8	52.6	52.5	30.9	53.1	46.9	41.1	22.3
	RAG 5	66.7	48.7	51.2	31.4	43.8	37.5	37.5	23.4
	RAG 7	64.1	56.4	49.4	34.9	44.6	24.1	53.1	**48.4**
Claude 3 Sonnet	No RAG	56.4	38.5	43.9	21.6	21.9	15.1	28.6	13.7
	RAG 3	**79.5**	68.4	45.7	28.2	34.4	31.3	30.4	21.4
	RAG 5	66.7	53.8	46.0	29.6	40.63	35.4	39.3	22.4
	RAG 7	71.8	51.3	52.2	33.8	41.1	23.2	43.8	43.8
Claude 3 Opus	No RAG	**79.5**	66.7	57.1	33.0	37.5	28.1	50.0	27.5
	RAG 3	**79.5**	**71.0**	61.6	44.9	43.8	40.1	48.2	27.5
	RAG 5	76.9	66.7	62.9	45.0	40.6	37.0	55.4	33.4
	RAG 7	74.4	59.0	65.2	**46.9**	50.0	48.4	48.2	29.7
GPT-3.5	No RAG	66.7	38.5	51.4	23.4	43.8	30.2	32.1	14.4
	RAG 3	74.4	56.4	55.6	29.8	43.8	35.4	51.8	19.8
	RAG 5	69.2	61.5	55.6	31.5	40.6	31.3	44.6	23.2
	RAG 7	56.4	41.0	56.1	32.5	50.0	47.9	51.8	21.9
GPT-4 turbo	No RAG	76.9	51.3	62.3	31.3	43.8	29.7	44.6	17.0
	RAG 3	76.9	55.1	60.3	33.7	62.5	52.1	48.2	31.1
	RAG 5	**79.5**	69.2	**69.4**	37.9	62.5	**56.3**	60.7	21.4
	RAG 7	71.8	59.0	66.5	38.3	**65.6**	53.1	57.1	22.4
GPT-4o	No RAG	71.8	53.9	61.3	34.7	56.3	44.3	62.5	30.7
	RAG 3	**79.5**	65.8	65.2	39.4	56.3	54.7	**69.6**	37.4
	RAG 5	69.2	56.4	68.6	40.6	59.4	54.7	**69.6**	39.9
	RAG 7	74.4	60.7	68.1	41.3	56.3	54.7	62.5	32.7
Gemini 1.0 Pro	No RAG	43.6	30.8	37.4	17.3	31.3	23.4	23.2	7.7
	RAG 3	61.5	56.4	47.8	30.0	43.8	38.5	28.6	18.8
	RAG 5	64.1	51.3	46.5	27.8	37.5	33.9	28.6	19.6
	RAG 7	66.7	53.9	48.1	28.2	40.6	34.4	34.0	17.3
Gemini 1.5 Pro	No RAG	61.5	48.7	46.8	23.8	28.1	25.0	33.9	15.2
	RAG 3	71.8	62.4	55.6	35.2	46.9	39.1	50.0	26.2
	RAG 5	76.9	57.7	57.4	36.0	46.9	40.6	39.3	30.1
	RAG 7	74.4	61.5	55.1	37.3	40.6	28.7	41.1	21.4

Table 10: Performance (%) of open-source models regarding different answer difficulties.

Model		True/False		Drawing		Calculation		Hybrid	
		Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy	Pass Rate	Accuracy
Llama 3	No Fine-tune	43.6	33.3	28.3	10.0	15.6	12.5	26.8	8.3
	Code Only	82.1	71.8	59.2	42.0	34.4	31.3	60.7	**43.6**
	Code+RAG 3	**84.6**	44.0	56.9	29.0	50.0	37.5	66.1	37.2
	Code+RAG 5	66.7	36.8	53.5	25.4	37.5	28.1	60.7	36.3
	Code+RAG 7	66.7	37.2	50.9	24.4	50.0	35.9	64.3	39.3
	Doc+Code	82.1	**73.1**	64.4	43.7	40.6	31.8	**67.9**	41.3
Deepseek Coder	No Fine-tune	66.7	41.5	47.8	22.1	**53.1**	39.4	46.4	18.2
	Code Only	71.8	61.5	60.0	41.1	50.0	**45.3**	62.5	42.1
	Code+RAG 3	71.8	48.3	57.7	32.2	**53.1**	**45.3**	44.6	22.8
	Code+RAG 5	71.8	53.9	50.7	29.3	40.6	34.4	39.3	28.6
	Code+RAG 7	74.4	54.7	50.4	28.7	37.5	34.4	48.2	31.4
	Doc+Code	79.5	68.0	**66.2**	**46.0**	37.5	34.4	66.1	42.3

https://doi.org/10.52202/079017-4478

Table 11: Performance (%) of closed-source models regarding different question types.

Model		Easy		Hard	
		Pass Rate	Accuracy	Pass Rate	Accuracy
Claude 3 Haiku	No RAG	55.7	31.1	29.3	10.2
	RAG 3	65.3	42.8	34.9	18.2
	RAG 5	62.7	42.0	33.0	18.4
	RAG 7	64.3	48.0	30.2	19.5
Claude 3 Sonnet	No RAG	56.0	30.5	21.7	9.0
	RAG 3	58.7	40.4	27.8	17.0
	RAG 5	61.0	42.1	25.9	15.4
	RAG 7	65.7	46.0	32.6	18.4
Claude 3 Opus	No RAG	68.0	44.6	41.0	20.6
	RAG 3	71.0	54.7	45.3	30.5
	RAG 5	71.7	53.7	47.6	**32.5**
	RAG 7	73.3	**55.8**	48.6	32.3
GPT-3.5	No RAG	62.3	32.2	32.6	12.4
	RAG 3	66.7	42.4	40.6	15.0
	RAG 5	64.3	44.3	40.6	16.8
	RAG 7	66.7	44.2	39.2	17.1
GPT-4 turbo	No RAG	73.3	39.9	42.0	18.8
	RAG 3	71.3	43.6	44.8	25.7
	RAG 5	**76.0**	48.8	58.5	26.6
	RAG 7	73.3	48.6	55.2	25.6
GPT-4o	No RAG	66.0	43.6	56.1	26.0
	RAG 3	70.7	52.4	59.9	27.6
	RAG 5	71.0	51.3	**64.2**	30.2
	RAG 7	72.7	54.0	59.4	26.6
Gemini 1.0 Pro	No RAG	47.3	25.1	19.8	7.2
	RAG 3	61.3	44.1	25.5	13.3
	RAG 5	59.7	41.0	25.0	12.2
	RAG 7	62.3	41.9	26.4	11.6
Gemini 1.5 Pro	No RAG	57.0	33.1	28.8	13.1
	RAG 3	67.3	45.7	39.2	23.7
	RAG 5	69.0	47.1	38.2	23.4
	RAG 7	67.0	47.5	35.9	21.9

Table 12: Performance (%) of open-source models regarding different question types.

Model		Easy		Hard	
		Pass Rate	Accuracy	Pass Rate	Accuracy
Llama 3	No Fine-tune	36.3	16.2	17.5	5.4
	Code Only	64.3	47.7	52.8	38.1
	Code+RAG 3	61.7	29.6	56.6	34.5
	Code+RAG 5	56.7	25.0	50.9	31.4
	Code+RAG 7	56.0	26.0	50.0	30.2
	Doc+Code	67.7	49.1	**60.4**	39.0
Deepseek Coder	No Fine-tune	58.3	31.8	36.8	13.6
	Code Only	64.7	46.1	54.7	38.7
	Code+RAG 3	60.0	34.7	52.8	31.1
	Code+RAG 5	55.0	31.5	43.9	31.2
	Code+RAG 7	54.7	32.9	46.2	29.2
	Doc+Code	**69.7**	**50.6**	59.4	**40.8**

H Experimental Results of Best-performing Models

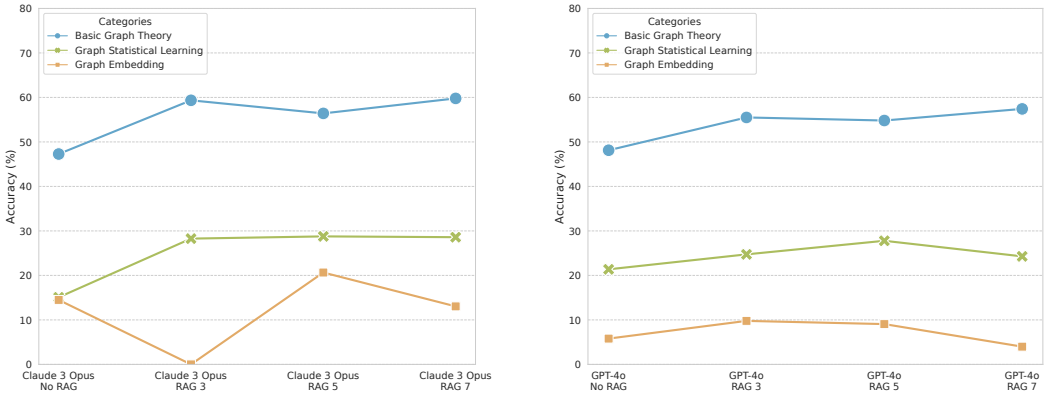


Figure 7: Accuracy of closed-source models regarding different task categories.

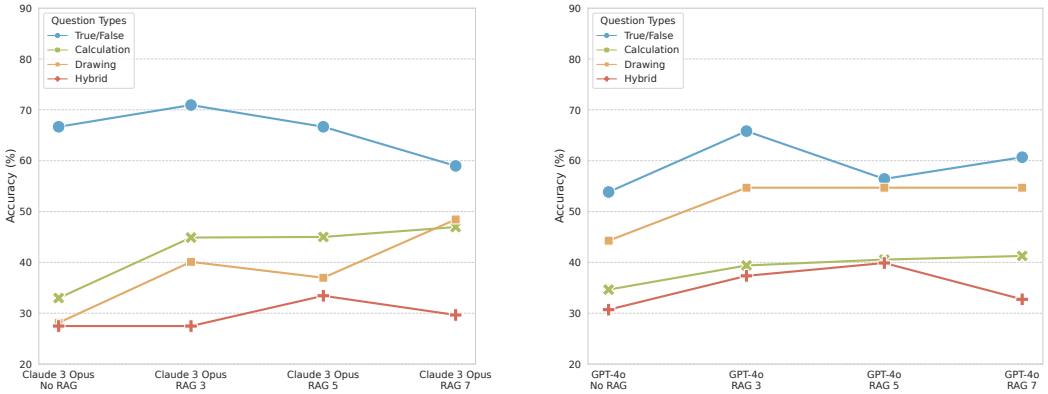


Figure 8: Accuracy of closed-source models regarding different answer difficulties.

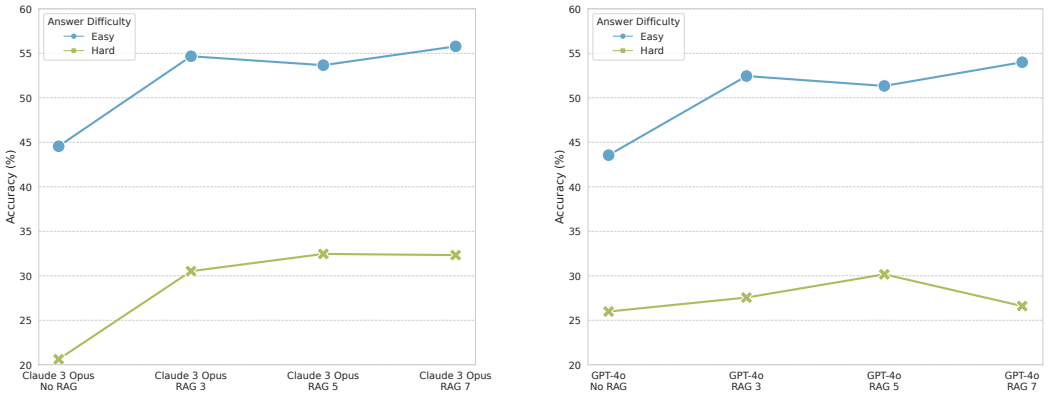


Figure 9: Accuracy of closed-source models regarding different question types.

https://doi.org/10.52202/079017-4478

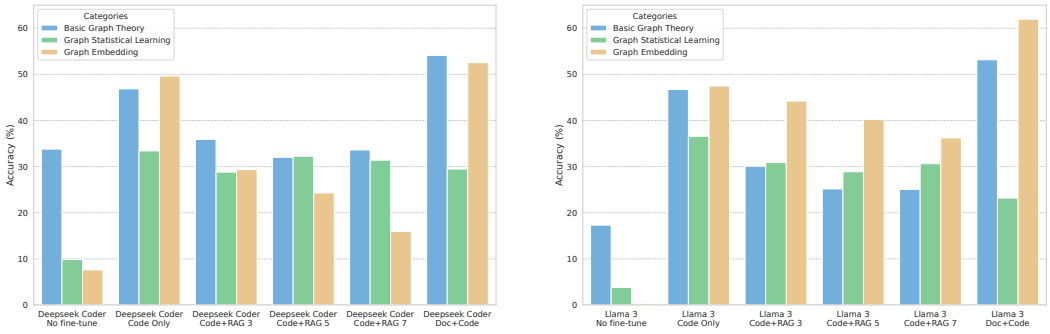


Figure 10: Accuracy of open-source models regarding different task categories.

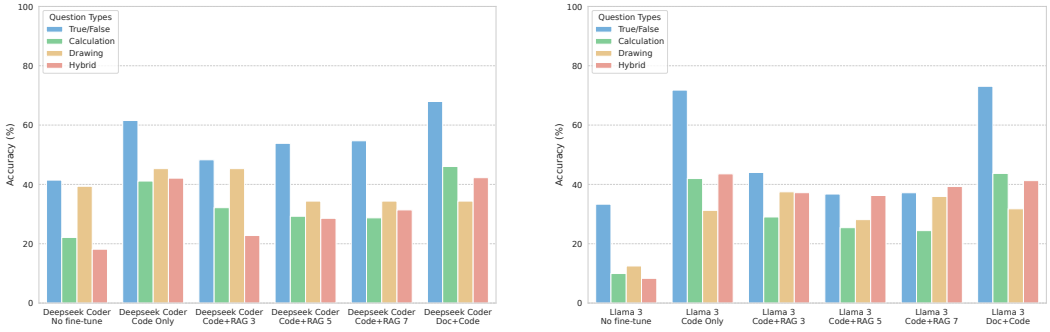


Figure 11: Accuracy of open-source models regarding different answer difficulties.

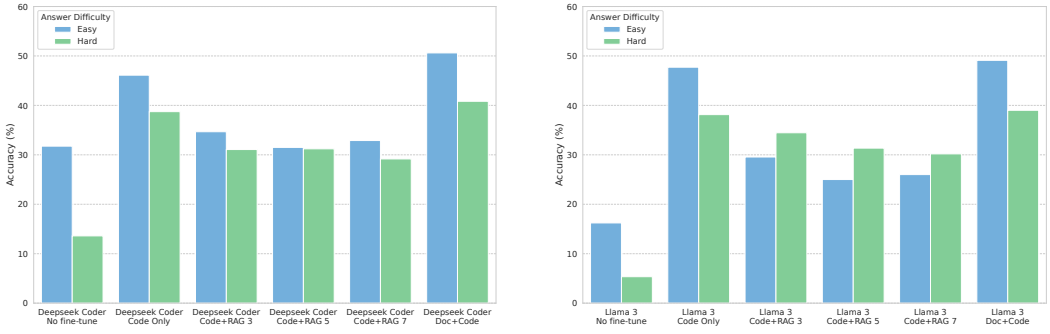


Figure 12: Accuracy of open-source models regarding different question types.

I Annotation Manual for ProGraph Benchmark

Annotation This manual provides detailed guidelines for annotating a benchmark dataset by creating and documenting tasks related to graph-related Python packages such as NetworkX and graph. Each annotator will document a function, propose a question, create a graph, and provide a reference code with execution results.

1. **Objective**
   - To create a high-quality benchmark dataset for evaluating the performance of graph-related functions in Python packages.
   - To document function usage, practical questions, graph creation, corresponding results, and key APIs.
2. **Annotation Tools**
   - **Graph Libraries:** NetworkX, igraph, CDlib, graspologic, Karate Club, Little Ball of Fur
   - **Code Editor:** Any Python IDE (e.g., PyCharm, VS Code, Jupyter Notebook)
   - **Documentation Resources:** Official documentation for the 6 Python libraries.
3. **Data Types**
   - **Function Documentation:** Description and usage of a specific function.
   - **Graph:** Manually crafted or randomly generated graphs.
   - **Code:** Python code snippets for graph creation and function execution.
   - **Results:** Output from executing the provided code.
   - **Key APIs:** APIs that are crucial and indispensable for solving a problem.
4. **Annotation Guidelines**
   - **Function Documentation:**
     - Select an API from the provided graph-related packages.
     - Document the API name, parameters, return type, and a brief description.
   - **Proposing a Question:**
     - Formulate a clear, practical question that the API can solve.
     - Ensure the question is specific and relevant to the API's capabilities.
   - **Creating a Graph:**
     - **Manual Crafting:** Draw a graph that fits the proposed question.
     - **Random Generation:** Use code to generate a random graph appropriate for the function.
     - **Graph Description:** Provide a brief description of the graph structure.
   - **Reference Code:**
     - Write Python code to create the graph.
     - Include the API call with appropriate parameters.
     - Ensure the code is well-commented and readable.
   - **Execution Result:**
     - Execute the code and record the output.
     - Provide a detailed explanation of the result.
5. **Annotation Process**
   - **Step-by-Step Procedure:**
     (a) **API Selection:** Choose an API from the provided documentation.
     (b) **Question Formulation:** Develop a practical question for the API.
     (c) **Graph Creation:** Create a graph manually or using code.
     (d) **Code Writing:** Write reference code to demonstrate the API.
     (e) **Result Recording:** Execute the code and document the output.
     (f) **Review and Submission:** Review the annotation for accuracy and clarity, then submit.
   - **Examples:**

https://doi.org/10.52202/079017-4478

  - Provide examples for each step to guide annotators.

6. **Quality Control**
   - **Review Process:**
     - Conduct peer reviews of annotations to ensure consistency and accuracy.
     - Provide feedback and request revisions if necessary.
   - **Consistency Checks:**
     - Ensure all annotations follow the same structure and guidelines.
     - Verify the correctness of code and results.
7. **Common Issues**
   - **Ambiguity in Questions:** Ensure questions are specific and clear.
   - **Errors in Code:** Double-check code for syntax and logical errors.
   - **Inconsistent Results:** Verify that results match the expected output.
8. **Annotation Tips**
   - Be precise and consistent in the document.
   - Use the provided examples as references.
   - Clarify any doubts with the project coordinator.
9. **Frequently Asked Questions (FAQs)**
   - **Q:** What if I encounter an error in the API execution?
   - **A:** Check the document and debug the code. If the error persists, seek help from the coordinator.
   - **Q:** How detailed should the graph description be?
   - **A:** Provide enough detail to understand the graph structure and its relevance to the API.