Zero-shot Graph Reasoning via Retrieval Augmented Framework with LLMs

Anonymous ACL submission

Abstract

We propose a new, training-free method, Graph Reasoning via Retrieval Augmented Framework (GRRAF), that harnesses retrieval-augmented generation (RAG) alongside the code-generation capabilities of large language models (LLMs) to address a wide range of graph reasoning tasks. In GRRAF, the target graph is stored in a graph database, and the LLM is prompted to generate executable code queries that retrieve the necessary information. This approach circumvents the limitations of existing methods that require extensive finetuning or depend on predefined algorithms, and it incorporates an error feedback loop with a time-out mechanism to ensure both correctness and efficiency. Experimental evaluations on the GraphInstruct dataset reveal that GRRAF achieves 100% accuracy on most graph reasoning tasks, including cycle detection, bipartite graph checks, shortest path computation, and maximum flow, while maintaining consistent token costs regardless of graph sizes. Imperfect but still very high performance is observed on subgraph matching. Notably, GRRAF scales effectively to large graphs with up to 10,000 nodes.

1 Introduction

Graph reasoning plays a pivotal role in modeling and understanding complex systems across numerous domains (Wu et al., 2020). Graphs naturally represent entities and their interrelations in areas such as social networks, transportation systems, biological networks, and communication infrastructures. Graph reasoning tasks like determining connectivity, detecting cycles, and finding the shortest path are not only central to theoretical computer science but also have practical implications in network optimization, anomaly detection, decision support systems, etc (Scarselli et al., 2008). However, addressing these tasks requires a deep understanding of graph topology combined with precise computational procedures, underscoring the critical challenge of developing efficient graph reasoning methods in contemporary machine learning research (Zhao et al., 2024).

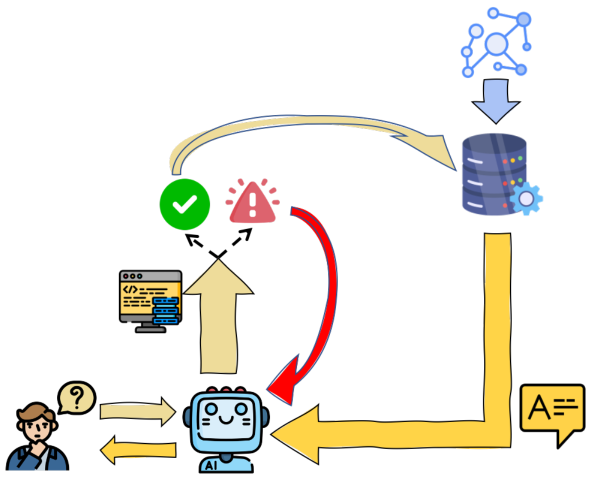


Figure 1: A schematic representation of the GRRAF concept. When a user asks a graph reasoning question, the LLM generates code to query the target graph stored in a graph database, retrieves the answer, and presents it as the response. An error feedback loop is integrated into GRRAF to prompt the LLM to refine the code whenever execution or time-out errors occur.

Large language models (LLMs) have demonstrated an impressive capacity for multi-step reasoning, which enables them to interpret complex graph-related questions expressed in natural language and generate human-readable responses (Guo et al., 2023). Several recent studies have leveraged LLMs to tackle graph reasoning problems by converting graph structures into textual representations or latent embeddings through graph neural networks (GNNs), thereby exploiting the powerful natural language reasoning capabilities of LLMs (Perozzi et al., 2024; Guo et al., 2023; Zhang, 2023; Wang et al., 2024a; Fatemi et al., 2024; Skianis et al., 2024; Lin et al., 2024). However, even when advanced prompting techniques are employed, these methods tend to perform poorly on fundamental graph reasoning tasks, such as evaluating connectivity or identifying the shortest path, with average accuracies ranging from 20% to 60%. Alternative approaches that achieve higher accuracy typically either require extensive finetuning—which results in poor performance on out-of-domain questions (Chen et al., 2024; Zhang, 2023)—or rely on predefined algorithms as input, thereby limiting their ability to address unseen tasks (Hu et al., 2024).

To address these limitations, we introduce a training-free and zero-shot method, ***the Graph Reasoning via Retrieval Augmented Framework*** (GRRAF), that leverages retrieval-augmented generation (RAG) (Lewis et al., 2020) alongside the code-writing capabilities of large language models. In GRRAF, the target graph is stored in a graph database, and the LLM is prompted to generate appropriate queries, written as code, that extract the desired answer by retrieving relevant information from the database. This strategy harnesses the LLM's robust reasoning ability and its proficiency in generating executable code, thereby achieving high accuracy on a range of graph reasoning tasks without requiring additional finetuning or predefined algorithms. In addition, we incorporate an error feedback loop combined with a time-out mechanism to ensure that the LLM produces correct queries in a time-efficient manner. Furthermore, since accurate code reliably yields the correct answer regardless of the graph's size, GRRAF can easily scale for polynomial problems to accommodate larger graphs without a drop in accuracy. In GRRAF, we use Neo4j, an interactive graph database, and NetworkX, a Python library for graphs. GRRAF accepts the target graph as either plain text or data already stored in Neo4j, specified in the prompt by the graph file name. In the former case, the prompt must specify if Neo4j or NetworkX is to be used. The LLM then must either create code to insert the graph specified in the prompt to Neo4j or to a NetworkX graph object.

GRRAF offers a fully automated, end-to-end framework for handling graph-reasoning problems written entirely in text. By leveraging the world knowledge encoded in LLMs, it generates correct code and returns accurate answers automatically for a wide range of graph-reasoning tasks expressed as natural-language questions. In addition, GRRAF establishes a foundation for future work on real-world structured relational-inference problems—ranging from knowledge-graph completion to molecular analysis—that are naturally represented as graph-structured data. An LLM user could potentially accomplish the same by directly prompting the LLM to create Python or Neo4j queries for the task on hand. Our approach offers the benefits of graph reading and loading, the execution of the code with the error-feedback loop, and the fallback approach.

Experimental results demonstrate that GRRAF achieves 100% accuracy on many graph reasoning tasks, outperforming state-of-the-art benchmarks. Moreover, GRRAF is applicable to large graphs containing up to 10,000 nodes, maintaining 100% accuracy with no increase in token cost. Although GRRAF only achieves 86.5% accuracy on subgraph matching, it still outperforms other state-of-the-art methods. Our contributions are listed below.

- **Novel Graph Reasoning Approach:** This work introduces a new method that leverages RAG to address graph reasoning tasks, such as connectivity analyses, cycle detection, and shortest path computations. It represents the first application of RAG in the domain of graph reasoning.
- **Error Feedback Loop Innovation:** The paper introduces the integration of a time out mechanism within an error feedback loop, along with the dynamic refreshing of a prompt to guide the LLM to produce more efficient code. This mechanism enhances robustness and efficiency of the generated query by preventing an infinite loop.
- **Scalable State-of-the-Art Performance:** The proposed method achieves state-of-the-art accuracy and demonstrates exceptional scalability, being the first to handle large graphs effectively without significant degradation in accuracy or substantial cost increases.

All implementations and datasets are available in `https://github.com/Anonymous-Author980/zero_shot_GRRAF/tree/main`.

2 Related Works

2.1 Graph RAG

There exist numerous prior works that employ graph data within RAG frameworks to enhance the capabilities of LLMs, a paradigm often referred

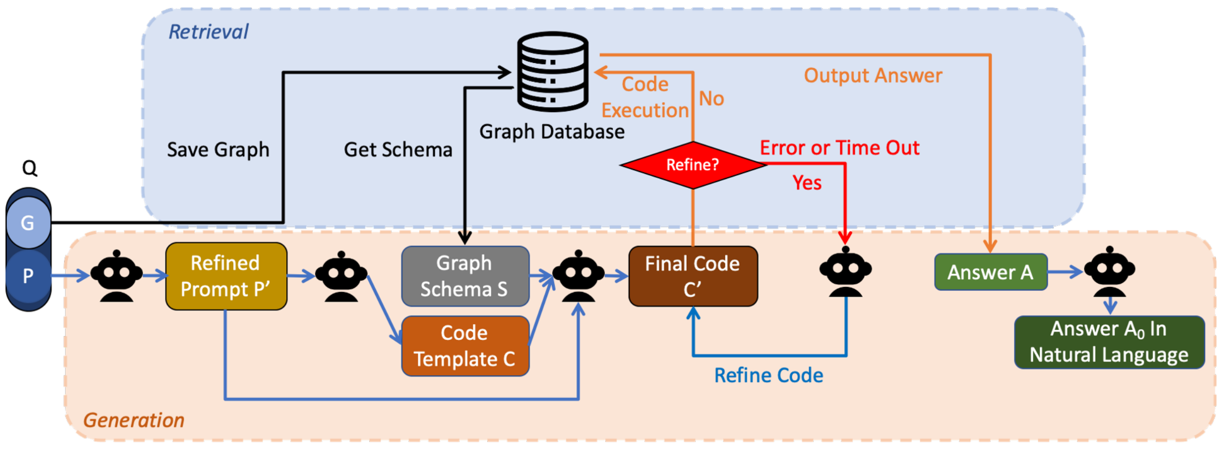
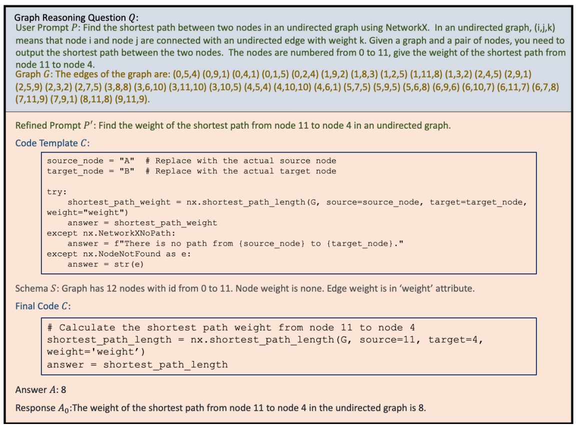


Figure 2: GRRAF workflow. The retrieval component represents the interaction with the graph database through code, while the generation component involves prompting an LLM to produce the output.

to as GraphRAG (Peng et al., 2024). These approaches retrieve graph elements containing relational knowledge relevant to a given query from a pre-constructed graph database (Edge et al., 2024). Several studies have contributed to the development of open-source knowledge graph datasets for GraphRAG (Auer et al., 2007; Suchanek et al., 2007; Vrandečić and Krötzsch, 2014; Sap et al., 2019; Liu and Singh, 2004; Bollacker et al., 2008). Building on these datasets, many methods opt to convert graphs to other easily retrievable forms, such as text (Li et al., 2023; Huang et al., 2023; Yu et al., 2023; Edge et al., 2024; Dehghan et al., 2024) or vectors (He et al., 2024; Sarmah et al., 2024), to improve the efficiency of query operations on graph databases. To further enhance the quality of retrieved data, several approaches optimize the retrieval process within GraphRAG by refining the retriever component (Delile et al., 2024; Zhang et al., 2022a; Kim et al., 2023; Wold et al., 2023; Jiang et al., 2023; Mavromatis and Karypis, 2024), optimizing the retrieval paradigm (Wang et al., 2024b; Sun et al., 2024c; Lin et al., 2019), and editing a user query or the retrieved information (Jin et al., 2024; LUO et al., 2024; Ma et al., 2025; Sun et al., 2024a; Taunk et al., 2023; Yasunaga et al., 2021). Furthermore, many methods enhance the answer generation process of GraphRAG to ensure that the LLM fully utilizes the retrieved graph data to generate the correct answer (Dong et al., 2023; Mavromatis and Karypis, 2022; Jiang et al., 2024; Sun et al., 2024b; Zhang et al., 2022b; Zhu et al., 2024; Wen et al., 2024; Shu et al., 2022; Baek et al., 2023). However, these methods focus exclusively on knowledge graphs and cannot be directly applied to solve graph reasoning questions. In contrast, GRRAF is the first method to employ RAG for addressing graph reasoning questions on pure graphs.

2.2 Graph Reasoning

Recent work has explored the use of LLMs to address graph reasoning problems. Several methods rely solely on prompt engineering techniques to enhance LLM reasoning capabilities on graphs (Liu and Wu, 2023; Guo et al., 2023; Wang et al., 2024a; Zhang et al., 2024; Fatemi et al., 2024; Wu et al., 2024; Tang et al., 2025; Skianis et al., 2024; Lin et al., 2024). Building on them, Perozzi et al. (2024) integrate a trained graph neural network (Scarselli et al., 2008) with an LLM to improve its performance on graph reasoning tasks by encoding each graph into a token provided as input to the LLM. Meanwhile, Zhang (2023) and Chen et al. (2024) finetune an LLM with instructions tailored to graph reasoning tasks to boost performance. In another approach, Hu et al. (2024) propose a multi-agent solution for graph reasoning problems by assigning an LLM agent to each node and enabling communication among agents based on a predefined algorithm. In contrast, GRRAF employs RAG to address graph reasoning problems without extensive prompt engineering. This approach is training-free and thus unsupervised and does not depend on any predefined algorithm. Furthermore, unlike previous methods, the LLM in GRRAF does not receive the entire graph as input; consequently, the token usage remains independent of graph size, thereby enabling efficient scalability to very large graphs.



Graph Reasoning Question Q:
User Prompt P: Find the shortest path between two nodes in an undirected graph using NetworkX. In an undirected graph, (i,j,k) means that node i and node j are connected with an undirected edge with weight k. Given a graph and a pair of nodes, you need to output the shortest path between the two nodes. The nodes are numbered from 0 to 11, give the weight of the shortest path from node 11 to node 4.
Graph G: The edges of the graph are: (0,5,4) (0,9,1) (0,4,1) (0,1,5) (0,2,4) (1,9,2) (1,8,3) (1,2,5) (1,11,8) (1,3,2) (2,4,5) (2,9,1) (2,5,9) (2,3,2) (2,7,5) (3,8,8) (3,6,10) (3,11,10) (3,10,5) (4,5,4) (4,10,10) (4,6,1) (5,7,5) (5,9,5) (5,6,8) (6,9,6) (6,10,7) (6,11,7) (6,7,8) (7,11,9) (7,9,1) (8,11,8) (9,11,9).

Refined Prompt P': Find the weight of the shortest path from node 11 to node 4 in an undirected graph.

Code Template C:

```
source_node = "A"  # Replace with the actual source node
target_node = "B"  # Replace with the actual target node

try:
    shortest_path_weight = nx.shortest_path_length(G, source=source_node, target=target_node,
weight="weight")
    answer = shortest_path_weight
except nx.NetworkXNoPath:
    answer = f"There is no path from {source_node} to {target_node}."
except nx.NodeNotFound as e:
    answer = str(e)
```

Schema S: Graph has 12 nodes with id from 0 to 11. Node weight is none. Edge weight is in 'weight' attribute.

Final Code C:

```
# Calculate the shortest path weight from node 11 to node 4
shortest_path_length = nx.shortest_path_length(G, source=11, target=4,
weight='weight')
answer = shortest_path_length
```

Answer A: 8

Response A_0:The weight of the shortest path from node 11 to node 4 in the undirected graph is 8.

Figure 3: An illustrative example demonstrating the application of GRRAF to solve a shortest path question by using NetworkX. Graph G in text is stored as an NetworkX object by code.

Task	Node Size	# of Test Graphs
Cycle Detection	[2, 100]	400
Connectivity	[2, 100]	400
Bipartite Graph Check	[2, 100]	400
Topological Sort	[2, 50]	400
Shortest Path	[2, 100]	400
Maximum Triangle Sum	[2, 25]	400
Maximum Flow	[2, 50]	400
Subgraph Matching	[2, 30]	400
Indegree Calculation	[2, 50]	400
Outdegree Calculation	[2, 50]	400

Table 1: The detailed information of GraphInstruct dataset and two additional tasks (indegree calculation and outdegree calculation). The subgraph matching task is to verify if there exists a subgraph in G that is isomorphic to a given graph G'.

3 Method

In this section, we explain how GRRAF integrates RAG to address graph reasoning questions and retrieve accurate answers. The entire workflow of GRRAF is demonstrated in Figure 2. A graph reasoning question, denoted as Q, consists of two components: a graph G and a user prompt P. The graph G represents the target graph associated with Q and is stored either in Neo4j or as a NetworkX graph object (code written by an LLM and executed by an agent). The prompt P contains a graph-specific question regarding G (e.g., "Does node 2 connect to node 5?" or "What is the shortest path from node 5 to node 8?"). To enhance code generation by the language model, we initially input P into the LLM, requesting it to refine the prompt, clarify the format, and eliminate redundant information. The resulting refined prompt is denoted as P'. Then, the LLM is prompted to generate a generic code template C that addresses P' without incorporating graph-specific details. For example, if P' states "Find the shortest path from node 3 to node 5," the template C encapsulates a generic shortest path algorithm that does not include the specific node identifiers. Additionally, we extract the schema S (comprising of node properties and edge properties) from the graph database using a hard-coded procedure. This schema ensures that the LLM-generated code utilizes correct variable names.

Subsequently, we provide P', C, and S to the LLM and instruct it to generate the final code C' that produces an answer A corresponding to P'. An error feedback loop is incorporated into this process. If an error arises during the execution of C', the error message, along with C', is supplied back to the LLM, prompting it to produce a revised version of the code. To promote the generation of time-efficient code, given that multiple algorithms with varying time complexities may be applicable, we integrate a time-out mechanism within the error feedback loop. Specifically, a time limit t is imposed on the execution of C'. If the execution time exceeds t, the process is halted, and the LLM is asked to modify C' so that it runs faster. If the feedback loop iterates more than n times, the system reverts to using the original question Q as a prompt to directly obtain the answer A from the LLM. This forced exit is designed to prevent perpetual iterations when addressing computationally intractable NP-hard problems (e.g., substructure matching on large graphs), where no modification of C' can reduce the execution time below the threshold t.

In the final step, the answer A is provided to the LLM to generate a reader-friendly natural language response A_0 that addresses the graph reasoning question Q. An example of solving a graph reasoning question with GRRAF is demonstrated in Figure 3.

4 Computational Assessment

4.1 Dataset and Benchmark

We conduct experiments on GraphInstruct (Chen et al., 2024), a dataset that comprises of nine graph reasoning tasks with varying complexities. Due to its diversity in graph reasoning tasks and its prior

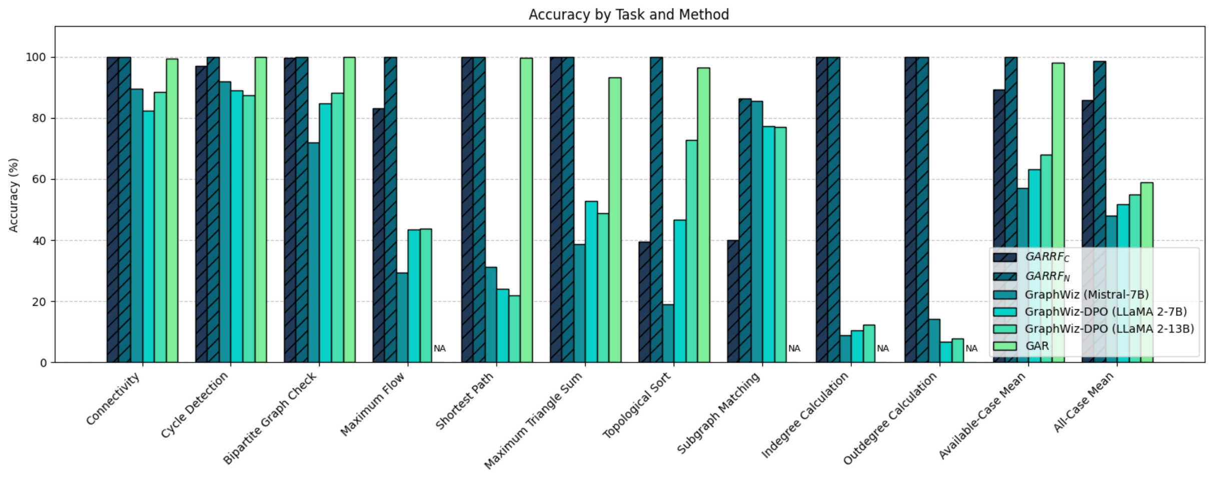


Figure 4: Performance of GRRAF and benchmark models across all ten graph reasoning tasks. Missing data are indicated as "NA" in the plot. The available-case mean refers to the average accuracy of each method calculated using only the tasks where complete data is available (excluding maximum flow, subgraph matching, indegree calculation, and outdegree calculation). The all-case mean refers to the average accuracy across all tasks, treating 'NA' as 0.

use in evaluating state-of-the-art methods (Chen et al., 2024; Hu et al., 2024), we select this dataset for our evaluation. However, the task of finding a Hamilton path lacks publicly available ground truth labels and generating such labels through code is infeasible due to the NP-hard nature of the problem; consequently, we exclude this task from our experiments. Accordingly, we assess the performance of GRRAF on the following eight tasks: cycle detection, connectivity, bipartite graph check, topological sort, shortest path, maximum triangle sum, maximum flow, and subgraph matching. Details of these tasks are provided in Table 1. Moreover, to achieve a more robust performance evaluation, we augment the test dataset with two additional simple tasks—indegree calculation and outdegree calculation (as shown in Table 1)—to facilitate a comprehensive evaluation of GRRAF and the state-of-the-art benchmarks. Each task has 400 question–graph pairs, each with a single correct answer. We measure a method's performance on one task by its accuracy—that is, the proportion of questions answered correctly out of the total (400).

GRRAF, i.e., its LLM, generates code which is either correct or not. This is the reason why most accuracies are going to be 100%. For tasks with less than 100% accuracy, GRRAF yields correct code but the underlying problems are NP-hard and for some test graphs the execution times out. One can argue that the output code is correct and thus appropriate credit should be given, but on the other hand, a more efficient algorithm and code can be potentially produced. Sometimes the generated code does not handle edge cases correctly, yet other times the code or algorithms are incorrect (they solve only some test graphs by coincidence).

We compare the performance of GRRAF against two state-of-the-art benchmarks: GraphWiz (Chen et al., 2024) and GAR (Hu et al., 2024). GraphWiz is trained on 17,158 questions and 72,785 answers, complete with reasoning paths, from the training set of GraphInstruct. Since no single version of GraphWiz consistently outperforms the others across all tasks, we include three versions in our comparisons: GraphWiz (Mistral-7B), GraphWiz-DPO (LLaMA 2-7B), and GraphWiz-DPO (LLaMA 2-13B). GAR is a training-free multi-agent framework that relies on a predefined library of distributed algorithms created by humans. As a result, it is incapable of solving unseen graph reasoning tasks that require algorithms not present in its library. Therefore, some results from GAR are missing in the subsequent comparisons because of its limitation.

4.2 Experiments

We conduct experiments using GRRAF with a time limit of $t = 5$ minutes and a maximum error feedback loop iteration of $n = 3$. The backbone LLM is GPT-4o. These parameter choices are justified by the sensitivity analysis in Appendix A. For the graph querying code, we evaluate two approaches: Cypher, a query language for Neo4j, and NetworkX, a Python library for graphs, which we denote as GRRAF$_C$ and GRRAF$_N$, respectively. We deal with graph plain text, and thus can be converted into either Neo4j data or NetworkX objects.

Figure 4 demonstrates that GARRF$_N$ outperforms all benchmark methods, achieving 100% accuracy on most graph reasoning tasks. GARRF$_C$ exhibits comparable or superior performance relative to other benchmarks on the majority of tasks, except for topological sort and subgraph matching. Although GraphWiz outperforms GARRF$_C$ in topological sort and subgraph matching, its inadequate performance on indegree calculation and outdegree calculation suggests that it struggles with even simple out-of-domain graph reasoning tasks. Furthermore, due to its inherent limitations, GAR is inapplicable to out-of-domain tasks such as maximum flow, subgraph matching, indegree calculation, and outdegree calculation. Consequently, considering both performance and generalization ability, GARRF$_C$ and GARRF$_N$ are better for addressing graph reasoning tasks than the other benchmark models. The example code generated by GARRF$_N$ for each graph reasoning task is presented in Appendix B.

Subgraph matching is NP-complete, and the code produced by GARRF$_N$ has exponential time complexity. For graphs of 20 nodes, executing that code can take over a day—exceeding the time limit t. Based on Section 3, in such cases GARRD$_N$ falls back to using the original question Q as a prompt to obtain the answer A directly from the LLM, which may yield incorrect results. GRRAF$_C$ likewise falls short of 100% accuracy on cycle detection and bipartite-graph checking, since Cypher queries execute more slowly than NetworkX. For the maximum-flow task, GRRAF$_C$ produces code that overlooks certain edge cases. And for topological sorting and subgraph matching, it generates code that only succeeds on some graphs by chance.

Across the ten tasks, solving a single graph reasoning question requires GRRAF$_N$ to use an average of 767 input tokens and 124 output tokens, while GRRAF$_C$ uses 796 input tokens and 201 output tokens. In comparison, GraphWiz (Mistral-7B) consumes an average of 1,046 input tokens and 126 output tokens per question, whereas GraphWiz-DPO (LLaMA 2-7B) requires 1,046 input tokens and 290 output tokens on average, and GraphWiz-DPO (LLaMA 2-13B) uses 1,046 input tokens and 301 output tokens per question. Notably, GAR demands more resources, averaging 8,095 input tokens and 5,987 output tokens for each graph reasoning question. Thus, comparing to other benchmark methods, GRRAF$_N$ and GRRAF$_C$ achieve high accuracy in graph reasoning tasks while utilizing fewer token resources.

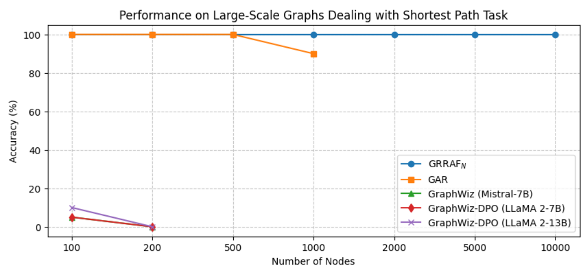


Figure 5: Accuracy of each method on the shortest path task across graphs of differenct sizes (number of nodes).

Since the largest graph in GraphInstruct (Chen et al., 2024) comprises of only 100 nodes, which remains insufficient for real-world graph reasoning scenarios (Hu et al., 2024), we further evaluate the best-performing method, GRRAF$_N$, on large-scale graphs. Following the approach of Hu et al. (2024), we assess GRRAF$_N$ on the shortest path task using larger graphs with 20 test samples for each graph size. Whereas their work scales graphs to 1,000 nodes, we extend this evaluation by scaling graphs to 10,000 nodes to thoroughly assess the performance of GRRAF$_N$. According to Figure 5, GRRAF$_N$ achieves 100% accuracy across all graph sizes, demonstrating its exceptional scalability. GAR attains 100% accuracy on graphs with 100, 200, and 500 nodes, but its accuracy decreases to 90% on graphs with 1,000 nodes. Due to token limitations, GAR is unable to address questions on graphs with 2,000 nodes or more. In contrast, all three versions of GraphWiz perform poorly on large graphs, achieving only 5-10% accuracy on graphs with 100 nodes and failing entirely on graphs with 200 nodes. The token limits of their base model prevent them from processing graphs larger than 200 nodes.

We also record the variation in token cost required to solve a single graph reasoning question as the graph size increases on the shortest path task. As illustrated in Figure 6, the number of tokens used by GRRAF$_N$ remains constant regardless of the graph size. As detailed in Section 3, GRRAF interacts with the graph database solely via the graph database through code execution; thus, the graph description (nodes, edges, weights) is not directly input to the LLM, and the token cost remains unaffected by increases in graph size. In contrast, the token cost

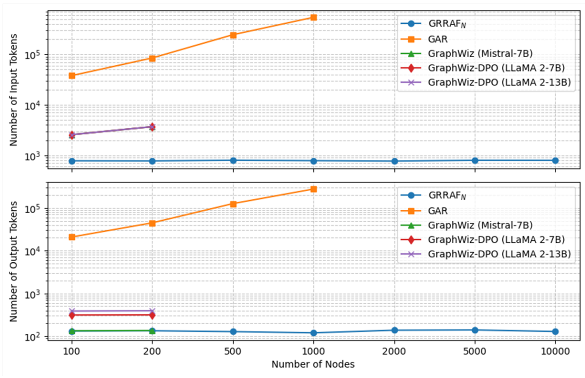


Figure 6: Average token cost for solving a graph reasoning problem across graphs of varying sizes on the shortest path task.

Method	Execution Error	Time-out
GRRAF$_N$	2.2%	5.4%
GRRAF$_C$	4.9%	9.1%

Table 2: Percentage of graph reasoning questions over 10 tasks triggering error feedback loop due to execution errors or time-outs for each method.

for GraphWiz increases linearly with graph size because it must pass the information of each node and edge to the LLM. The token cost for GAR is considerably higher than that for GRRAF$_N$ and grows nearly exponentially with graph size. This is due to GAR's design, where each node is assigned an LLM agent, and each agent communicates with every adjacent agent in each iteration (Hu et al., 2024). As the number of nodes increases, so do the number of agents, the number of adjacent agents per node (i.e., edges), and the number of iterations required to obtain an answer, all of which contribute to a significant rise in token cost. Therefore, compared to other benchmarks, GRRAF$_N$ can readily scale to very large graphs (up to 10,000 nodes) without compromising performance and increasing token cost.

To evaluate the effectiveness of the error feedback loop, we quantify the total percentage of questions that activate this loop, as reported in Table 2. In general, GRRAF$_C$ triggers the error feedback loop more frequently than GRRAF$_N$. For both variants, the loop is activated due to time-outs more often than due to execution errors, underscoring the importance of time efficiency in graph reasoning tasks. Overall, the backbone LLM generates correct code queries in most instances, and the integration of an error feedback loop with a time-out mechanism further enhances code accuracy and efficiency.

5 Conclusion

In this work, we introduced GRRAF, a novel framework that integrates RAG with the code-writing prowess of LLMs to address graph reasoning questions. Our approach, which operates without additional training or reliance on predefined algorithms, leverages a graph database to store target graphs and employs an error feedback loop with a time-out mechanism to ensure the generation of correct and efficient code queries. Comprehensive experiments on the GraphInstruct dataset and two extra tasks (indegree and outdegree) demonstrate that GRRAF outperforms existing state-of-the-art benchmarks, achieving 100% accuracy on a majority of graph reasoning tasks while effectively scaling to graphs containing up to 10,000 nodes without incurring extra token costs. These findings underscore the potential of combining retrieval-based techniques with LLM-driven code generation for solving complex graph reasoning problems. Future work could explore extending this framework to dynamic graph scenarios and additional reasoning tasks, further enhancing its applicability and robustness.

6 Limitations

Although GRRAF$_N$ attains 100% accuracy on all polynomial-time graph reasoning tasks, it nevertheless struggles to solve NP-complete problems—such as subgraph matching—both accurately and efficiently. Moreover, the inferior performance of GRRAF$_C$ relative to GRRAF$_N$ indicates that our framework currently generates lower-quality Cypher queries than the equivalent Python code. These two issues constitute the primary limitations of our method.

References

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *International Semantic Web Conference*, pages 722–735.

Jinheon Baek, Soyeong Jeong, Minki Kang, Jong Park, and Sung Hwang. 2023. Knowledge-augmented language model verification. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1720–1736.

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1247–1250.

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*, pages 353–364.

Mohammad Dehghan, Mohammad Alomrani, Sunyam Bagga, David Alfonso-Hermelo, Khalil Bibi, Abbas Ghaddar, Yingxue Zhang, Xiaoguang Li, Jianye Hao, Qun Liu, Jimmy Lin, Boxing Chen, Prasanna Parthasarathi, Mahdi Biparva, and Mehdi Rezagholizadeh. 2024. EWEK-QA : Enhanced web and efficient knowledge graph retrieval for citation-based question answering systems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14169–14187.

Julien Delile, Srayanta Mukherjee, Anton Van Pamel, and Leonid Zhukov. 2024. Graph-based retriever captures the long tail of biomedical knowledge. In *ICML'24 Workshop ML for Life and Material Science: From Theory to Industry Applications*.

Junnan Dong, Qinggang Zhang, Xiao Huang, Keyu Duan, Qiaoyu Tan, and Zhimeng Jiang. 2023. Hierarchy-aware multi-hop question answering over knowledge graphs. In *Proceedings of the ACM web conference 2023*, pages 2519–2527.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From local to global: A graph RAG approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*.

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*.

Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. 2023. GPT4Graph: Can large language models understand graph structured data? an empirical evaluation and benchmarking. In *arXiv preprint arXiv:2305.15066*.

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 *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

Yuwei Hu, Runlin Lei, Xinyi Huang, Zhewei Wei, and Yongchao Liu. 2024. Scalable and accurate graph reasoning with LLM-based multi-agents. In *arXiv preprint arXiv:2410.05130*.

Yongfeng Huang, Yanyang Li, Yichong Xu, Lin Zhang, Ruyi Gan, Jiaxing Zhang, and Liwei Wang. 2023. MVP-Tuning: Multi-view knowledge retrieval with prompt tuning for commonsense reasoning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13417–13432.

Boran Jiang, Yuqi Wang, Yi Luo, Dawei He, Peng Cheng, and Liangcai Gao. 2024. Reasoning on efficient knowledge paths: knowledge graph guides large language model for domain question answering. In *2024 IEEE International Conference on Knowledge Graph*, pages 142–149.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251.

Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Zheng Li, Ruirui Li, Xianfeng Tang, Suhang Wang, Yu Meng, and Jiawei Han. 2024. Graph chain-of-thought: Augmenting large language models by reasoning on graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 163–184.

Jiho Kim, Yeonsu Kwon, Yohan Jo, and Edward Choi. 2023. KG-GPT: A general framework for reasoning on knowledge graphs using large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*.

Patrick 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. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Shiyang Li, Yifan Gao, Haoming Jiang, Qingyu Yin, Zheng Li, Xifeng Yan, Chao Zhang, and Bing Yin. 2023. Graph reasoning for question answering with triplet retrieval. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 3366–3375.

Bill Yuchen Lin, Xinyue Chen, Jamin Chen, and Xiang Ren. 2019. KagNet: Knowledge-aware graph networks for commonsense reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 2829–2839.

Tianqianjin Lin, Pengwei Yan, Kaisong Song, Zhuoren Jiang, Yangyang Kang, Jun Lin, Weikang Yuan, Junjie Cao, Changlong Sun, and Xiaozhong Liu. 2024. LangGFM: A large language model alone can be a powerful graph foundation model. In *arXiv preprint arXiv:2410.14961*.

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

Hugo Liu and Push Singh. 2004. ConceptNet—a practical commonsense reasoning tool-kit. In *BT Technology Journal*, volume 22, pages 211–226.

LINHAO LUO, Yuan-Fang Li, Reza Haf, and Shirui Pan. 2024. Reasoning on graphs: Faithful and interpretable large language model reasoning. In *The Twelfth International Conference on Learning Representations*.

Shengjie Ma, Chengjin Xu, Xuhui Jiang, Muzhi Li, Huaren Qu, Cehao Yang, Jiaxin Mao, and Jian Guo. 2025. Think-on-graph 2.0: Deep and faithful large language model reasoning with knowledge-guided retrieval augmented generation. In *The Thirteenth International Conference on Learning Representations*.

Costas Mavromatis and George Karypis. 2022. ReaRev: Adaptive reasoning for question answering over knowledge graphs. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2447–2458.

Costas Mavromatis and George Karypis. 2024. GNN-RAG: Graph neural retrieval for large language model reasoning. In *arXiv preprint arXiv:2405.20139*.

Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. 2024. Graph retrieval-augmented generation: A survey. In *arXiv preprint arXiv:2408.08921*.

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

Maarten Sap, Ronan Le Bras, Emily Allaway, Chandra Bhagavatula, Nicholas Lourie, Hannah Rashkin, Brendan Roof, Noah A Smith, and Yejin Choi. 2019. Atomic: An atlas of machine commonsense for if-then reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3027–3035.

Bhaskarjit Sarmah, Dhagash Mehta, Benika Hall, Rohan Rao, Sunil Patel, and Stefano Pasquali. 2024. HybridRAG: Integrating knowledge graphs and vector retrieval augmented generation for efficient information extraction. In *Proceedings of the 5th ACM International Conference on AI in Finance*, pages 608–616.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. In *IEEE Transactions on Neural Networks*, volume 20, pages 61–80.

Yiheng Shu, Zhiwei Yu, Yuhan Li, Börje Karlsson, Tingting Ma, Yuzhong Qu, and Chin-Yew Lin. 2022. TIARA: Multi-grained retrieval for robust question answering over large knowledge base. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 8108–8121.

Konstantinos Skianis, Giannis Nikolentzos, and Michalis Vazirgiannis. 2024. Graph reasoning with large language models via pseudo-code prompting. In *arXiv preprint arXiv:2409.17906*.

Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, pages 697–706.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. 2024a. Think-on-Graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. 2024b. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*.

Lei Sun, Zhengwei Tao, Youdi Li, and Hiroshi Arakawa. 2024c. ODA: Observation-driven agent for integrating LLMs and knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 7417–7431.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. 2025. GraphArena: Evaluating and exploring large language models on graph computation. In *The Thirteenth International Conference on Learning Representations*.

Dhaval Taunk, Lakshya Khanna, Siri Venkata Pavan Kumar Kandru, Vasudeva Varma, Charu Sharma, and Makarand Tapaswi. 2023. GrapeQA: Graph augmentation and pruning to enhance question-answering. In *Companion Proceedings of the ACM Web Conference 2023*, pages 1138–1144.

Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. In *Communications of the ACM*, volume 57, pages 78–85.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2024a. Can language models solve graph problems in natural language? In *Advances in Neural Information Processing Systems*, volume 36.

Yu Wang, Nedim Lipka, Ryan A Rossi, Alexa Siu, Ruiyi Zhang, and Tyler Derr. 2024b. Knowledge graph prompting for multi-document question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 19206–19214.

Yilin Wen, Zifeng Wang, and Jimeng Sun. 2024. MindMap: Knowledge graph prompting sparks graph of thoughts in large language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10370–10388.

Sondre Wold, Lilja Øvrelid, and Erik Velldal. 2023. Text-to-KG alignment: Comparing current methods on classification tasks. In *Proceedings of the First Workshop on Matching From Unstructured and Structured Data*, pages 1–13.

Qiming Wu, Zichen Chen, Will Corcoran, Misha Sra, and Ambuj K Singh. 2024. Grapheval2000: Benchmarking and improving large language models on graph datasets. In *arXiv preprint arXiv:2406.16176*.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. In *IEEE Transactions on Neural Networks and Learning Systems*, volume 32, pages 4–24.

Michihiro Yasunaga, Hongyu Ren, Antoine Bosselut, Percy Liang, and Jure Leskovec. 2021. QA-GNN: Reasoning with language models and knowledge graphs for question answering. In *North American Chapter of the Association for Computational Linguistics*.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Yang Wang, Zhiguo Wang, and Bing Xiang. 2023. DecAF: Joint decoding of answers and logical forms for question answering over knowledge bases. In *The Eleventh International Conference on Learning Representations*.

Jiawei Zhang. 2023. Graph-toolformer: To empower LLMs with graph reasoning ability via prompt augmented by chatgpt. In *arXiv preprint arXiv:2304.11116*.

Jing Zhang, Xiaokang Zhang, Jifan Yu, Jian Tang, Jie Tang, Cuiping Li, and Hong Chen. 2022a. Subgraph retrieval enhanced model for multi-hop knowledge base question answering. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5773–5784.

Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. 2022b. GreaseLM: Graph REASoning enhanced language models. In *International Conference on Learning Representations*.

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*, page 4350–4361.

Jianan Zhao, Le Zhuo, Yikang Shen, Meng Qu, Kai Liu, Michael M. Bronstein, Zhaocheng Zhu, and Jian Tang. 2024. Graphtext: Graph reasoning in text space. In *Adaptive Foundation Models: Evolving AI for Personalized and Efficient Learning*.

Yun Zhu, Yaoke Wang, Haizhou Shi, and Siliang Tang. 2024. Efficient tuning and inference for large language models on textual graphs. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, pages 5734–5742.

A Sensitivity Analysis

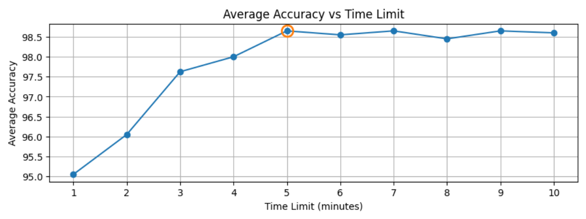


Figure 7: Average accuracy of GRRAF$_N$ with different time limit t.

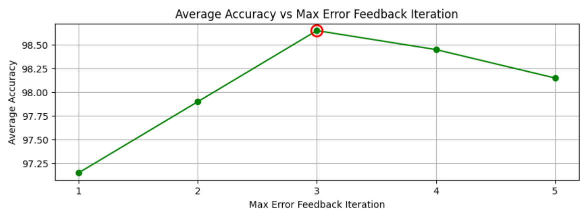


Figure 8: Average accuracy of GRRAF$_N$ with different maximum error feedback loop iteration n.

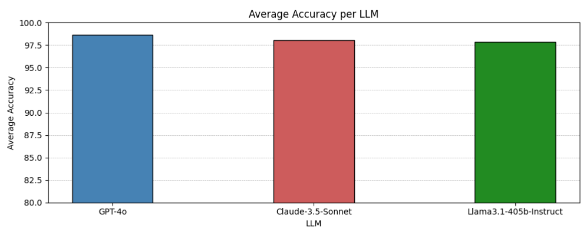


Figure 9: Average accuracy of GRRAF$_N$ with different backbone LLM.

We perform sensitivity analyses on GRRAF$_N$ to assess the impact of the time limit t, the maximum number of error-feedback loop iterations n, and the choice of backbone LLM. We report the average accuracy across all ten graph reasoning tasks. As shown in Figure 7, the accuracy increases with t up to five minutes, after which no further gains are observed. Figure 8 indicates that accuracy peaks at $n = 3$ and declines slightly for $n > 3$. Finally, we evaluated GRRAF$_N$ using three backbone LLMs—GPT-4o, Claude-3.5-Sonnet, and Llama3.1-405b-Instruct—and found that all three yield comparable results, with GPT-4o achieving a slightly higher average accuracy than the others (Figure 9).

B Example Code

This section presents example code generated by GRRAF$_N$ for each graph reasoning task in our experiments: cycle detection (Figure 10), connectivity (Figure 11), bipartite graph check (Figure 12), topological sort (Figure 13), shortest path (Figure 14), maximum triangle sum (Figure 15), maximum flow (Figure 16), subgraph matching (Figure 17), indegree calculation (Figure 18), and outdegree calculation (Figure 19). All these examples produce correct answers.

We also include in Figure 20 an example Cypher query generated by GRRAF$_C$ for the maximum-flow task. Although this query attempts to implement the Ford–Fulkerson algorithm, it omits the backward residual edges, preventing any rerouting of earlier flows. Consequently, on certain edge cases (e.g., the graph in Figure 21), it produces incorrect results. Similarly, Figure 22 shows an instance where GRRAF$_C$ generates an incorrect Cypher query for topological sorting. That query builds a spanning tree rooted at a node of zero indegree to derive the ordering—a method that is unsound and succeeds only by chance on some graphs.

Cycle Detection

```
try: # Attempt to find a cycle in the graph G
     nx.find_cycle(G, orientation='ignore’)
     answer = "Yes."
except nx.exception.NetworkXNoCycle:
     answer = "No."
```

Figure 10: An example of the final code C' generated for the cycle detection task.

Connectivity

```
# Check if there is a path between node 8 and node 10 in the graph G
if nx.has_path(G, 8, 10):
    answer = "Yes."
else:
    answer = "No."
```

Figure 11: An example of the final code C' generated for the connectivity task.

Bipartite Graph Check

```
# Check if the directed graph G is bipartite
is_bipartite = nx.is_bipartite(G)
# Generate the answer string
answer = "Yes" if is_bipartite else "No"
```

Figure 12: An example of the final code C' generated for the bipartite graph check task.

Topological Sort

```
# Assuming G is already defined and is a directed graph
# Find one topological sorting path of the graph
topological_order = list(nx.topological_sort(G))
# Convert the list to a string representation
answer = ' -> '.join(map(str, topological_order))
```

Figure 13: An example of the final code C' generated for the topological sort task.

Shortest Path

```
# Calculate the shortest path weight from node 11 to node 4
shortest_path_length = nx.shortest_path_length(G, source=11, target=4, weight='weight')
answer = shortest_path_length
```

Figure 14: An example of the final code C' generated for the shortest path task.

Maximum Triangle Sum

```
from itertools import combinations
# Assume G is already defined as a directed graph
max_sum = -float('inf’)
max_nodes = None
# Iterate over all combinations of three nodes in the graph
for nodes in combinations(G.nodes, 3):
     a, b, c = nodes
     # Check if the three nodes are interconnected, i.e., all pairs have edges in both directions
     if (G.has_edge(a, b) and G.has_edge(b, a) and G.has_edge(b, c) and G.has_edge(c, b) and G.has_edge(a, c) and
     G.has_edge(c, a)):
          # Calculate the sum of their weights
          total_weight = (G.nodes[a].get('weight', 0) + G.nodes[b].get('weight', 0) + G.nodes[c].get('weight', 0))
          # Update the maximum sum and the corresponding node set if needed
          if total_weight > max_sum:
          max_sum = total_weight
          max_nodes = nodes
# Generate the answer string with the maximum sum
answer = f”Answer is {max_sum}."
```

Figure 15: An example of the final code C' generated for the maximum triangle sum task.

Maximum Flow

```
# Calculate the maximum flow from node 0 to node 8
flow_value, flow_dict = nx.maximum_flow(G, 0, 8)
# Store the result in the variable 'answer’
answer = str(flow_value)
```

Figure 16: An example of the final code C' generated for the maximum flow task.

Subgraph Matching

```
from networkx.algorithms import isomorphism
import itertools
# First, try to find a subgraph isomorphic to G_prime using the standard DiGraphMatcher
matcher = isomorphism.DiGraphMatcher(G, G_prime)
if matcher.subgraph_is_isomorphic():
     # We found a subgraph isomorphic to G_prime (this covers the node-induced scenario)
     answer = "Yes, G_prime is a subgraph of G."
else:
     # If no node-induced subgraph match was found, we try edge-induced.
     # An edge-induced subgraph of G is defined by a subset of edges of G.
     # We must find a set of edges in G that, when taken alone (with their endpoints),
     # forms a subgraph isomorphic to G_prime.
     # Get edges and nodes of G_prime for reference
     G_prime_edges = list(G_prime.edges())
     G_prime_edge_count = len(G_prime_edges)
     G_prime_nodes = set(G_prime.nodes())
     found_edge_induced = False
     for edge_subset in itertools.combinations(G.edges(), G_prime_edge_count):
          # Build a subgraph of G induced by these edges
          sub_nodes = set()
          for u, v in edge_subset:
               sub_nodes.add(u)
               sub_nodes.add(v)
          # Create a new directed graph with just these edges
          sub_G = G.subgraph(sub_nodes).copy()
          # Remove all edges not in edge_subset
          sub_G.remove_edges_from([e for e in sub_G.edges() if e not in edge_subset])
          # Now check if sub_G is isomorphic to G_prime
          test_matcher = isomorphism.DiGraphMatcher(sub_G, G_prime)
          if test_matcher.is_isomorphic():
                    found_edge_induced = True
                    break
     if found_edge_induced:
          answer = "Yes, G_prime is a subgraph of G."
     else:
          answer = "No, G_prime is not a subgraph of G."
```

Figure 17: An example of the final code C' generated for the subgraph matching task.

Indegree Calculation

```
# Assuming G is already defined as a directed graph in the existing code
indegree_node_5 = G.in_degree(5)
answer = f"{indegree_node_5}"
```

Figure 18: An example of the final code C' generated for the indegree calculation task.

Outdegree Calculation

```
# Calculate the outdegree of node 2
outdegree_node_2 = G.out_degree(2)
# Create a string with the result
answer = f"{outdegree_node_2}."
```

Figure 19: An example of the final code C' generated for the outdegree calculation task.

Maximum Flow (Cypher)

```
" MATCH ()-[r:CONNECTS_flow_8]->() SET r.flow = 0 ”

" // 1. Find any path from source(id=2) to sink(id=6)
//    where every edge still has spare capacity (weight > flow)
MATCH path = (source:Node_flow_8 {id: 2})-[:CONNECTS_flow_8*]->(sink:Node_flow_8 {id: 6})
WHERE ALL(r IN relationships(path) WHERE r.weight > r.flow)

// 2. Compute the minimum “residual capacity” along that path
WITH
  path,
  REDUCE(
    minCap = toFloat(INFINITY),
    r IN relationships(path) |
      CASE
        WHEN (r.weight - r.flow) < minCap
          THEN (r.weight - r.flow)
        ELSE
          minCap
      END
  ) AS residualCapacity

// 3. Augment: for each edge in the path, add the residualCapacity to its flow
WITH path, residualCapacity
UNWIND relationships(path) AS r
SET r.flow = r.flow + residualCapacity; "
" MATCH (source:Node_flow_8 {id: 2})-[r:CONNECTS_flow_8]->() RETURN SUM(r.flow) AS maxFlow "
```

Figure 20: An example of the final code C' in Cypher query by GARRF$_C$ generated for the maximum flow task.

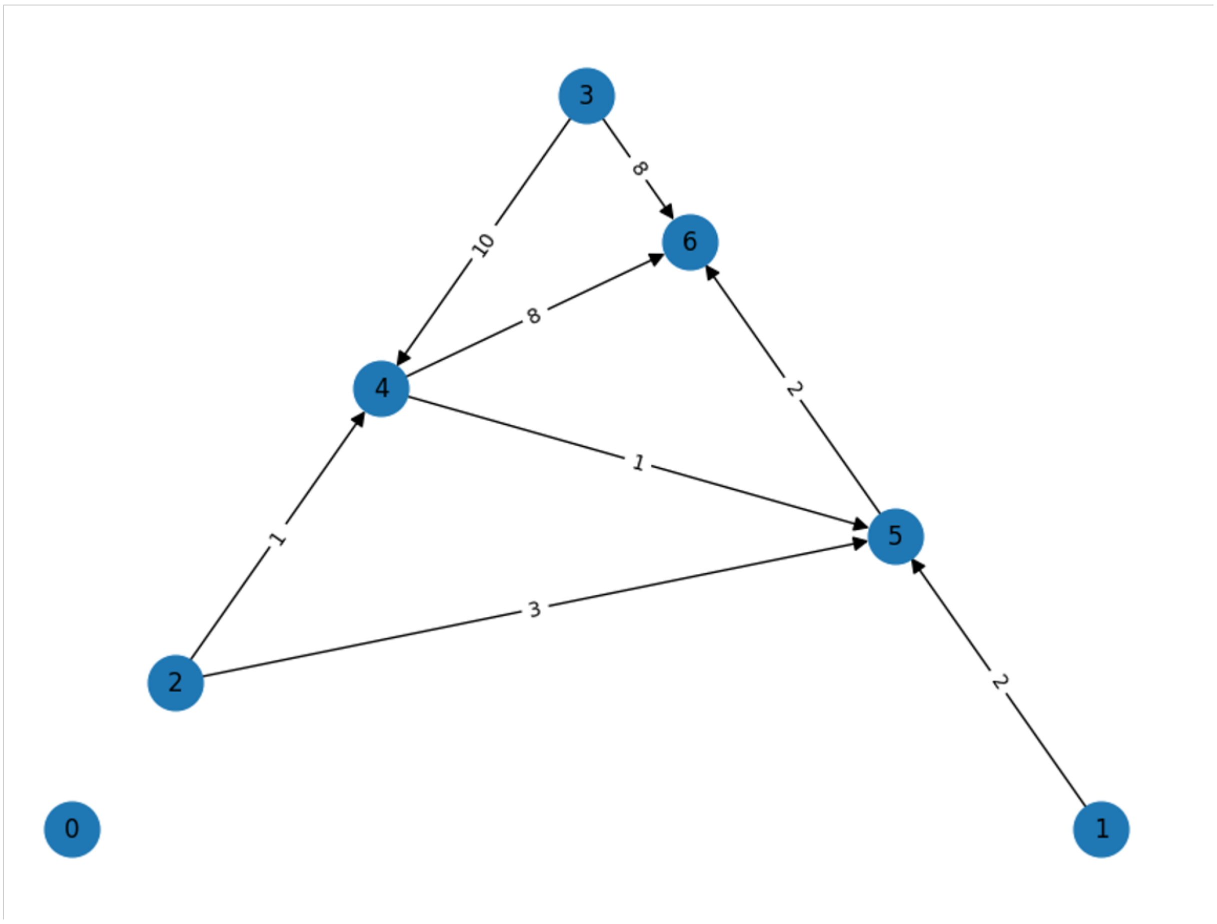


Figure 21: An example directed graph with edge weights. The correct maximum flow from node 2 to 6 is 3 but the Cypher query in Figure 20 returns 4 as the answer.

Topological Sort

```
// 1. Find all "root" nodes (nodes with no incoming CONNECTS_topology_0 edges)
MATCH (n:Node_topology_0)
WHERE NOT (n)<-[:CONNECTS_topology_0]-()

// 2. For each root, traverse a spanning tree following outgoing CONNECTS_topology_0 relationships
CALL apoc.path.spanningTree(
  n,
  {
    relationshipFilter: 'CONNECTS_topology_0>',
    labelFilter: 'Node_topology_0',
    bfs: false
  }
) YIELD path

// 3. Extract all nodes from each tree path
WITH nodes(path) AS nds
UNWIND nds AS n

// 4. Deduplicate and collect the node IDs into a list (your "topoSort")
WITH DISTINCT n
RETURN collect(n.id) AS topoSort;
```

Figure 22: An example of the final code C' in Cypher query by GARRF$_C$ generated for the topological sort task.