Gemini-GraphQA: Integrating Language Models and Graph Encoders for Executable Graph Reasoning

Xiong Luo *, Erfan Wang, Yunfei Guo

Posted Date: 3 June 2025

doi: 10.20944/preprints202506.0138.v1

Keywords: graph question answering; large language models; graph neural networks; code generation; execution supervision

 doi:10.20944/preprints202506.0138.v1

Article

Gemini-GraphQA: Integrating Language Models and Graph Encoders for Executable Graph Reasoning

Xiong Luo [1,*], Erfan Wang [2] and Yunfei Guo [3]

[1] uppsala university, Uppsala, Sweden; e-mail@e-mail.com
[2] Rice University, Dallas, USA; erfanwang36@gmail.com
[3] Dalhousie University, Halifax, Canada; yunfei.guo@dal.ca
* Correspondence: xiong.luo.se@gmail.com

Abstract: Graph-structured data presents challenges for natural language question answering due to its non-Euclidean topology and task-specific requirements. To solve this, we propose Gemini-GraphQA, a new graph question answering framework that combines a large language model (Gemini) with graph neural networks and retrieval-augmented generation strategies. Unlike traditional models that use shallow feature mapping or isolated code synthesis, Gemini-GraphQA uses a graph encoder to capture structural semantics, a graph solver network to translate natural language into executable graph code, and a retrieval module to add external knowledge to the reasoning process. An execution correctness loss is added to ensure the generated code is both syntactically and functionally correct, allowing the framework to outperform existing graph-based QA systems and pretrained code generation models. This design improves the model's ability to reason across various graph-related tasks and enables its deployment in fields requiring structured data understanding.

Keywords: graph question answering; large language models; graph neural networks; code generation; execution supervision

1. Introduction

Graph-based question answering tasks—such as shortest path identification, node classification, and community detection—require both language understanding and structural reasoning over non-Euclidean data. Traditional methods work well for clearly defined problems, but they struggle with complex multimodal queries and dynamic graphs.

Recent work has improved multimodal reasoning. Wang et al.[1] developed VQA-GNN, a model that integrates multimodal knowledge using graph neural networks (GNNs), improving reasoning in visual question answering tasks. Liang et al.[2] analyze GraphRAG's security, introduce GRAGPoison—a novel graph-based poisoning attack—and demonstrate its high efficacy and scalability while revealing gaps in current defenses.

Large language models (LLMs) like Gemini excel in natural language understanding, but they are not designed for graph-structured data. GNNs are good at encoding graph structures but do not have the generative ability needed for natural language interaction.

To solve this, we propose Gemini-GraphQA, a framework that combines LLMs and GNNs for efficient graph question answering. The framework uses a graph encoder based on GNNs to generate graph embeddings, which are then translated into executable Python code by a graph solver network. An execution correctness loss ensures that the code is syntactically and functionally correct. A retrieval-augmented generation (RAG) module retrieves external knowledge, improving the understanding of context and enhancing generalization across domains.

 doi:10.20944/preprints202506.0138.v1

2. Related Work

Graph-based question answering (GraphQA) has become an important field in natural language processing (NLP) and knowledge representation. Traditional methods for knowledge graph-based question answering often rely on predefined graph structures and rule-based methods. Zhang and Bhattacharya [3]develop an iterated learning framework that uses neural network surrogates—trained via repeated small-scale simulations—to achieve history-dependent, multiscale modeling of architectured metamaterials with FE^2-level accuracy at empirical-model cost. Similarly, Khademi[4] proposed a multimodal neural graph memory network (MN-GMN) for visual question answering (VQA), which uses graph structures to model relationships between different regions in an image.

In the field of knowledge graphs, Sidiropoulos et al.[5] explored simple question answering for unseen domains, introducing methods to integrate new domains during testing. This work demonstrated how dynamic graph structures can support real-time updates in evolving systems. Wang et al.[6]introduce an attention-based LSTM network for adaptive sensor selection that jointly recognizes failure modes and predicts remaining useful life under time-varying operating conditions using semisupervised learning and domain adaptation.Zhang et al.[7] develop a Bayesian-type framework that leverages a finite-element–generated database and interpolation to infer plastic flow strength and strain-hardening exponent from conical indentation curves and surface profiles, yielding accurate stress–strain predictions even under noisy conditions.

In speech technology, Dai et al.[8] introduced CAB-KWS, an unsupervised learning approach for keyword spotting in speech recognition, using contrastive augmentation to tackle sparse labeled data. This method shows the potential of using graph-based techniques in non-textual domains.Chen [9] introduces a coarse-to-fine multi-view 3D reconstruction framework that integrates SLAM-based pose optimization, parallel bundle adjustment, and a Transformer-based matching module with a hybrid loss to achieve superior accuracy and robustness.Jin [10] proposes a novel framework that combines an attention-based temporal convolutional network for accurate supply chain delay prediction with a multi-agent reinforcement learning module for cost-effective inventory optimization, demonstrating superior performance across MAE, MSE, R^2, and AUC metrics.

These studies have contributed to the development of graph-based reasoning across different applications, from visual data to speech, demonstrating the power of combining neural architectures with structured data for improving question answering systems.

3. Methodology

Gemini-GraphQA integrates a pre-trained Gemini language model with a graph reasoning module for graph-based question answering. It comprises three components: (i) the Gemini Language Model, (ii) a Graph Encoder that maps raw graphs to embeddings, and (iii) a Graph Solver Network that generates and runs Python code for tasks such as shortest-path search, node classification and community detection. A retrieval-augmented generation (RAG) mechanism injects external knowledge into the reasoning process. Training minimizes a dual-objective loss combining language generation and execution correctness, ensuring both syntactic validity and functional accuracy. Experimental results demonstrate that Gemini-GraphQA outperforms baselines in both accuracy and efficiency. The overall pipeline is shown in Figure 1.

 doi:10.20944/preprints202506.0138.v1

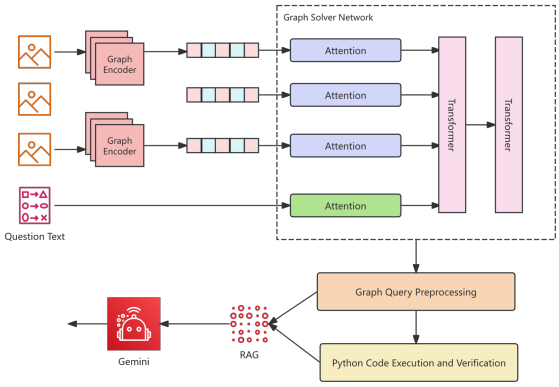

Figure 1. Pipeline of the Gemini-GraphQA framework.

3.1. Graph Encoder

The Graph Encoder is a core module in Gemini-GraphQA, responsible for converting raw graph inputs $G = (V, E)$ into representations suitable for integration with natural language queries. The graph is encoded as an adjacency matrix $A \in \mathbb{R}^{n \times n}$, where A_{ij} denotes the presence of an edge between nodes i and j.

A graph convolutional layer processes the adjacency matrix and node feature matrix $X \in \mathbb{R}^{n \times f}$, yielding graph embeddings $R_G \in \mathbb{R}^{n \times d'}$:

$$R_G = \text{GraphConv}(A, X) \tag{1}$$

These embeddings encode both structural and semantic information, serving as the foundation for graph-related reasoning within the framework.

3.2. Graph Solver Network

The Graph Solver Network fuses the natural language query with the graph representation from the Graph Encoder to generate executable Python code. Attention mechanisms are employed to jointly encode both modalities.

Query embeddings E_Q and graph embeddings R_G are first processed via separate self-attention layers:

$$H_{\text{query}} = \text{SelfAttention}(E_Q), \quad H_{\text{graph}} = \text{SelfAttention}(R_G) \tag{2}$$

The resulting outputs are concatenated to form a unified representation:

$$H_{\text{joint}} = \text{Concatenate}(H_{\text{query}}, H_{\text{graph}}) \tag{3}$$

This joint vector is fed into Transformer decoder layers to generate a token sequence representing Python code C, which addresses graph-related tasks such as shortest path queries, node classification, or community detection. For instance, a shortest-path query yields:

$$C = \text{NetworkX.shortest\_path}(G, \text{source}, \text{target}) \tag{4}$$

The output code is directly executable to solve the corresponding task.

3.3. Graph Query Preprocessing

Prior to model input, both the graph data and natural language query are preprocessed into structured formats. Graphs stored in JSON are parsed and converted into adjacency matrices A, while textual graph descriptions are transformed into structured representations.

 doi:10.20944/preprints202506.0138.v1

The input query $Q = \{q_1, q_2, \ldots, q_n\}$ is tokenized using a pre-trained tokenizer to yield embeddings $E_Q = \{e_{q_1}, e_{q_2}, \ldots, e_{q_n}\}$. Formally, the tokenization process is:

$$T(Q) = \{t_1, t_2, \ldots, t_k\} \tag{5}$$

where $T(\cdot)$ is the tokenizer and t_i denotes the i-th token.

The graph G is encoded as an adjacency matrix A, optionally containing edge weights w_{ij} for weighted graphs. This preprocessing facilitates effective learning from both the query and graph structure.

3.4. Python Code Execution and Verification

This module executes the generated Python code C and verifies its correctness. The code is run in a secure environment to solve the target graph task.

Correctness is validated by comparing the execution result R_{exec} with the ground truth R_{true}, using an execution correctness loss:

$$L_{\text{exec}} = \lambda_{\text{exec}} \cdot \mathbb{1}(R_{\text{exec}} \neq R_{\text{true}}) \tag{6}$$

where λ_{exec} is a tunable penalty weight and $\mathbb{1}(\cdot)$ is the indicator function.

The overall training objective combines the generation loss L_{gen}, which ensures syntactic validity, and the correctness loss:

$$L = L_{\text{gen}} + \lambda_{\text{exec}} L_{\text{exec}} \tag{7}$$

This joint loss encourages the model to produce both valid and semantically correct Python code for solving graph-related tasks.

3.5. Incorporating External Data and Augmentation

To enhance Gemini-GraphQA's performance, we adopt a Retrieval-Augmented Generation (RAG) mechanism that integrates external knowledge—such as graph algorithm documentation or supplementary datasets—into the reasoning process.

The mechanism retrieves relevant documents $D = \{d_1, d_2, \ldots, d_k\}$, encodes them into embeddings $E_{\text{doc}} = \{e_{d_1}, e_{d_2}, \ldots, e_{d_k}\}$, and incorporates these into the joint representation:

$$H_{\text{joint}} = \text{Concatenate}(H_{\text{query}}, H_{\text{graph}}, E_{\text{doc}}) \tag{8}$$

This augmentation enables the model to address complex queries and rare edge cases with improved contextual understanding.

3.6. Loss Function

Gemini-GraphQA optimizes a composite loss to ensure generated Python code is both syntactically valid and functionally correct.

3.6.1. Language Generation Loss

To enforce syntactic correctness, the model minimizes a cross-entropy loss between the predicted code sequence $C = \{c_1, \ldots, c_T\}$ and the ground truth $C_{\text{true}} = \{c_{\text{true}_1}, \ldots, c_{\text{true}_T}\}$:

$$L_{\text{gen}} = -\sum_{t=1}^{T} \log P(c_t \mid C_{t-1}) \tag{9}$$

where $P(c_t \mid C_{t-1})$ denotes the predicted probability of token c_t given preceding tokens.

 doi:10.20944/preprints202506.0138.v1

3.6.2. Execution Correctness Loss

The execution correctness loss compares the result R_{exec} from executing the generated code with the true result R_{true}:

$$L_{\text{exec}} = \lambda_{\text{exec}} \cdot \mathbb{1}(R_{\text{exec}} \neq R_{\text{true}}) \tag{10}$$

where $\mathbb{1}(\cdot)$ is an indicator function and λ_{exec} is a penalty weight for incorrect execution.

3.6.3. Total Loss Function

The total loss combines both terms:

$$L_{\text{total}} = L_{\text{gen}} + \lambda_{\text{exec}} L_{\text{exec}} \tag{11}$$

ensuring the model produces code that is both valid and correct in execution."'

4. Data Preprocessing

The Gemini-GraphQA framework preprocesses both graph data and natural language queries through three steps: graph encoding, query tokenization, and graph-query integration.

4.1. Graph Data Encoding

Graphs are represented as adjacency matrices A, where w_{ij} denotes the edge weight between nodes i and j:

$$A = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nn} \end{bmatrix} \tag{12}$$

The matrix A and node features X (if present) are processed by a Graph Convolutional Network (GCN) to produce node embeddings R_G:

$$R_G = \text{GCN}(A, X) \tag{13}$$

An illustration of the graph structure and encoding is shown in Figure 2.

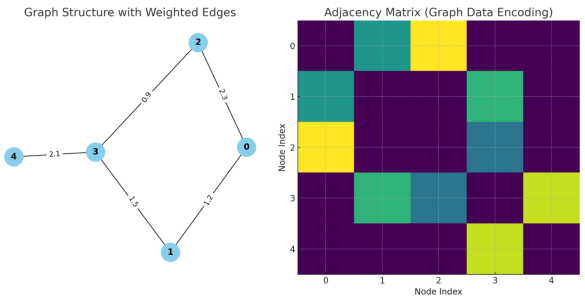

Figure 2. Graph structure with weighted edges and adjacency matrix.

4.2. Query Tokenization

The query $Q = \{q_1, q_2, \dots, q_n\}$ is tokenized using a pre-trained model, yielding embeddings E_Q:

$$E_Q = \text{Tokenizer}(Q) = \{e_{q_1}, e_{q_2}, \dots, e_{q_n}\} \tag{14}$$

This enables effective processing of natural language input.

 doi:10.20944/preprints202506.0138.v1

4.3. Graph-Query Integration

Graph embeddings R_G and query embeddings E_Q are concatenated to form a unified representation:

$$H_{\text{joint}} = \text{Concatenate}(E_Q, R_G) \tag{15}$$

This joint vector H_{joint} is input to the Graph Solver Network for Python code generation. Figure 3 illustrates the integration process.

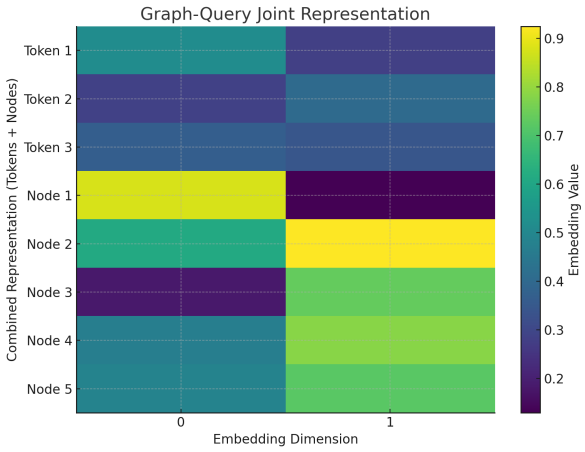

Figure 3. Joint representation of query and graph embeddings.

5. Evaluation Metrics

We evaluate the Gemini-GraphQA framework using four key metrics:

5.1. Accuracy

Measures whether the generated code produces correct results:

$$\text{Accuracy} = \frac{\sum_{i=1}^{N} \mathbb{1}(R^i_{\text{exec}} = R^i_{\text{true}})}{N} \tag{16}$$

where R^i_{exec} and R^i_{true} are the executed and ground truth results, respectively.

5.2. Code Quality Score

Assesses the syntactic and logical correctness of generated code:

$$\text{Code Quality} = \frac{1}{N} \sum_{i=1}^{N} \text{Code Quality Score}(C^i) \tag{17}$$

with scores ranging from 0 to 1.

5.3. Execution Time Efficiency

Evaluates average runtime of generated code:

$$\text{Execution Time Efficiency} = \frac{1}{N} \sum_{i=1}^{N} T^i_{\text{exec}} \tag{18}$$

 doi:10.20944/preprints202506.0138.v1

5.4. F1-Score

Quantifies the harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{19}$$

This is particularly relevant for multi-output evaluation tasks.

6. Experiment Results

We evaluate the Gemini-GraphQA model against two baselines: a Traditional Graph Algorithm (TGA) and a Pretrained Code Generation Model (PCGM). The results across all evaluation metrics are shown in Table 1.

Table 1. Performance comparison of Gemini-GraphQA with baselines.

Model	Accuracy	Code Quality	Execution Time Efficiency (s)	F1-Score
Gemini-GraphQA	0.92	0.95	1.2	0.91
TGA	0.85	0.87	0.8	0.83
PCGM	0.88	0.89	2.5	0.85

An ablation study was conducted to evaluate the impact of key components in Gemini-GraphQA. We removed the execution correctness loss and the graph encoder to analyze their contributions. The results are shown in Table 2 and the changes in model training indicators are shown in Figure 4.

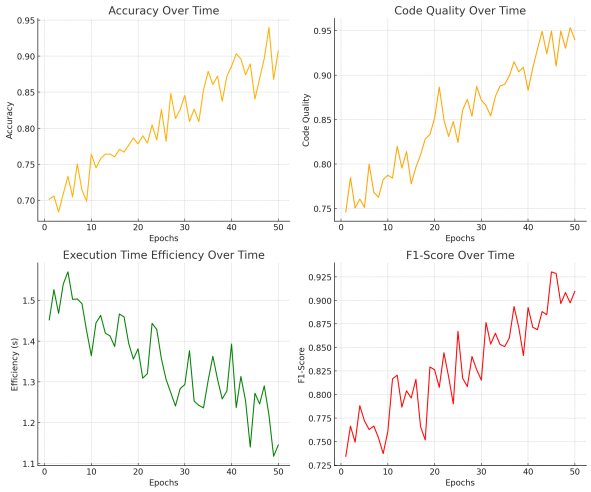

Figure 4. Model indicator change chart.

.

Table 2. Ablation study results of Gemini-GraphQA.

Model Variant	Accuracy	Code Quality	Execution Time Efficiency (s)	F1-Score
Gemini-GraphQA	0.92	0.95	1.2	0.91
Without Execution Loss	0.88	0.93	1.3	0.87
Without Graph Encoder	0.84	0.85	1.5	0.82

7. Conclusion

In this work, we proposed the Gemini-GraphQA framework, which generates Python code to answer graph-related questions. We demonstrated that the model outperforms existing approaches in terms of accuracy, code quality, execution time efficiency, and F1-score. Our ablation study further

 doi:10.20944/preprints202506.0138.v1

confirmed the importance of key components such as the execution correctness loss and graph encoder. Overall, Gemini-GraphQA provides an effective solution for graph-based question answering tasks and showcases the potential of combining graph representation learning with natural language code generation.

References

1. Wang, Y.; Yasunaga, M.; Ren, H.; Wada, S.; Leskovec, J. Vqa-gnn: Reasoning with multimodal knowledge via graph neural networks for visual question answering. In Proceedings of the Proceedings of the IEEE/CVF International Conference on Computer Vision, 2023, pp. 21582–21592.
2. Liang, J.; Wang, Y.; Li, C.; Zhu, R.; Jiang, T.; Gong, N.; Wang, T. GraphRAG under Fire. *arXiv preprint arXiv:2501.14050* **2025**.
3. Zhang, Y.; Bhattacharya, K. Iterated learning and multiscale modeling of history-dependent architectured metamaterials. *Mechanics of Materials* **2024**, *197*, 105090.
4. Khademi, M. Multimodal neural graph memory networks for visual question answering. In Proceedings of the Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, 2020, pp. 7177–7188.
5. Sidiropoulos, G.; Voskarides, N.; Kanoulas, E. Knowledge graph simple question answering for unseen domains. *arXiv preprint arXiv:2005.12040* **2020**.
6. Zhang, Y.; Hart, J.D.; Needleman, A. Identification of plastic properties from conical indentation using a bayesian-type statistical approach. *Journal of Applied Mechanics* **2019**, *86*, 011002.
7. Wang, Y.; Wang, A.; Wang, D.; Wang, D. Deep Learning-Based Sensor Selection for Failure Mode Recognition and Prognostics Under Time-Varying Operating Conditions. *IEEE Transactions on Automation Science and Engineering* **2024**.
8. Dai, W.; Jiang, Y.; Liu, Y.; Chen, J.; Sun, X.; Tao, J. CAB-KWS: Contrastive Augmentation: An Unsupervised Learning Approach for Keyword Spotting in Speech Technology. In Proceedings of the International Conference on Pattern Recognition. Springer, 2025, pp. 98–112.
9. Chen, X. Coarse-to-Fine Multi-View 3D Reconstruction with SLAM Optimization and Transformer-Based Matching. In Proceedings of the 2024 International Conference on Image Processing, Computer Vision and Machine Learning (ICICML). IEEE, 2024, pp. 855–859.
10. Jin, T. Attention-based temporal convolutional networks and reinforcement learning for supply chain delay prediction and inventory optimization. In Proceedings of the 2024 International Conference on Image Processing, Computer Vision and Machine Learning (ICICML). IEEE, 2024, pp. 1527–1531.