GraphEval2000: Benchmarking and Improving Large Language Models on Graph Datasets

Qiming Wu*
UC Santa Barbara
qimingwu@cs.ucsb.edu

Zichen Chen*
UC Santa Barbara
zichen_chen@ucsb.edu

Will Corcoran
UC Santa Barbara
wcorcoran@ucsb.edu

Misha Sra
UC Santa Barbara
sra@ucsb.edu

Ambuj K. Singh
UC Santa Barbara
ambuj@cs.ucsb.edu

Abstract

Large language models (LLMs) have achieved remarkable success in natural language processing (NLP), demonstrating significant capabilities in processing and understanding text data. However, recent studies have identified limitations in LLMs' ability to reason about graph-structured data. To address this gap, we introduce GraphEval2000, the **first** comprehensive graph dataset, comprising 40 graph data structure problems along with 2000 test cases. Additionally, we introduce an evaluation framework based on GraphEval2000, designed to assess the graph reasoning abilities of LLMs through coding challenges. Our dataset categorizes test cases into four primary and four sub-categories, ensuring a comprehensive evaluation. We evaluate eight popular LLMs on GraphEval2000, revealing that LLMs exhibit a better understanding of directed graphs compared to undirected ones. While private LLMs consistently outperform open-source models, the performance gap is narrowing. Furthermore, to improve the usability of our evaluation framework, we propose Structured Symbolic Decomposition (SSD), an instruction-based method designed to enhance LLM performance on GraphEval2000. Results show that SSD improves the performance of GPT-3.5, GPT-4, and GPT-4o on complex graph problems, with the increase of 11.11%, 33.37%, and 33.37%, respectively.

1 Introduction

Large language models (LLMs) GPTs [Achiam et al., 2023, Brown et al., 2020a, Chen et al., 2021], Gemini-Pro [Team et al., 2023], Gemini-1.5 [Reid et al., 2024], Claude-3-Haiku, Claude-3-Sonnet, Claude-3-Opus [Anthropic, 2024], LLaMA-3-70b [Touvron et al., 2023], and Mixtral-8x7b [Jiang et al., 2024] have achieved remarkable success in solving a wide range of natural language processing (NLP) tasks. For example, question answering [Devlin et al., 2018, Brown et al., 2020b, Raffel et al., 2020]), machine translation [Raffel et al., 2020, Brown et al., 2020b], text classification [Raffel et al., 2020, Yang et al., 2019, Liu et al., 2019], and text generation [Yang et al., 2019, Achiam et al., 2023].

However, their performance on complex graph reasoning tasks has been notably inadequate [Zhang, 2023]. Current research highlights that while LLMs can handle basic graph-related queries, they have a challenge with more complex graph structures and multi-step reasoning processes [Liu and Wu, 2023, Wang et al., 2024, Creswell et al., 2022].

Submitted to the 38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks. Do not distribute.

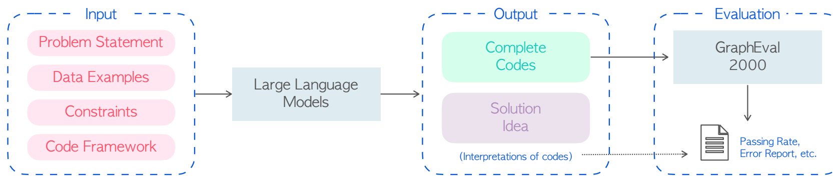


Figure 1: **Overview of the Evaluation Framework**. For each problem, we input problem statement, data examples, and code framework to LLMs. And then they complete the code and give explanations. Finally, the framework evaluates the code on `GraphEval2000` and returns the result details.

Recognizing the potential of leveraging LLMs' programming capabilities in computational contexts [Yang et al., 2024, Murphy et al., 2024], we propose to use their programming abilities to enhance reasoning on graphs. To this end, we introduce `GraphEval2000`, the **first dataset** designed to evaluate the graph reasoning capability of LLMs through coding challenges. It includes 40 data structure problems and 2,000 test cases. Each problem includes: (1) problem statement, (2) data examples, (3) constraints, and (4) code framework. The dataset has four main graph categories: Sparse, Planar, Regular, and Complete graphs. Within each main category, there are four sub-categories: connected, disconnected, cyclic, and acyclic graphs. Based on `GraphEval2000`, we propose the evaluation framework, which provides real-time feedback to users. The entire framework is illustrated in Figure 1. Unlike traditional coding challenges (e.g., LeetCode) that obscure test case details [Hou and Ji, 2024, Hu et al., 2024], the framework will return failed test cases along with excution results.

To better help readers use our framework to evaluate and improve the performance of LLMs, we propose an instruction-based method, Structured Symbolic Decomposition (`SSD`). `SSD` is designed to enhance LLMs' ability to understand and solve complex graph problems. Drawing inspiration from human cognitive strategies [Paas and van Merriënboer, 2020], `SSD` decomposes complex tasks into a "cognitive step" and an "action step", thereby improving model comprehensions. Experiments demonstrate that `SSD` significantly boosts the performance of GPT-3.5, GPT-4, and GPT-4o in hard-level graph problems, yielding improvements of 11.11%, 33.37%, and 33.37%, respectively. We hope this example inspires the community to further utilize our framework to explore LLMs' reasoning abilities on graphs.

Our contributions are summarized as follows:

- We propose `GraphEval2000`, the **first graph dataset** specifically designed to evaluate the graph reasoning abilities of LLMs through code. This dataset includes 40 data structure problems and 2000 graph test cases. It is organized into four primary graph categories, each containing four sub-categories, ensuring a diverse representation of graph structures;
- Based on `GraphEval2000`, we propose an evaluation framework to systematically assess the graph reasoning abilities of LLMs. This framework not only tests the models but also provides real-time feedback, enabling users to iteratively improve their models' performance;
- We have established the **first benchmark** for LLMs on graph data structure problems, involving eight of the most popular LLMs. Our experiment reveals that LLMs understand directed graph structures better than undirected ones. While open-source models generally underperform relative to private LLMs, the performance gap is narrow, and in certain graph categories, they demonstrate comparable performance.
- To enhance the usability of our evaluation framework and `GraphEval2000`, we proposed Structured Symbolic Decomposition (`SSD`), an instruction-based method. `SSD` decomposes complicated problems into distinct reasoning components, facilitating better understanding by LLMs. Experimental results demonstrate the effectiveness of SSD, resulting in an average 25% improvement in performance for GPT-3.5, GPT-4 and GPT-4o.

2 Related Work

2.1 LLMs on Graph Reasoning

Recent research has investigated the application of LLMs for handling graph data and reasoning tasks. These studies have utilized LLMs as predictors to generate answers for graph problems. Currently, the proposed methods can be classified into the following two categories:

Direct Answering LLMs are provided with a graph and are required to return answers to related questions. Researchers have found that LLMs can provide a preliminary understanding of simple graph structures and answer basic questions such as "count the number of nodes", "identify node degrees", and "determine connectivity". However, their performance declines significantly when the graph structures become complex or when more intricate reasoning is required [Jin et al., 2023, Wang et al., 2024, Liu and Wu, 2023, Guo et al., 2023]. Notably, providing additional examples in a few-shot learning scenario can improve performance on easier problems, although the improvement is marginal [Zhang et al., 2023b, Fatemi et al., 2023, Chai et al., 2023].

Heuristic Reasoning Due to the difficulty for LLMs to directly produce correct answers based on the input, and inspired by the Chain-of-Thought (CoT) method [Wei et al., 2022, Kojima et al., 2022], researchers have proposed a step-by-step reasoning approach for graph problems, known as zero-shot CoT. This technique has been widely studied [Chai et al., 2023, Wang et al., 2024, Liu and Wu, 2023, Guo et al., 2023, Zhang et al., 2023b, Fatemi et al., 2023, Sun et al., 2023], but the observed improvements remain limited.

2.2 Enhancing LLM Reasoning through Code Utilization

Recent research has demonstrated that LLMs exhibit superior reasoning capabilities in code-prompted scenarios compared to text-prompted ones [Suzgun et al., 2022, Liang et al., 2023, Hendy et al., 2023]. Researchers have transformed natural language problems into code prompts to facilitate better interaction with LLMs, thereby enhancing their reasoning abilities [Madaan et al., 2022, Zhang et al., 2023a, Bi et al., 2024, Dong et al., 2022, Yan et al., 2023]. This insight leads us to evaluate the graph reasoning abilities of LLMs on data structure problems by assessing the code generated by these models. Moreover, by prompting LLMs to generate more accurate code, we aim to further improve their reasoning capabilities on graphs, thereby advancing the research in this area.

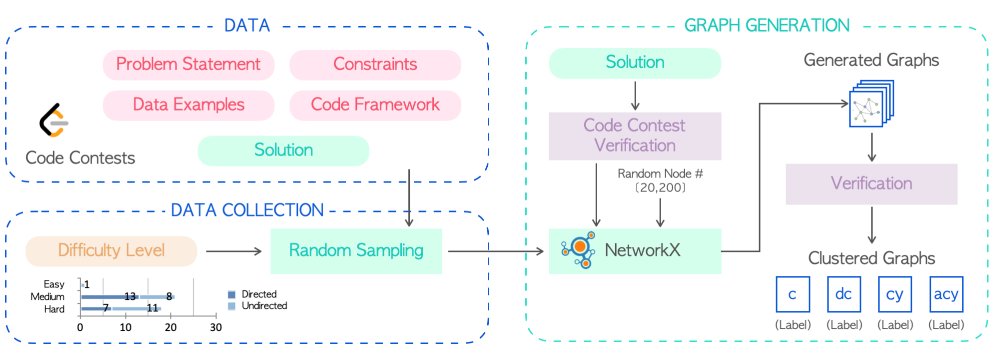


Figure 2: The overview pipeline of `GraphEval2000` dataset construction.

3 Dataset Construction

3.1 Data Collection

Problem Set We collected a total of 40 graph data structure problems from the LeetCode website (`https://leetcode.com/tag/graph/`), comprising 20 undirected and 20 directed graph problems.

Most of these problems are recently released, which minimizes the likelihood of their inclusion in the training sets of the verification LLMs. The majority of these problems are classified as medium- and hard-level according to their difficulty.

For each selected problem, we collected the following contents: problem statement, input/output examples, data constraints, and code framework. This information, accessible on the LeetCode website, was then stored in the format of JSONL strings. Finally, based on the keywords in the problem statement, we classified these problems as either directed or undirected graphs. The whole process is summarized in the left part of Figure 2.

3.2 Graph Generation

Consider a graph $G = (V, E)$, where V denotes the set of vertices and E denotes the set of edges in the graph. We classify graphs into four main categories: Sparse Graph, Planar Graph, Regular Graph, and Complete Graph. Below, we provide definitions of these graphs using mathematical notations.

Sparse Graph A graph $G = (V, E)$ is considered sparse if the number of edges $|E|$ is much less than the maximum possible number of edges, which is

$$\begin{cases} |E| \ll \frac{|V|(|V|-1)}{2} & \text{(undirected graph)}, \\ |E| \ll |V|(|V|-1) & \text{(directed graph)}. \end{cases} \tag{1}$$

Planar Graph A planar graph is one that can be drawn such that no edges intersect at any point that is not a vertex. For a finite, connected planar graph with F faces, the following holds:

$$|V| - |E| + |F| = 2 \tag{2}$$

Regular Graph A graph G is k-regular if every vertex has the same degree k. That is,

$$\forall v \in V, \, \deg(v) = k. \tag{3}$$

Complete Graph A graph G is complete if there is an edge between every pair of distinct vertices. The number of edges in a complete graph with $|V| = n$ is:

$$\begin{cases} |E| = \frac{n(n-1)}{2} & \text{(undirected graph)}, \\ |E| = n(n-1) & \text{(directed graph)}. \end{cases} \tag{4}$$

Graph Classification To diversify the graph data and encompass a broader range of edge cases, we have developed four sub-categories within each main category. These sub-categories include connected graphs, disconnected graphs, cyclic graphs, and acyclic graphs. However, due to the inherent characteristics of the main categories, some sub-categories require adjustments. For example, a planar graph inherently contains cycles, thus it cannot be classified as acyclic. After these adjustments, we have identified a total of 11 distinct types of graphs. The adjusted graph classification is presented in Figure 3.

To efficiently manage the graph generation process, we utilize "NetworkX" [Developers, 2024], a widely used Python package for creating and analyzing complex networks. Leveraging the NetworkX framework, we can easily construct both directed and undirected graphs according to our classification requirements. Specifically, we generate 10 graph samples for each sub-category, with the number of vertices in each graph being a random integer between 20 and 200. This approach allows us to create a graph dataset with varying levels of complexity.

As most recently-released problems do not have official answers, we select the solutions with the most votes for each problem as our label generation code. These solutions have been verified by the LeetCode contest. Subsequently, we run the solution code on graph test cases to generate the corresponding labels. The overview of this pipeline is summarized in the right part of Figure 2.

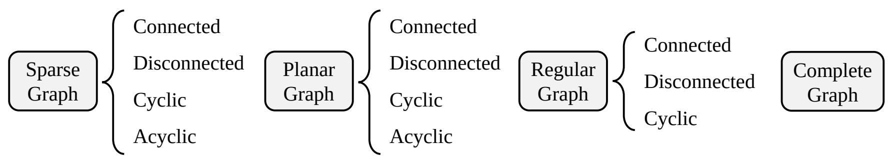


Figure 3: **Classification of Graph Test Cases**. The graph test cases are classified into four main categories: sparse graph, planar graph, regular graph, and complete graph. Each category is further divided into four sub-categories: connected graph, disconnected graph, cyclic graph, and acyclic graph. Note that sub-categories may be adjusted based on specific features of the main categories.

4 Evaluations

Experimental Setup For our experiments, we evaluate the reasoning abilities of LLMs using two approaches: assessment on the LeetCode platform and evaluation against our proposed dataset. The first approach employs the LeetCode platform to estimate LLM performance on graph data structure problems. This method, widely adopted by researchers, serves as a common benchmark to determine whether the model has comprehensively understood the problem [Hou and Ji, 2024]. In presenting the experimental results, we categorize the data into two parts: directed and undirected graphs. This division allows for a nuanced analysis of the LLMs' understanding of different graph structures.

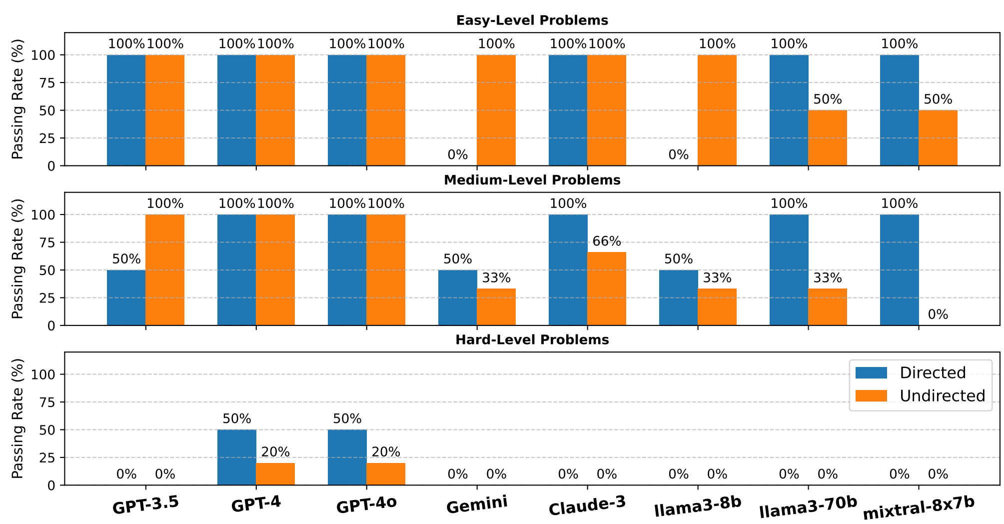


Figure 4: Evaluation results of LLMs on the LeetCode platform. This figure shows the passing rates of LLMs on selected graph data structure problems, categorized into 3 easy-level problems, 5 medium-level problems, and 9 hard-level problems.

Assessment on the LeetCode Platform We selected a total of 17 graph data structure problems from the LeetCode platform for evaluation, categorized into 3 easy-level, 5 medium-level, and 9 hard-level problems, with difficulty levels defined by LeetCode. The evaluation process involved incorporating the problem statement, data examples, and code framework into a prompt. The LLMs were then tasked with generating a complete code based on this prompt. The generated code was subsequently tested on the LeetCode platform to assess its accuracy and performance. The summarized results of this assessment are presented in Figure 4:

- For easy-level problems, most LLMs successfully pass the directed graph problems. Interestingly, private LLMs (e.g., the GPT family, Gemini, and Claude-3) outperform open-source LLMs (e.g., the LLaMA family and Mixtral-8x7b) on undirected graph problems.

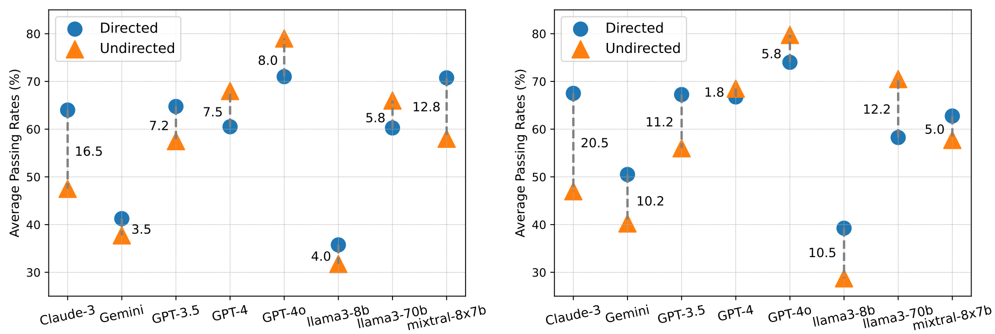
Table 1: **Evaluation Results on** `GraphEval2000`. This table presents the passing rates (%) of LLMs across various graph categories. The first row lists the names of the evaluated LLMs, while the first column categorizes the graphs as follows: "SG" denotes sparse graphs, "PG" denotes planar graphs, "RG" denotes regular graphs, and "CG" denotes complete graphs. Additionally, the following abbreviations are used to describe graph characteristics: "c" for connected graphs, "dc" for disconnected graphs, "cy" for cyclic graphs, and "acy" for acyclic graphs. This classification corresponds to Figure 3. The passing rates (%) are represented as "directed | undirected" for each graph category, where the number preceding the bar indicates the result for directed graphs, and the number following the bar indicates the result for undirected graphs. "NA" signifies that the category is not applicable. We bold the largest number in rach row.

		Claude-3 (Sonnet)	**Gemini (Pro)**	**GPT (3.5)**	**GPT (4)**	**GPT (4o)**	**llama3 (8b)**	**llama3 (70b)**	**mixtral (8x7b)**
SG	c	66 | 39	46 | 42	59 | 48	58 | 67	62 | 78	39 | 29	69 | 68	**76** | **54**
	dc	56 | 42	32 | 38	61 | 63	65 | 70	**82** | **83**	26 | 33	54 | 66	65 | 64
	cy	71 | 59	46 | 33	64 | 70	50 | 76	58 | **84**	42 | 41	67 | 72	**73** | 62
	acy	63 | 50	41 | 38	**75** | 49	69 | 59	82 | **71**	36 | 24	51 | 58	69 | 52
PG	c	**80** | 38	75 | 34	75 | 47	60 | 60	60 | **75**	60 | 26	75 | 77	60 | 53
	dc	65 | 50	39 | 45	64 | 58	78 | 76	**88** | **84**	29 | 28	52 | 62	65 | 62
	cy	68 | 52	46 | 36	62 | 70	58 | 76	67 | **83**	38 | 38	65 | 76	**65** | 59
	acy	57 | 48	42 | 46	68 | 49	71 | 62	**81** | **77**	30 | 23	41 | 67	61 | 57
RG	c	NA | 63	NA | 39	NA | 69	NA | 83	NA | **93**	NA | 52	NA | 78	NA | 72
	dc	NA | 67	NA | 39	NA | 61	NA | 89	NA | **96**	NA | 44	NA | 80	NA | 74
	cy	NA | **81**	NA | 50	NA | 74	NA | 87	NA | **99**	NA | 57	NA | 84	NA | 77
CG	c	NA | 33	NA | 50	NA | 67	NA | 86	NA | **88**	NA | 43	NA | 67	NA | 57

- For medium-level problems, LLMs demonstrate a better understanding of directed graph problems compared to undirected ones. Open-source LLMs perform worse than private LLMs, with a passing rate lower than 50%. However, private LLMs and open-source LLMs perform comparably on directed graph problems.
- For hard-level problems, only GPT-4 and GPT-4o achieve a 50% passing rate on directed graph problems and a 20% passing rate on undirected ones. All other LLMs fail to pass these problems, indicating that the GPT model exhibits the strongest reasoning ability among the tested LLMs.

Evaluating LLMs on `GraphEval2000` As illustrated in Figure 3, the `GraphEval2000` has four main categories and four sub-categories. However, due to the common constraints of input data in directed graph problems, we can only apply planar and sparse graphs to them. For example, many directed graph problems require a Directed Acyclic Graph (DAG) as input, making it impractical to apply regular and complex graphs to these problems. To analyze the results in Table 1 and Figure 5, we will answer the following questions: 1. *What is the performance difference between private and open-sourced models?* 2. *How are the performances of LLMs on directed and undirected graph problems?* 3. *How are the performances of LLMs on each main graph category?*

- Private models, especially GPT-4 and GPT-4o, consistently outperform open-sourced models across all graph categories. The performance gap is most pronounced in complex graph types such as regular and complete graphs, where private models achieve passing rates close to 100%, while open-sourced models like mixtral (8x7b) manage competitive but lower rates around 70-80%.



(a) Results on Sparse Graphs

(b) Results on Planar Graphs

Figure 5: Average passing rate on sparse and planar graphs. We mark the absolute value of the difference between results of directed and undirected graphs.

- LLMs generally perform better on directed graph samples than on undirected graph samples (as shown in Figure 5). The performance gap between private and open-sourced models is more pronounced in directed graph samples, highlighting the strength of private models in handling directed graph complexities. Private models, such as Claude-3 and GPT-3.5, typically exhibit a larger performance gap between directed and undirected graphs compared to open-source models. Interestingly, GPT-4 and GPT-4o seem to understand undirected graphs better than directed graphs.
- Private models consistently outperform across all graph categories, with GPT-4o leading in most cases and achieving near-perfect scores in complex graph types like regular and complete graphs. Open-sourced models like mixtral-8x7b shows competitive performance but remains behind private models. llama3 models, particularly llama3-8b, exhibit lower performance, indicating potential areas for improvement. The performance gap is most pronounced in directed graph samples and complex graph structures.

5 Unleashing the Power of LLMs on Graph Solving via `GraphEval2000`

To enhance the usability of our evaluation framework and `GraphEval2000`, we introduce Structured Symbolic Decomposition (`SSD`), an instruction-based method utilizing test cases from `GraphEval2000` for graph problems. Our approach aims to enable LLMs to perform better graph reasoning, especially, for hard-level problems.

5.1 Methodology

We hypothesize that breaking down complex graph problems into smaller, more manageable subproblems and turning them into symbolic forms [Dinu et al., 2024, Fang et al., 2024, Yang et al., 2024] will enhance the problem-solving capabilities of LLMs. Current methods rely on implicit knowledge and lack explicit guidance [Wei et al., 2022, Jin et al., 2024, Huang et al., 2024], leading to suboptimal performance, especially in complex scenarios. Our method mirrors human cognitive strategies [Paas and van Merriënboer, 2020, Romero et al., 2023], which simplify complex tasks by decomposing them into two parts: *cognitive step* and *action step*, thereby improving comprehension and facilitating more effective solutions. We selected the problems from `GraphEval2000` to be evaluation problems. The test cases from `GraphEval2000` are used for problem understanding and program testing.

5.1.1 Instructions for LLM

The instructions are composed of four parts: *problem clarification*, *problem breakdown*, *solution formulation*, and *program implementation*.

- **Problem Clarification:**
  1. Cognitive Step: *You must first understand and clearly articulate the problem, including all inputs and desired outputs.*
  2. Action Step: *Identify and list any specific rules, constraints, or conditions that influence the solution. Use the **[test case]** examples to assist the understanding.*

- **Problem Breakdown:**
  1. Cognitive Step: *Decompose the problem into smaller, manageable sub-problems, translating it into a symbolic form and identifying the key components and relationships within the problem.*
  2. Action Step: *Outline the sequential steps required to solve the overall problem.*

- **Solution Formulation:**
  1. Cognitive Step: *Formulate solving strategies using the symbolic form developed in the previous step and define the algorithms and methods needed to address each sub-problem.*
  2. Action Step: *Detail the algorithms and approaches for each sub-problem, ensuring they are logically connected and comprehensive.*

- **Program Implementation:**
  1. Cognitive Step: *Conceptualize the implementation of each solution component.*
  2. Action Step: *Write a program for each breakdown part, ensuring it aligns with the formulated strategy. Run **[test cases]** to verify the correctness of each component.*

We demonstrate the comparison of `SSD`-enhanced graph reasoning with CoT-based graph reasoning in Box 5.1.1. The LLM we used is GPT-4o. The demonstration involves a hard-level question and a test case from `GraphEval2000`. The reasoning processes are represented in green boxes ("`SSD`-enhanced GPT-4o's Graph Reasoning" and "CoT-based GPT-4o's Graph Reasoning"), and the code details are provided in Listings 1 and 2. Our results show that the `SSD`-enhanced GPT-4o reduces hallucinations and improves graph reasoning capabilities. In the following, we evaluate our method on a set of hard-level graph problem selected from `GraphEval2000`.

Graph Question - Hard Level

There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to $n - 1$. The edges in the graph are represented by a given 2D integer array *edges*, where *edges[i]* = [u_i, v_i] denotes an edge between vertex u_i and vertex v_i. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.
Return the length of the shortest cycle in the graph. If no cycle exists, return -1.
A cycle is a path that starts and ends at the same node, and each edge in the path is used only once.

Test Case Example

Input: {'edges': [[0, 1], [2, 4], [4, 0], [2, 1], [4, 1], [0, 2], [3, 2], [1, 3]]}, {'edges': [[0, 1], [0, 4], [1, 5], [4, 2], [3, 0], [2, 3], [5, 3], [4, 1], [5, 2]]}, {'edges': [[0, 1], [1, 2], [3, 1], [0, 3], [2, 3], [0, 2]]}, {'edges': [[0,1], [1,2]]}, {'edges': [[4,2],[5,1],[5,0],[0,3],[5,2],[1,4],[1,3],[3,4]]}
{'n': 5}, {'n': 6}, {'n': 4}, {'n': 4}, {'n': 6}
Output: {'label': 3}, {'label': 3}, {'label': 3}, {'label': -1}, {'label': 3}

✔ `SSD`-enhanced GPT-4o's Graph Reasoning

[GPT-4o's Response]
<Problem Clarification>
The goal is to find the length of the shortest cycle in a bi-directional graph, where:
1. Each vertex is labeled from 0 to $n - 1$.
2. Edges are given as a list of pairs, each representing a connection between two vertices.
Inputs: n: Number of vertices in the graph. *edges*: List of edges where each edge is a pair $[u_i, v_i]$ indicating a bi-directional connection between vertices u_i and v_i.
Outputs: The function returns the length of the shortest cycle if it exists, otherwise returns -1.
Constraints: ......
<Problem Breakdown>
1. Graph Representation: Convert the list of edges into an adjacency list to efficiently represent the graph. **2. Cycle Detection:** Implement a strategy to find and determine the length of the shortest cycle using BFS or DFS.
Steps to Solve:
1. Create an adjacency list from the given edges. 2. Traverse the graph from each vertex using BFS to find the shortest cycle involving that vertex. 3. Track and update the shortest cycle found during the traversals.
<Solution Formulation>
- **Breadth-First Search (BFS):** 1. Initialize an adjacency list from the edges. 2. For each vertex, perform a BFS to detect the shortest cycle starting from that vertex: ......
- **Handling Edge Cases:**......
<Program Implementation>
(the code is shown in Listing 1 5.1.1).

✘ CoT-based GPT-4o's Graph Reasoning

[GPT-4o's Response]
To solve the problem of finding the shortest cycle in an undirected graph, we can use the Breadth-First Search (BFS) technique. BFS is particularly suitable for this task because it explores all vertices at the present depth level before moving on to vertices at the next depth level, allowing us to find the shortest paths efficiently.
Here's an approach to implement this solution: **(the code is in Listing 2 5.1.1), where hallucination happens.**
Graph Representation: Represent the graph using an adjacency list. BFS from Each Node: For each node, perform a BFS to find the shortest cycle starting and ending at that node. Cycle Detection: During the BFS, if we encounter a previously visited node that is not the parent of the current node, we have detected a cycle. Track Minimum Cycle Length: Track the length of the shortest cycle found during all BFS traversals. Edge Cases: If no cycle is found after all BFS traversals, return -1.

```python
class Solution:
  def findShortestCycle(self, n:int, edges:List[List[int]]) -> int:
    adjacency_list = [[] for _ in range(n)]
    for u, v in edges:
    ......
    def bfs(start):
        ......
        while queue:
            node, parent, depth = queue.popleft()
            for neighbor in adjacency_list[node]:
                if neighbor == parent:
                    continue
                if neighbor in visited:
                    # Cycle detected
                    cycle_length = depth + visited[neighbor] + 1
                    if cycle_length >= 3:
                        shortest_cycle = min(shortest_cycle, cycle_length)
                else:
                    visited[neighbor] = depth + 1
                    queue.append((neighbor, node, depth + 1))
    for i in range(n):
        bfs(i)
    ......
TestCase: n=6, edge=[[4,2],[5,1],[5,0],[0,3], \
        [5,2],[1,4],[1,3],[3,4]]
Output: 3 (Correct)
```

Listing 1: Code output generated by the SSD-enhanced GPT-4o.

```python
class Solution:
  def findShortestCycle(self, n:int, edges:List[List[int]]) -> int:
     graph = defaultdict(list)
     ......
     def bfs(start):
        dist = [-1] * n
        dist[start] = 0
        queue = deque([(start, -1)])
        while queue:
            current, parent = queue.popleft()
            # Hallucination Happens!
            for neighbor in graph[current]:
                if dist[neighbor] == -1:
                    dist[neighbor] = dist[current] + 1
                    queue.append((neighbor, current))
                elif neighbor != parent:
                    return dist[current] + dist[neighbor] + 1
        return float('inf')
    shortest_cycle = float('inf')
    for i in range(n):
        shortest_cycle = min(shortest_cycle, bfs(i))
    ......
TestCase: n=6, edge=[[4,2],[5,1],[5,0],[0,3], \
        [5,2],[1,4],[1,3],[3,4]]
Output: 4 (Incorrect)
```

Listing 2: Code output generated by the CoT-based GPT-4o (hallucinated).

Table 2: Passing rate comparison of GPT-3.5, GPT-4, and GPT-4o on graph problems, demonstrating LLM's graph reasoning can be enhanced by SSD.

	GPT-3.5	GPT-4	GPT-4o
Easy-level (3)	100%	100%	100%
Medium-level (5)	80%	100%	100%
Hard-level (9)	0%	33.30%	33.30%
Hard-level (9) + SSD	11.11%	66.67%	66.67%
Performance Gain	**+11.11%**	**+33.37%**	**+33.37%**

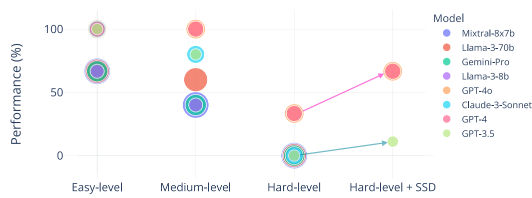


Figure 6: Passing rate comparison across eight LLMs, focusing on GPT-3.5, GPT-4, and GPT-4o to highlight the performance gains from using SSD.

5.2 Results

We conducted experiments using three versions of GPT models (GPT-3.5, GPT-4, and GPT-4o) on a set of hard-level graph problems (the same as Figure 4). Given the near-perfect performance on easy- and medium-level problems, our evaluation focuses primarily on the hard-level problems from GraphEval2000. We compare the performance of SSD method with the CoT-based method. The results are shown in Table 2 and Figure 6. All the settings are the same as in Figure 4. Our method outperforms the CoT-based method on three GPT models, demonstrating the effectiveness of SSD in enhancing LLMs' graph reasoning capabilities. The increase from a 0% passing rate to 11.11% with GPT-3.5, and 33.30% to 66.67% with GPT-4 and GPT-4o. This improvement validates our hypothesis that a structured, decomposed framework with test cases aids LLMs in solving complex graph problems. We notice that the performance of GPT-3.5 is not improved as much as the other two models. This is because GPT-3.5 has a smaller model size and may not have enough knowledge to solve the hard-level graph problems.

6 Conclusion

We introduce GraphEval2000, the first graph dataset that help evaluate the reasoning ability of LLMs through coding challenges. It includes 40 data structure problems with 2000 graph test cases. To better help readers use our dataset, we propose SSD, an instruction-based method, to improve the reasoning ability of LLMs. Our experimental results validate the usefulness and effectiveness of GraphEval2000 and SSD.

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] See Section.
- Did you include the license to the code and datasets? [No] The code and the data are proprietary.
- Did you include the license to the code and datasets? [N/A]

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [N/A]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments (e.g. for benchmarks)...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. Introducing the next generation of claude. *https://www.anthropic.com/ news/claude-3-family/*, 2024.

Zhen Bi, Jing Chen, Yinuo Jiang, Feiyu Xiong, Wei Guo, Huajun Chen, and Ningyu Zhang. Codekgc: Code language model for generative knowledge graph construction. *ACM Transactions on Asian and Low-Resource Language Information Processing*, 23(3):1–16, 2024.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020a.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020b.

Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. Graphllm: Boosting graph reasoning ability of large language model. *arXiv preprint arXiv:2310.05845*, 2023.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*, 2022.

NetworkX Developers. Networkx homepage. *https://networkx.org/*, 2024.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Marius-Constantin Dinu, Claudiu Leoveanu-Condrei, Markus Holzleitner, Werner Zellinger, and Sepp Hochreiter. Symbolicai: A framework for logic-based approaches combining generative models and solvers. *arXiv preprint arXiv:2402.00854*, 2024.

Yijiang River Dong, Lara J Martin, and Chris Callison-Burch. Corrpus: Code-based structured prompting for neurosymbolic story understanding. *arXiv preprint arXiv:2212.10754*, 2022.

Meng Fang, Shilong Deng, Yudi Zhang, Zijing Shi, Ling Chen, Mykola Pechenizkiy, and Jun Wang. Large language models are neurosymbolic reasoners. *arXiv preprint arXiv:2401.09334*, 2024.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560*, 2023.

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. *arXiv preprint arXiv:2305.15066*, 2023.

Amr Hendy, Mohamed Gomaa Abdelrehim, Amr Sharaf, Vikas Raunak, Mohamed Gabr, Hitokazu Matsushita, Young Jin Kim, Mohamed Afify, and Hany Hassan Awadalla. How good are gpt models at machine translation? a comprehensive evaluation. *ArXiv*, abs/2302.09210, 2023. URL https://api.semanticscholar.org/CorpusID:257038384.

Wenpin Hou and Zhicheng Ji. A systematic evaluation of large language models for generating programming code. *arXiv preprint arXiv:2403.00894*, 2024.

Xueyu Hu, Kun Kuang, Jiankai Sun, Hongxia Yang, and Fei Wu. Leveraging print debugging to improve code generation in large language models. *arXiv preprint arXiv:2401.05319*, 2024.

Qian Huang, Hongyu Ren, Peng Chen, Gregor Kržmanc, Daniel Zeng, Percy S Liang, and Jure Leskovec. Prodigy: Enabling in-context learning over graphs. *Advances in Neural Information Processing Systems*, 36, 2024.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. Large language models on graphs: A comprehensive survey. *arXiv preprint arXiv:2312.02783*, 2023.

Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Suhang Wang, Yu Meng, and Jiawei Han. Graph chain-of-thought: Augmenting large language models by reasoning on graphs. *arXiv preprint arXiv:2404.07103*, 2024.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, Benjamin Newman, Binhang Yuan, Bobby Yan, Ce Zhang, Christian Cosgrove, Christopher D. Manning, Christopher R'e, Diana Acosta-Navas, Drew A. Hudson, E. Zelikman, Esin Durmus, Faisal Ladhak, Frieda Rong, Hongyu Ren, Huaxiu Yao, Jue Wang, Keshav Santhanam, Laurel J. Orr, Lucia Zheng, Mert Yuksekgonul, Mirac Suzgun, Nathan S. Kim, Neel Guha, Niladri S. Chatterji, O. Khattab, Peter Henderson, Qian Huang, Ryan Chi, Sang Michael Xie, Shibani Santurkar, Surya Ganguli, Tatsunori Hashimoto, Thomas F. Icard, Tianyi Zhang, Vishrav Chaudhary, William Wang, Xuechen Li, Yifan Mai, Yuhui Zhang, and Yuta Koreeda. Holistic evaluation of language models. *Annals of the New York Academy of Sciences*, 1525:140 – 146, 2023. URL https://api.semanticscholar.org/CorpusID:253553585.

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

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang, editors, *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.90. URL https://aclanthology.org/2022.emnlp-main.90.

Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. Autoformalizing euclidean geometry. *arXiv preprint arXiv:2405.17216*, 2024.

Fred Paas and Jeroen JG van Merriënboer. Cognitive-load theory: Methods to manage working memory load in the learning of complex tasks. *Current Directions in Psychological Science*, 29(4): 394–398, 2020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Oscar J Romero, John Zimmerman, Aaron Steinfeld, and Anthony Tomasic. Synergistic integration of large language models and cognitive architectures for robust ai: An exploratory analysis. In *Proceedings of the AAAI Symposium Series*, volume 2, pages 396–405, 2023.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model with knowledge graph. *arXiv preprint arXiv:2307.07697*, 2023.

Mirac Suzgun, Nathan Scales, Nathanael Scharli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed Huai hsin Chi, Denny Zhou, and Jason Wei. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Annual Meeting of the Association for Computational Linguistics*, 2022. URL https://api.semanticscholar.org/CorpusID:252917648.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

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

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

Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zhu, Shuiguang Deng, et al. Codescope: An execution-based multilingual multitask multidimensional benchmark for evaluating llms on code understanding and generation. *arXiv preprint arXiv:2311.08588*, 2023.

Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. Leandojo: Theorem proving with retrieval-augmented language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *Advances in neural information processing systems*, 32, 2019.

Jiawei Zhang. Graph-toolformer: To empower llms with graph reasoning ability via prompt augmented by chatgpt. *arXiv e-prints*, pages arXiv–2304, 2023.

Li Zhang, Hainiu Xu, Yue Yang, Shuyan Zhou, Weiqiu You, Manni Arora, and Chris Callison-Burch. Causal reasoning of entities and events in procedural texts. In Andreas Vlachos and Isabelle Augenstein, editors, *Findings of the Association for Computational Linguistics: EACL 2023*, pages 415–431, Dubrovnik, Croatia, May 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-eacl.31. URL https://aclanthology.org/2023.findings-eacl.31.

Zeyang Zhang, Xin Wang, Ziwei Zhang, Haoyang Li, Yijian Qin, Simin Wu, and Wenwu Zhu. Llm4dyg: Can large language models solve problems on dynamic graphs? *arXiv preprint arXiv:2310.17110*, 2023b.