GraphEval36K: Benchmarking Coding and Reasoning Capabilities of Large Language Models on Graph Datasets

Qiming Wu*, **Zichen Chen***, **Will Corcoran, Misha Sra, Ambuj Singh**
University of California, Santa Barbara
{qimingwu,zichen_chen,wcorcoran,sra,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 manipulate, program, and reason about structured data, especially graphs. We introduce GraphEval36K[1], the **first** comprehensive graph dataset, comprising 40 graph coding problems and 36,900 test cases to evaluate the ability of LLMs on graph problem-solving. Our dataset is categorized into eight primary and four sub-categories to ensure a thorough evaluation across different types of graphs. We benchmark ten LLMs, finding that private models outperform open-source ones, though the gap is narrowing. We also analyze the performance of LLMs across directed vs undirected graphs, different kinds of graph concepts, and network models. 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 complex graph tasks. Results show that SSD improves the average passing rate of GPT-4, GPT-4o, Gemini-Pro and Claude-3-Sonnet by 8.38%, 6.78%, 29.28% and 25.28%, respectively.

1 Introduction

Large language models (LLMs) such as GPTs (Achiam et al., 2023; Brown et al., 2020a; Chen et al., 2021), Gemini (Team et al., 2023; Reid et al., 2024), Claude-3 (Anthropic, 2024), LLaMA-3 (Touvron et al., 2023), Mixtral (Jiang et al., 2024), DeepSeek-V3 (Liu et al., 2024a) and Qwen-2.5-coder(Hui 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, LLMs have difficulty with complex coding problems, particularly those involving structured data like graphs (Zhang, 2023).

Current research highlights that while LLMs can handle basic graph-related queries, their performance declines on more complex coding challenges involving graph algorithms and multi-step problem solving (Liu and Wu, 2023; Wang et al., 2024; Creswell et al., 2022). These shortcomings highlight the need for targeted evaluation and improvement of LLM's coding abilities in graph-related tasks (Liu et al., 2024b; Cai et al., 2024).

To analyze the above gaps, we propose GraphEval36K, the **first dataset** designed to evaluate the graph-solving capabilities of LLMs through coding problems. GraphEval36K includes 40 graph coding problems with 36,900 test cases, covering a wide range of graph characteristics and algorithmic challenges. Each problem provides: (1) a problem statement, (2) data examples, (3) constraints, and (4) a coding framework for LLMs to build solutions. The dataset is organized into eight primary categories of graph structures: sparse, planar, regular, dense, complete (Diestel, 2024), Small-world (Watts and Strogatz, 1998), Erdos-Renyi (Erdos et al., 1960), and Power-law (Barabási and Albert, 1999) graphs, with sub-categories such as connected, disconnected, cyclic, and acyclic graphs to ensure comprehensive coverage. Based on GraphEval36K, we propose an evaluation framework that is designed to give immediate feedback by returning failed test cases and execution details, promoting deeper model understanding and troubleshooting. This mechanism differentiates our approach from traditional coding platforms (e.g.,

*These authors contributed equally.
[1] GraphEval36K is available at https://grapheval36k.github.io/, under MIT license.

Findings of the Association for Computational Linguistics: NAACL 2025, pages 8110–8132
April 29 - May 4, 2025 ©2025 Association for Computational Linguistics

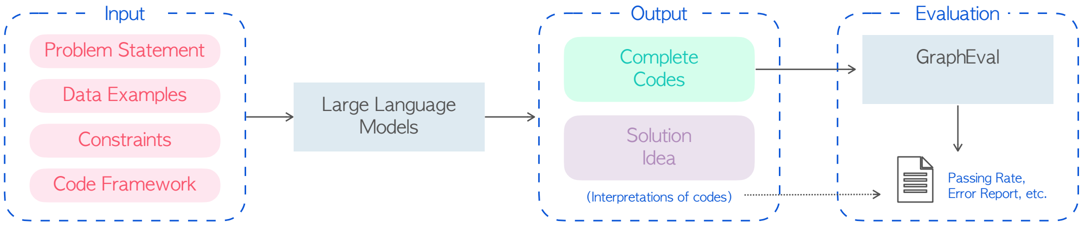


Figure 1: Overview of the Evaluation Framework. For each problem, we input problem statement, data examples, and code framework to LLMs. The LLMs generate the corresponding code and provide explanations. Finally, we evaluate the code on GraphEval36K and return the score details.

LeetCode), where test case details are often hidden from users (Hou and Ji, 2024; Hu et al., 2024).

To further enhance the usability of our evaluation framework and GraphEval36K, we propose an instruction-based method, Structured Symbolic Decomposition (SSD). Inspired by human problem-solving techniques (Paas and van Merriënboer, 2020), SSD decomposes complex tasks into manageable components: a "cognitive step" for understanding the problem and an "action step" for implementing the solution. The experiments show that SSD enhances the average passing rate of GPT-4, GPT-4o, Gemini-Pro and Claude-3 Sonnet by 8.38%, 6.78%, 29.28% and 25.28%, respectively.

Figure 1 presents an overview of our evaluation framework that consists of three steps: problem selection, code generation, and evaluation. Our contributions are summarized as follows:

1. We introduce GraphEval36K, a graph coding and benchmarking dataset with 40 coding problems and 36,900 test cases, designed to evaluate LLMs' graph-solving abilities across diverse and complex graphs.

2. We evaluate the performance of ten LLMs across various graph types (directed vs. undirected, Power-law, Small-world, Erdos-Renyi), fundamental graph concepts (traversal, construction, path finding, topological sorting, cycle detection), and problem difficulty levels (easy, medium, hard).

3. We propose SSD, an instruction-based method that decomposes complex problems into reasoning components for LLMs. Experiments show SSD improves performance by an average of 17.43% across GPT-4, GPT-4o, Gemini-Pro, and Claude-3-Sonnet.

4. We develop an evaluation framework with real-time feedback, paired with SSD to improve LLM performance on complex graph tasks, particularly for models with lower baseline performance, yielding up to a 48.50% improvement.

Our work examines LLM performance on graph problems by categorizing difficulty levels, fundamental concepts, and graph types while evaluating state-of-the-art LLMs. We provide insights into LLM capabilities in graph coding and manipulation, offering a roadmap for future advancements. For LLM users, our work clarifies when and how LLMs can effectively solve graph problems, highlighting their strengths and limitations. Additionally, we introduce SSD to improve LLM performance on complex graph tasks.

2 Related Work

2.1 LLMs on Graph Problem-Solving

Recent research on using LLMs for graph-related tasks follows two approaches: (1) natural language interaction, and (2) code generation.

In the natural language interaction approach, LLMs are provided with a graph and they generate answers based on their understanding of the graph structure. Current research finds that they perform adequately on basic graph problems, however, their performance declines on more complex graph problems (Jin et al., 2023; Wang et al., 2024; Liu and Wu, 2023; Guo et al., 2023). Inspired by the Chain-of-Thought (CoT) method (Wei et al., 2022; Kojima et al., 2022), researchers propose a step-by-step reasoning framework to help LLMs solve graph problems in a more structured way (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., 2023a). Notably, when LLMs are provided with additional example solutions, their performance improves. However, the improvement is marginal (Zhang et al., 2023b; Fatemi et al., 2023; Chai et al., 2023).

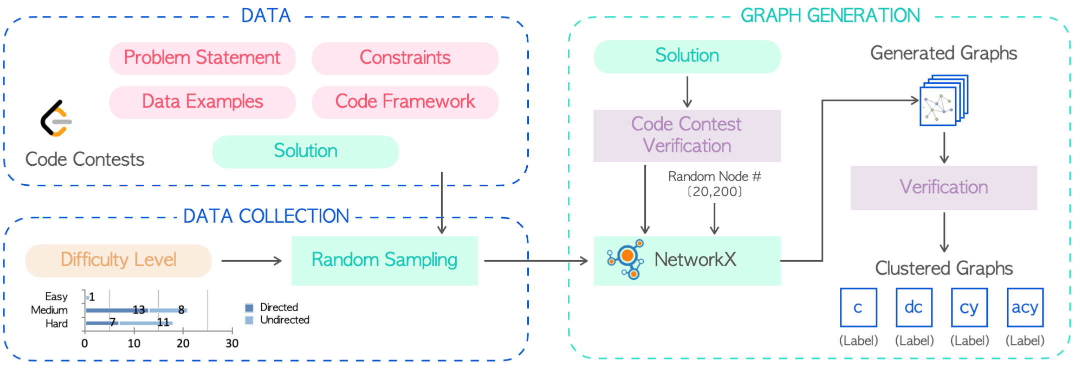


Figure 2: The GraphEval36K dataset is constructed through a pipeline that begins with data collection from code contests (LeetCode). Next, problems are randomly sampled according to their difficulty levels, and corresponding graphs are generated using NetworkX. These graphs are then clustered and labeled based on whether they are connected (c), disconnected (dc), cyclic (cy), or acyclic (acy). Verification steps ensure labeling accuracy, though the exact labels may vary depending on each graph's characteristics.

In the code generation approach, recent research demonstrates that LLMs exhibit significantly improve problems-solving abilities through code generation, instead of relying solely on natural language responses (Suzgun et al., 2022; Liang et al., 2023; Hendy et al., 2023). Researchers transform complex problems into code problems to facilitate effective interaction and enhance the performance of LLMs (Madaan et al., 2022; Zhang et al., 2023a; Bi et al., 2024; Dong et al., 2022; Yan et al., 2023; Cai et al., 2024).

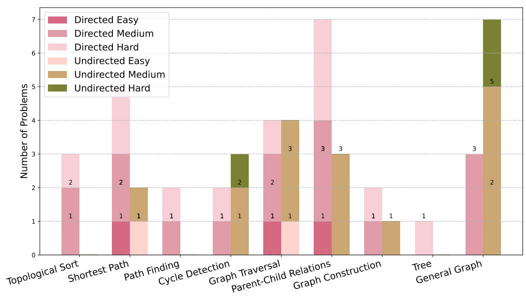


Figure 3: Distribution of graph problems on concepts and difficulty levels.

2.2 LLMs vs. Traditional Graph Machine Learning

Traditional graph machine learning (ML) models, such as Graph Neural Networks (GNNs) (Kipf and Welling, 2016; Velickovic et al., 2017), demonstrate strong performance in tasks like node classification (Xiao et al., 2022), link prediction (Zhang and Chen, 2018), and graph classification (Zhang et al., 2018). However, they are designed for specific tasks with certain architectures (Garg et al., 2020). In contrast, LLMs have the potential to generalize across different graph-related tasks, without requiring specific tuning (Jin et al., 2024a; Sun et al., 2023b). In this work, we aim to evaluate the ability of LLMs to solve graph problems by generating correct code.

2.3 Comparison with Existing Datasets

Table 1 presents a comparative comparison of GraphEval36K with existing datasets designed to evaluate LLMs in graph-related tasks. Unlike previous datasets, which focus primarily on specific aspects of graph reasoning, such as algorithmic problem solving (CLRS-30 (Veličković et al., 2022)), logical inference (ProofWriter (Tafjord et al., 2021), PrOntoQA (Misra et al., 2023)) or hybrid graph analysis (HGB (Li et al., 2023)), GraphEval36K offers a **comprehensive evaluation framework** that includes both graph coding and reasoning. Furthermore, while datasets such as BIG-Bench (Srivastava et al., 2022) and GraCoRe (Yuan et al., 2024) include graph-related reasoning tasks, they are often constrained to small-scale graphs or heterogeneous structures, limiting their applicability to broader graph problem-solving.

3 Dataset Construction

An overview of the dataset construction pipeline is illustrated in Figure 2. The two major steps consist of (a) selection of graph problems with varying degrees of user-defined difficulty levels and graph concepts, (b) generation of graphs of different types.

Dataset Name	Graph Problems	Reasoning Scope	Graph Types	Scale	LLM Benchmark
CLRS-30 (Veličković et al., 2022)	Algorithmic	Limited	Planar, Trees	30 Problems	No
BIG-Bench (Srivastava et al., 2022)	Logic Puzzles	Indirect	Small Graphs	204 Tasks	Yes
ProofWriter (Tafjord et al., 2021)	Logical Reasoning	Limited	Implicit graphs	16k Examples	Yes
PrOntoQA (Misra et al., 2023)	Ontology Reasoning	Specific	Taxonomies	80k QA pairs	Yes
GraCoRe (Yuan et al., 2024)	Graph Reasoning	Pure & Heterogeneous Graphs	—	5k Graphs	Yes
HGB (Li et al., 2023)	Hybrid Graph Analysis	Complex Structures	Biology, Social	23 Datasets	No
`GraphEval36K` (**Ours**)	Graph Coding and Reasoning	Comprehensive	8 Main Categories with 4 Sub-categories	40 Problems with 36.9k Cases	Yes

Table 1: Comparison of `GraphEval36K` with existing datasets for evaluating LLMs on graph-related tasks.

3.1 Problem Collection

We collect a total of 40 graph data structure problems from the LeetCode[2], 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 distribution of the problems across difficulty levels and graph concepts is presented in Figure 3. For each problem, we collect: problem statement, input/output examples, data constraints, and code framework.

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 eight main categories: Sparse Graph, Planar Graph, Regular Graph, Dense Graph, Complete Graph (Diestel, 2024), Small-world Graph (Watts and Strogatz, 1998), Erdos-Renyi Graph (Erdos et al., 1960) and Power-law Graph (Barabási and Albert, 1999). Below, we provide definitions of these graphs and how they are generated.

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 (Bollobás and Riordan, 2011):

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

Planar Graph A planar graph can be drawn without edge intersections, except at vertices (Schnyder, 1989). For a finite, connected planar graph with F faces following Euler's formula, we have:

$$|V| - |E| + |F| = 2. \quad (2)$$

Regular Graph A graph G is k-regular if every vertex has the same degree k (Stanić, 2017).

[2] https://leetcode.com/tag/graph/

Dense Graph For an undirected graph, the density D is given by $D_{\text{undirected}} = \frac{2|E|}{|V|(|V|-1)}$ (Lee and Streinu, 2008). For a directed graph, the density D is $D_{\text{directed}} = \frac{|E|}{|V|(|V|-1)}$. In our experiment, we fix $D = 0.7$.

Complete Graph A graph G is complete if there is an edge between every pair of distinct vertices (Pirnot, 2001).

Small-World Graphs Small-world graphs, generated via the Watts-Strogatz model (Watts and Strogatz, 1998), use parameters n (nodes), k (nearest neighbors), and p (rewiring probability), where k is randomly chosen between 2 and $\frac{n}{2}$, and p between 0.1 and 0.3. Starting with a ring lattice where each node i connects to its k nearest neighbors, each edge (u, v) is rewired with probability p to a random node. The resulting graph has a logarithmic path length $L(G)$ and a high clustering coefficient $C(G)$.

Erdos-Renyi Graphs Erdos-Renyi graphs (Erdos et al., 1960) follow the $G(n, p)$ model, where n is the number of nodes, and p is the probability of forming an edge between any two nodes. Each edge (u, v), where $u \neq v$, is included in E with probability p.

Power-Law Graphs Power-law graphs are generated using the Barabasi-Albert (BA) model with parameters n (nodes) and m (edges per new node) (Barabási and Albert, 1999), where m is randomly chosen between 1 and 10. New nodes connect to m existing nodes via preferential attachment. This process yields a scale-free network with a Power-law degree distribution:

$$P(k) \sim k^{-\gamma}.$$

Graph Generation We use the "NetworkX" (Developers, 2024) to generate graph samples. By leveraging it, we construct both directed and undirected graphs according to our classification results

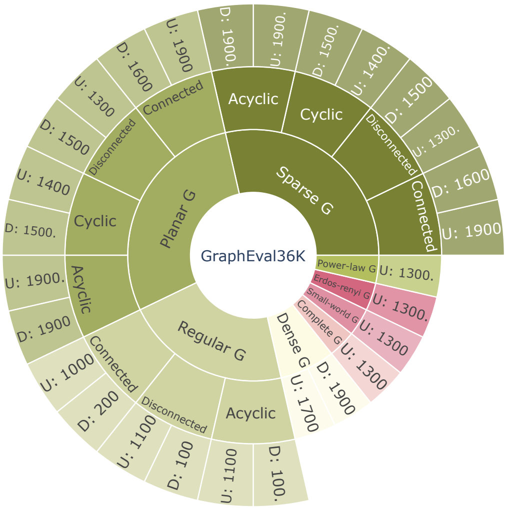


Figure 4: Structure of `GraphEval36K`. "U" denotes undirected graphs, "D" denotes directed graphs, with numbers indicating the count of cases in each category. The graphs are classified into eight main categories: sparse, planar, regular, dense, complete, Small-world, Erdos-Renyi, and Power-law. Some are further divided into four sub-categories: connected, disconnected, cyclic, and acyclic. Sub-categories may vary based on the characteristics of the main categories. Detailed dataset analysis is shown in Appendix A.

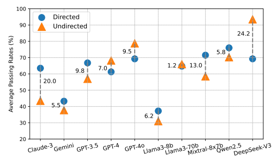
(as shown in Figure 4). We generate 100 graph samples for each sub-category, where the number of vertices in each graph is randomly chosen between 20 and 200. This enables us to create the dataset with varying levels of complexity.

3.3 LLMs under Consideration

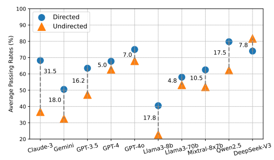
We evaluate ten LLMs in this study: Claude-3-Sonnet (Anthropic, 2024), Gemini-Pro (Reid et al., 2024), GPT-3.5 (Brown et al., 2020b), GPT-4, GPT-4o (Achiam et al., 2023), Llama-3-8b, Llama-3-70b (Dubey et al., 2024), Mixtral-8x7b (Jiang et al., 2024), Qwen2.5-Coder-32B (Hui et al., 2024) and DeepSeek-V3 (Liu et al., 2024a). The first five models are private, while the latter five are open-source. These models are selected for their diversity in scale, architecture, and relevance within the research community. Our objective is to assess their graph-solving capabilities and compare their performance on the `GraphEval36K` dataset.

4 Evaluation of LLMs on Graph Problems

In this section, we evaluate the LLMs' capability in solving graph problems. We first discuss the broad results of evaluating LLMs on graph problems (Table 2), and then perform three analytical studies. First, we evaluate how well LLMs understand core graph theory concepts in their problem-solving strategies (Figure 6). Second, we analyze the impact of directed vs. undirected graphs on LLM coding performance (Table 2 and Figure 5). Finally, we evaluate model generalization to complex graphs, including Small-world, Erdos-Renyi, and Power-law graphs (Figure 7).

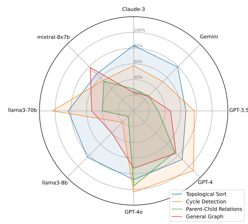


(a) Sparse Graphs

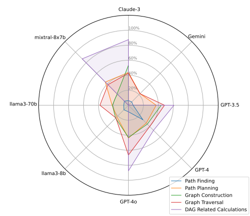


(b) Planar Graphs

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.



(a) Easy Concepts



(b) Difficult Concepts

Figure 6: Passing rate of LLMs on graph concepts. Part (a) presents the passing rate on easy concepts, and part (b) presents the passing rate on difficult concepts.

In Table 2, we observe that private models consistently outperform open-source models in all graph categories, especially GPT-4 and GPT-4o. The performance gap is notable in complex graph types such as dense and complete graphs, where private models achieve passing rates close to 100%, while open-source models like Mixtral-8x7b perform in a lower range of 70-80%. However, Qwen2.5-Coder-32B and DeepSeek-V3 have narrowed the performance gap. They perform better than private models on sparse and planar graphs.

Results on Graph Problem Concepts We classify graph problems into nine concepts: Topological Sort, Cycle Detection, Parent-Child Relations, General Graph, Path Finding, Path Planning, Graph Construction, Graph Traversal and DAG-Related Calculations. As shown in Figure 6, LLMs perform better on the concepts in Figure 6(b), and face more difficulty with those in Figure 6(a). For exam-

		Claude-3 -Sonnet	Gemini -Pro	GPT -3.5	GPT -4	GPT -4o	Llama-3 -8b	Llama-3 -70b	Mixtral -8x7b	Qwen2.5 -Coder-32B	DeepSeek -V3
SG	c	68 ǀ 39	55 ǀ 41	69 ǀ 48	61 ǀ 66	64 ǀ 77	46 ǀ 27	59 ǀ 70	81 ǀ 57	74 ǀ 72	61 ǀ **92**
	dc	54 ǀ 37	31 ǀ 40	59 ǀ 61	66 ǀ 68	**76** ǀ 81	25 ǀ 32	51 ǀ 62	64 ǀ 60	73 ǀ 66	70 ǀ **96**
	cy	69 ǀ 48	46 ǀ 32	64 ǀ 71	50 ǀ 80	59 ǀ 88	42 ǀ 40	67 ǀ 74	**73** ǀ 64	**73** ǀ 79	67 ǀ **97**
	acy	63 ǀ 50	41 ǀ 38	75 ǀ 48	68 ǀ 59	78 ǀ 69	36 ǀ 25	51 ǀ 58	68 ǀ 53	**84** ǀ 64	79 ǀ **89**
PG	c	**80** ǀ 30	47 ǀ 26	63 ǀ 41	60 ǀ 55	60 ǀ 65	45 ǀ 21	75 ǀ 60	60 ǀ 48	75 ǀ 56	75 ǀ **75**
	dc	64 ǀ 39	39 ǀ 37	61 ǀ 47	80 ǀ 70	**88** ǀ 69	29 ǀ 20	52 ǀ 44	65 ǀ 56	86 ǀ 62	80 ǀ **89**
	cy	**68** ǀ 38	46 ǀ 28	61 ǀ 59	58 ǀ 71	67 ǀ 73	37 ǀ 30	64 ǀ 62	64 ǀ 53	**68** ǀ 70	61 ǀ **84**
	acy	61 ǀ 40	42 ǀ 39	69 ǀ 42	73 ǀ 55	85 ǀ 65	36 ǀ 20	41 ǀ 47	61 ǀ 51	**90** ǀ 61	80 ǀ **79**
RG	c	NA ǀ 65	NA ǀ 39	NA ǀ 68	NA ǀ 85	NA ǀ **97**	NA ǀ 53	NA ǀ 81	NA ǀ 75	NA ǀ 89	NA ǀ 95
	dc	NA ǀ 61	NA ǀ 38	NA ǀ 58	NA ǀ 86	NA ǀ **94**	NA ǀ 42	NA ǀ 79	NA ǀ 73	NA ǀ 77	NA ǀ 93
	cy	NA ǀ 68	NA ǀ 50	NA ǀ 74	NA ǀ 87	NA ǀ **99**	NA ǀ 56	NA ǀ 84	NA ǀ 77	NA ǀ 90	NA ǀ 97
DG	c	38 ǀ 60	32 ǀ 36	37 ǀ 60	48 ǀ 75	48 ǀ 81	**49** ǀ 27	37 ǀ 70	39 ǀ 46	27 ǀ 65	20 ǀ **82**
CG	c	NA ǀ 33	NA ǀ 38	NA ǀ 67	NA ǀ **86**	NA ǀ 64	NA ǀ 43	NA ǀ 50	NA ǀ 57	NA ǀ 67	NA ǀ 68

Table 2: Evaluation Results on `GraphEval36K`. Passing rates (%) of ten LLMs across graph categories. The first column categorizes graphs: "SG" (sparse), "PG" (planar), "RG" (regular), "DG" (dense), and "CG" (complete). Abbreviations include "c" (connected), "dc" (disconnected), "cy" (cyclic), and "acy" (acyclic). Results are shown as "Directed ǀ Undirected" for each category, with "NA" indicating not applicable. Bold values highlight the highest passing rate per row.

ple, LLMs consistently show strong performance on Topological Sort, with Claude-3-sonnet, GPT-4, and GPT-4o achieving rates above 80%. LLMs generally excel at structured problems involving clear hierarchical relationships. However, Claude-3-sonnet and Gemini-pro perform poorly on Path Finding, with passing rates of 6.25% and 21.94%, respectively. In contrast, GPT-4 and GPT-4o show a stronger result on DAG-related Calculations, with GPT-4o achieving 87.50%. The results highlight the strengths and weaknesses of LLMs across different graph concepts, emphasizing the need for further research to improve performance on more complex graph problems.

Results on Directed and Undirected Graphs Most LLMs perform better on directed graphs, but it is hard to draw firm conclusions due to potential differences in problem complexity between directed and undirected graphs. As shown in Table 2 and Figure 5, the performance gap between private and open-source models is more notable on directed graphs, indicating strong capabilities of private LLMs in handling directed graph complexities. Furthermore, private models show a larger performance gap between directed and undirected graphs compared to open-sourced models. However, an interesting observation is that DeepSeek-V3 demonstrates superior performance on undirected graphs compared to directed graph cases. Notably, DeepSeek-V3 outperforms all evaluated LLMs on undirected sparse and planar graph cases,

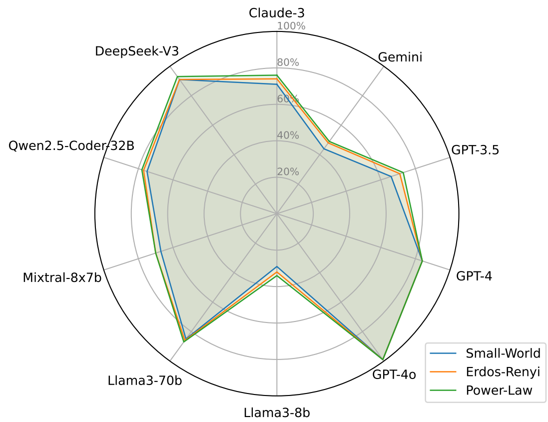


Figure 7: Evaluation results of LLM-generated code on complex graphs: Small-world, Erdos–Renyi, and Power-law.

highlighting its strength in these specific graph structures. Further analyses can be found in Appendix B and C.

Results on Complex Graphs The evaluation results of LLMs across three practical graph types, Small-world, Erdos–Renyi, and Power-law, are summarized in Figure 7. GPT-4o demonstrates the highest performance across all graph types, maintaining a passing rate of 99%. DeepSeek-V3 follows closely with a passing rate of 93%. Additionally, GPT-4, Qwen2.5-Coder-32B, and Llama-3-70B exhibit strong performance, with passing rates ranging from 84% to 87%, demonstrating their robust capability in solving graph-related problems. In contrast, Gemini-pro and Llama-3-8b show the lowest passing rates across all graph types, particularly for Small-world graphs where it achieves only 29%. Claude-3-sonnet, GPT-3.5, and Mixtral-8x7b occupy a middle ground, showing moderately high but varied performance across different graph types, with rates ranging from 66% to 76%. The results indicate that while models like GPT-4o and GPT-4 excel at generating code to solve practical graph problems, performance is variable across different LLMs. This highlights the need for further fine-tuning or architectural improvements to enhance the code generation capabilities of weaker models, particularly in complex graph scenarios.

5 Improving LLM Graph Solving

To enhance the usability of our evaluation framework and `GraphEval36K`, we introduce Structured Symbolic Decomposition (`SSD`), an instruction-based method utilizing test cases from `GraphEval36K` for graph problems. Our approach aims to enable LLMs to perform better graph problem-solving processes.

Methodology We hypothesize that decomposing complex graph problems into smaller, more manageable sub-problems and turning them into symbolic forms (Dinu et al., 2024; Fang et al., 2024; Yang et al., 2024) will enhance the graph-solving capabilities of LLMs. Current methods rely on implicit knowledge and lack explicit guidance (Wei et al., 2022; Jin et al., 2024b; 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 `GraphEval36K` to be evaluation problems. The test cases are used for problem understanding and program testing.

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

- **Problem Clarification:**
  Cognitive Step: *You must first understand and clearly articulate the problem, including all inputs and desired outputs.*
  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:**
  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.*
  Action Step: *Outline the sequential steps required to solve the overall problem.*

- **Solution Formulation:**
  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.*
  Action Step: *Detail the algorithms and approaches for each sub-problem, ensuring they are logically connected and comprehensive.*

- **Program Implementation:**
  Cognitive Step: *Conceptualize the implementation of each solution component.*
  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 method with CoT-based graph solving in the following example. The demonstration involves a "shortest path" question and test cases from `GraphEval36K`:

Problem Statement: *Let $G = (V, E)$ be a bi-directional graph with n vertices, labeled from 0 to $n - 1$. The edges in the graph are given as a 2D integer array edges, where each edges[i] = $[u_i, v_i]$ represents an edge between vertex u_i and vertex v_i. There is no self-loop, and at most one edge exists between any two vertices. 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 vertex, using each edge only once.*
Test Cases:
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]]}*
Number of vertices (n): *{'n': 5}, {'n': 6}, {'n': 4}, {'n': 4}, {'n': 6}*
Output: *{'label': 3}, {'label': 3}, {'label': 3}, {'label': -1}, {'label': 3}*

```
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:
```

	GPT-4		GPT-4o		Gemini-Pro		Claude-3-Sonnet	
	Vanilla	**+SSD**	**Vanilla**	**+SSD**	**Vanilla**	**+SSD**	**Vanilla**	**+SSD**
Path Finding	80.25	88.83	82.15	88.76	0	0.37	85.24	95.56
Path Planning	100.00	100.00	100.00	100.00	36.78	66.67	68.96	100.00
Graph Construction	16.13	19.29	22.32	25.49	12.97	19.04	31.44	36.75
DAG Related	85.50	100.00	90.35	100.00	0	2.44	54.88	84.15
Directed	51.25	64.92	91.67	92.31	15.54	40.71	66.67	75.00
Undirected	89.87	92.00	84.35	92.00	25.61	64.00	52.91	83.20
Small-World	100.00	100.00	100.00	100.00	0	68.50	50.00	98.50
Erdos-Renyi	98.00	100.00	100.00	100.00	33.33	79.33	64.67	95.67
Power-Law	100.00	100.00	100.00	100.00	33.33	80.00	66.67	100.00

Table 3: Passing rates of four LLMs (GPT-4, GPT-4o, Gemini-Pro, and Claude-3-Sonnet) across different graph problems and types of graph samples. We randomly sampled six problems, a total of 6,000 graph samples from our dataset, to compare model performance with and without (vanilla) the proposed SSD method.

```
                    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)
```

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

```
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)
```

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

We use GPT-4o as the demo model. The generated code details are shown in Code 1 and 2. More reasoning details and examples can be found in the Appendix F, G and H. We observe that the SSD-enhanced method GPT-4o reduces hallucinations and improves graph problem-solving capabilities. In the following, we further evaluate SSD on various graph problems selected from GraphEval36K.

The results are shown in Table 3, which compares the performance of GPT-4, GPT-4o, Gemini-pro, and Claude-3-sonnet. More results can be found in Appendix E.

Results on Graph Problem Concepts SSD leads to significant improvements across most problem types. For Path Finding, GPT-4 improves from 80.25% to 88.83%, GPT-4o from 82.15% to 88.76%, Claude-3-sonnet from 85.24% to 95.56%, while Gemini-pro shows a slight improvement from 0% to 0.37%. In Path Planning, GPT-4 and GPT-4o maintain 100% with or without SSD, while Gemini-pro and Claude-3-sonnet have a notable improvement from 36.78% to 66.67%, and 68.96% to 100%, respectively. For Graph Construction, GPT-4 improves from 16.13% to 19.29%, GPT-4o from 22.32% to 25.49%, Gemini-pro from 12.97% to 19.04%, and Claude-3-sonnet from 31.44% to 36.75%. In DAG-Related Calculations, GPT-4 and GPT-4o achieve 100% with SSD, compared to 85.50% and 90.35%, respectively, Claude-3-sonnet improves from 54.88% to 84.15%, while Gemini-pro from 0% to 2.44%. The results show that SSD improves performance across all models, highlighting our effectiveness in enhancing LLMs with varying capacities. GPT-4 and GPT-4o, already strong in tasks, reach near-perfect performance, while Claude-3-sonnet benefits greatly in more complex problems. Gemini-pro shows substantial gains, due to its original lower performance.

Results on Directed and Undirected Graphs In directed graphs, GPT-4 improves from 51.25% to 64.92%, and GPT-4o sees a slight increase from 91.67% to 92.31%. Gemini-pro shows a clear jump from 15.54% to 40.71%, and Claude-3-sonnet improves from 66.67% to 75.00%. In undirected graphs, the models generally perform better, with GPT-4 improving from 89.87% to 92.00% and GPT-4o from 84.35% to 92.00%. Gemini-pro shows a notable improvement from 25.61% to 64.00%, and Claude-3-sonnet from 52.91% to 83.20%. The results demonstrate that SSD effectively enhances model performance on both directed and undirected graphs, particularly for those with lower baseline performance.

Results on Complex Graphs SSD leads to significant improvements across all models for complex graph types. For Small-World graphs, GPT-4 and GPT-4o maintain 100%, while Gemini-pro improves from 0% to 68.50%, and Claude-3-sonnet from 50.00% to 98.50%. In Erdos-Renyi graphs, Gemini-pro jumps from 33.33% to 79.33%, and Claude-3-sonnet from 64.67% to 95.67%, with GPT-4 and GPT-4o both reaching 100%. Similarly, in Power-Law graphs, Gemini-pro improves from 33.33% to 80.00%, and Claude-3-sonnet from 66.67% to 100%. The results show that SSD significantly enhances models like Gemini-pro and Claude-3-sonnet, which initially perform poorly on complex graphs. SSD enables them to handle diverse structures more effectively.

6 Conclusion

In this work, we introduce GraphEval36K, the first graph dataset designed to benchmark LLMs' graph reasoning abilities through coding challenges. It includes 40 graph problems and 36,900 graph test cases, covering a range of difficulty levels and graph concepts. To enhance LLM performance, we propose SSD, an instruction-based approach aimed at improving reasoning capabilities on graph-related problems. Our experiments demonstrate the effectiveness of GraphEval36K and SSD in analyzing LLM performance across different graph types and concepts. Our findings highlight potential areas for improvement and identify the graph coding tasks where LLMs perform well and where they face challenges.

Limitation

Despite the strengths of our proposed dataset GraphEval36K, there are certain limitations to consider. The dataset includes 40 coding problems and 36,900 graph samples, which, while comprehensive, is smaller in size compared to other LLM evaluation datasets. Expanding the dataset could offer broader coverage and further insights into model performance across a wider array of graph problems. Additionally, while we evaluate ten prominent LLMs (GPT-3.5, GPT-4, GPT-4o, Claude-3 Sonnet, Gemini-Pro, llama3-8b, llama3-70b, Mixtral-8x7b, Qwen2.5-Coder-32B and DeepSeek-V3), we were unable to include all available models due to the fast-evolving nature of LLM development. This rapid progression in the field may mean that newer models could offer improved performance or demonstrate different behavior on the dataset. However, our dataset is designed to be flexible and general enough to accommodate future LLMs, enabling the research community to use it for evaluating a wide range of models, regardless of advancements in LLM development. Future work can also focus on expanding the dataset size and further refining its categories to address evolving needs in LLM evaluation.

Ethical Considerations

In conducting this study, we do not foresee any ethical concerns. Our dataset consists solely of synthetic graph coding problems and generated test cases, ensuring no involvement of personal data or human participants. The development of GraphEval36K and our evaluation of LLMs aimed at improving the understanding of graph problem-solving capabilities in large language models. Furthermore, all data used is anonymous, and the research is compliant with ethical guidelines for responsible AI research. No identifiable information, real-world consequences, or human subjects are involved in this work, thereby minimizing ethical risks.

Acknowledgment

This material is based upon work supported by the National Science Foundation under grant no. 2229876 and is supported in part by funds provided by the National Science Foundation, by the Department of Homeland Security, and by IBM. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or its federal agency and industry partners.

References

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

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

Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *science*, 286(5439):509–512.

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

Béla Bollobás and Oliver Riordan. 2011. Sparse graphs: metrics and random models. *Random Structures & Algorithms*, 39(1):1–38.

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

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

Qiaolong Cai, Zhaowei Wang, Shizhe Diao, James Kwok, and Yangqiu Song. 2024. Codegraph: Enhancing graph reasoning of llms with code. *arXiv preprint arXiv:2408.13863*.

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

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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

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

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

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

Reinhard Diestel. 2024. *Graph theory*. Springer (print edition); Reinhard Diestel (eBooks).

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

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

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Paul Erdos, Alfréd Rényi, et al. 1960. On the evolution of random graphs. *Publ. math. inst. hung. acad. sci*, 5(1):17–60.

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

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

Vikas Garg, Stefanie Jegelka, and Tommi Jaakkola. 2020. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning*, pages 3419–3430. PMLR.

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. *arXiv preprint arXiv:2305.15066*.

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

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

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

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

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

Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. 2024a. Large language models on graphs: A comprehensive survey. *IEEE Transactions on Knowledge and Data Engineering*.

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

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

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

Audrey Lee and Ileana Streinu. 2008. Pebble game algorithms and sparse graphs. *Discrete Mathematics*, 308(8):1425–1437.

Zehui Li, Xiangyu Zhao, Mingzhu Shen, Guy-Bart Stan, Pietro Liò, and Yiren Zhao. 2023. Hybrid graph: A unified graph representation with datasets and benchmarks for complex graphs. *arXiv preprint arXiv:2306.05108*.

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. 2023. Holistic evaluation of language models. *Annals of the New York Academy of Sciences*, 1525:140 – 146.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

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

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Wenmeng Zhou, Fei Wang, and Michael Shieh. 2024b. Codexgraph: Bridging large language models and code repositories via code graph databases. *arXiv preprint arXiv:2408.03910*.

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

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 1384–1403, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Dipendra Misra, Christopher Hayes, John Kummerfeld, and Yoav Artzi. 2023. Prontoqa: A challenge dataset for ontology reasoning. *arXiv preprint arXiv:2305.12515*.

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

T.L. Pirnot. 2001. *Mathematics All Around*. Addison Wesley.

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

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

Oscar J Romero, John Zimmerman, Aaron Steinfeld, and Anthony Tomasic. 2023. 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.

Walter Schnyder. 1989. Planar graphs and poset dimension. *Order*, 5:323–343.

Abhijit Srivastava, Nathan Major, Jasmine D. Hernandez, and et al. 2022. Beyond the imitation game: Quantifying and extrapolating llm capabilities with big-bench. *arXiv preprint arXiv:2206.04615*.

Zoran Stanić. 2017. *Regular graphs: a spectral approach*, volume 4. Walter de Gruyter GmbH & Co KG.

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

Xiangguo Sun, Jiawen Zhang, Xixi Wu, Hong Cheng, Yun Xiong, and Jia Li. 2023b. Graph prompt learning: A comprehensive survey and beyond. *arXiv preprint arXiv:2311.16534*.

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. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Annual Meeting of the Association for Computational Linguistics*.

Alexander Tafjord, Peter Clark, and Oyvind Tafjord. 2021. Proofwriter: Generating implications, proofs, and abductive statements over natural language. *arXiv preprint arXiv:2012.13048*.

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

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

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. 2017. Graph attention networks. *stat*, 1050(20):10–48550.

Petar Veličković, Alvaro Sanchez-Gonzalez, Charles Blundell, David Pfau, David G. T. Barrett, and Peter Battaglia. 2022. CLRS: a benchmark for algorithmic reasoning. *arXiv preprint arXiv:2205.15626*.

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

Duncan J Watts and Steven H Strogatz. 1998. Collective dynamics of 'small-world'networks. *nature*, 393(6684):440–442.

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

Shunxin Xiao, Shiping Wang, Yuanfei Dai, and Wenzhong Guo. 2022. Graph neural networks in node classification: survey and evaluation. *Machine Vision and Applications*, 33(1):4.

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

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

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

Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. 2024. Gracore: Benchmarking graph comprehension and complex reasoning in large language models. *arXiv preprint arXiv:2407.02936*.

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

Li Zhang, Hainiu Xu, Yue Yang, Shuyan Zhou, Weiqiu You, Manni Arora, and Chris Callison-Burch. 2023a. Causal reasoning of entities and events in procedural texts. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 415–431, Dubrovnik, Croatia. Association for Computational Linguistics.

Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An end-to-end deep learning architecture for graph classification. *Proceedings of the AAAI conference on artificial intelligence*, 32(1).

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

A Analysis of GraphEval36K

In this section, we analyze our dataset from several perspectives: data distributions, the complexity of graph test cases, and the time and memory usage during evaluations on GraphEval36K.

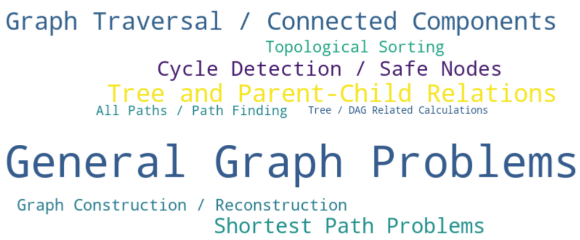


Figure 8: Word cloud of the distribution of data structure problems on concepts.

A.1 Data Distributions

We begin by enumerating the graph test cases. There are 13,400 samples in the directed graph category and 23,500 in the undirected graph category. Therefore, GraphEval36Khas 36,900 graph samples in total. Additionally, we examine the distribution of problems across different difficulty levels and concepts, as illustrated in Figure 3. Our dataset, GraphEval36K, includes eight main graph categories (as shown in Figure 9 and Figure 10, sparse, planar, regular, dense, complete, Small-world, erdos-renyi and Power-law) and four subcategories (connected, disconnected, cyclic, and acyclic) within each main category. For each subcategory, we generated ten graph test cases. Due to the specific characteristics of the main categories, our final dataset comprises **13,400** graph samples for directed graphs and **23,500** for undirected graphs. These samples are designed to evaluate and improve the graph reasoning abilities of LLMs. Additionally, we generate the word cloud (Figure 8) of our dataset by different graph problem concepts.

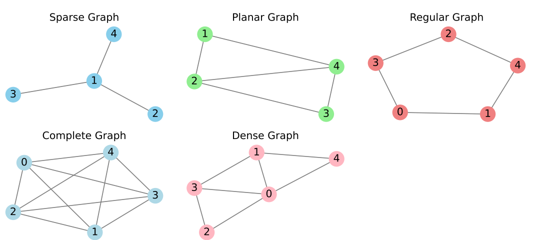


Figure 9: Graph Samples. We use NetworkX (https://networkx.org/) to plot the graph samples of eight main categories in our dataset.

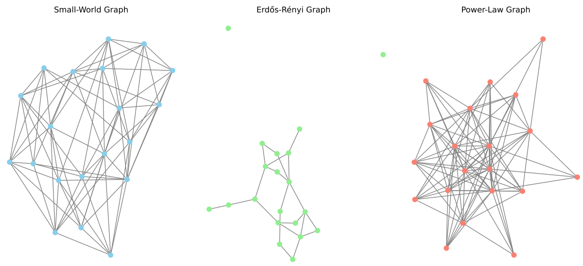


Figure 10: Graph Samples of Small-world, Erdos-Renyi and Power-law graphs. We use NetWorkX to plot these graph samples with the number of nodes to be 20.

The graph problems were collected from LeetCode (https://leetcode.com/tag/graph/), and we calculated the distribution of problems across different difficulty levels[3]. Given the near-perfect performance of LLMs on easy-level data structure problems, we focused primarily on medium- and hard-level problems.

A.2 Dataset Statistics

To analyze the complexity of the graph test cases in GraphEval36K, we include several key metrics: the average number of nodes and edges, the average degree, and the average degree variance. These metrics provide a comprehensive understanding of the structural characteristics of the graphs in our dataset, allowing for a more detailed assessment of the complexity and diversity of the test cases. The results are summarized in Table 4.

To begin with, we first introduce the evaluation metrics. Consider N_i to be number of nodes in the i-th graph, E_i represents the number of edges in the i-th graph. Then, average number of nodes $\overline{N}$ across m graph test cases is

$$\overline{N} = \frac{1}{m} \sum_{i=1}^{m} N_i, \tag{3}$$

and average number of edges $\overline{E}$ across m graph test cases is

$$\overline{E} = \frac{1}{m} \sum_{i=1}^{m} E_i. \tag{4}$$

The degree of a node in a graph is the number of edges connected to it. The average degree of a graph is the mean degree of all its nodes. For an undirected graph with N nodes and E edges, the average degree $\overline{d}$ is

$$\overline{d} = \frac{2E}{N}. \tag{5}$$

[3]The difficulty level is defined by LeetCode.

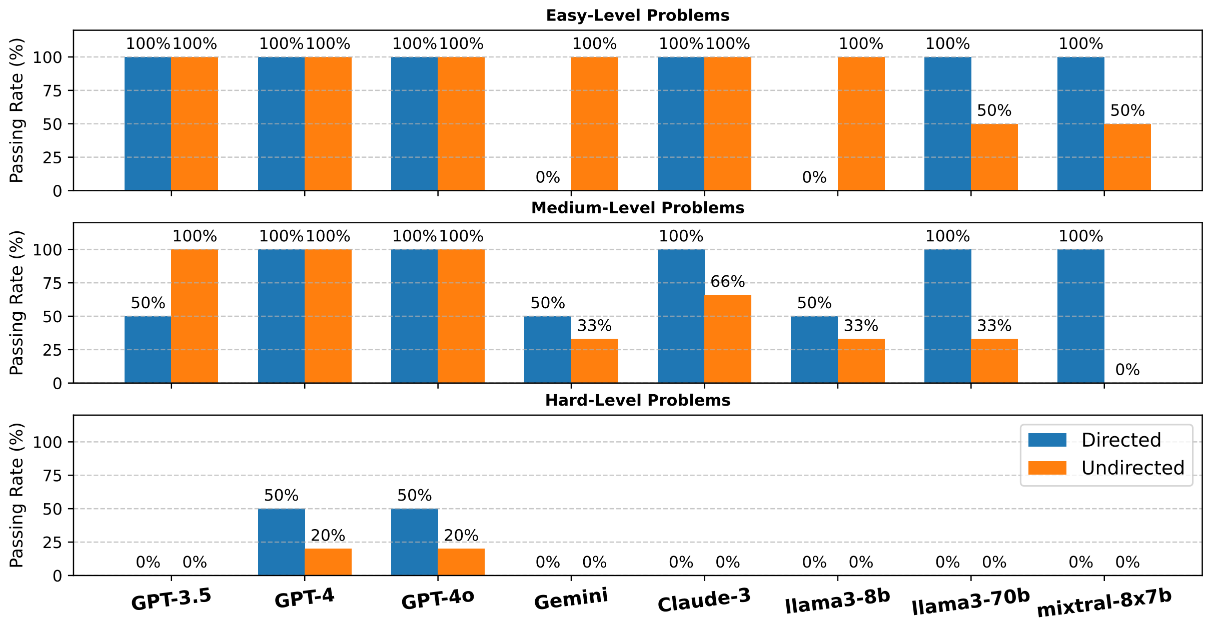


Figure 11: Evaluation results of LLMs on the LeetCode platform. This figure shows the passing rates of LLMs on selected graph data structure problems, categorized into 3 easy-level problems, 5 medium-level problems, and 9 hard-level problems.

Graph Type	**Average Number of Nodes**	**Average Number of Edges**	**Average Degree**	**Average Degree Variance**
Directed Sparse	103.97	268.21	4.79	3.72
Directed Planar	41.09	43.64	2.28	1.18
Undirected Sparse	88.51	89.16	1.95	0.72
Undirected Planar	70.52	71.89	1.98	0.74
Undirected Regular	82.96	201.16	5.01	0.64
Undirected Complete	73.04	4019.89	72.05	0.0
Small-world	101.22	1660.99	26.05	4.33
Erods-Renyi	103.43	824.91	12.74	10.65
Power-law	103.17	543.54	10.18	46.09

Table 4: Complexity Analysis of Graphs. We include the average number of nodes and edges, average degree, and average degree variance to analyze the graph test cases in `GraphEval36K`.

For a directed graph, the average degree is

$$\overline{d} = \frac{E}{N}. \tag{6}$$

Across multiple graphs, the average degree is the mean of the average degrees of each graph. The degree variance measures the variability of the degrees of the nodes in a graph. For a graph with N nodes and degrees $d_1, d_2, \ldots, d_N$, the variance of the degrees is calculated as:

$$\text{Var}(d) = \frac{1}{N}\sum_{i=1}^{N}(d_i - \overline{d})^2 \tag{7}$$

Here, $\overline{d}$ is the average degree of the nodes in that graph. Across multiple graphs, the average degree variance is the mean of the degree variances of each graph.

A.3 Summary

In our dataset, the directed graph has two categories: sparse and planar. The undirected graph has five categories: sparse, planar, regular, dense and complete. We summarized the experimental results based on these categories:

- Directed sparse graphs have a moderate number of nodes and edges. The average degree is relatively low, indicating sparsity. The degree variance suggests variability in node connectivity.
- Directed planar graphs have fewer nodes and

edges compared to sparse graphs. The average degree and variance are low, indicating even sparser connectivity with less variation in node degree.

- Undirected sparse graphs have a similar number of nodes to directed sparse graphs but significantly fewer edges. The average degree is just below 2, with low degree variance, indicating very sparse and consistent connectivity.
- Undirected planar graphs are similar to undirected sparse graphs in terms of average degree and degree variance but have fewer nodes and edges. They maintain sparse and consistent connectivity.
- Undirected regular graphs have a higher average number of edges and degree compared to sparse and planar graphs, indicating denser connectivity. The low degree variance suggests uniformity in node connectivity.
- Undirected complete graphs have a very high number of edges, as expected. Each node is connected to every other node, resulting in the maximum possible average degree for the given number of nodes. The degree variance is zero, indicating perfect uniformity in connectivity.

B Time and Memory Usage

In addition to evaluating the passing rates of LLMs on our dataset, we also present the time and memory usage data, summarized in Figure 12. These metrics are crucial for assessing the efficiency of the code generated by LLMs.

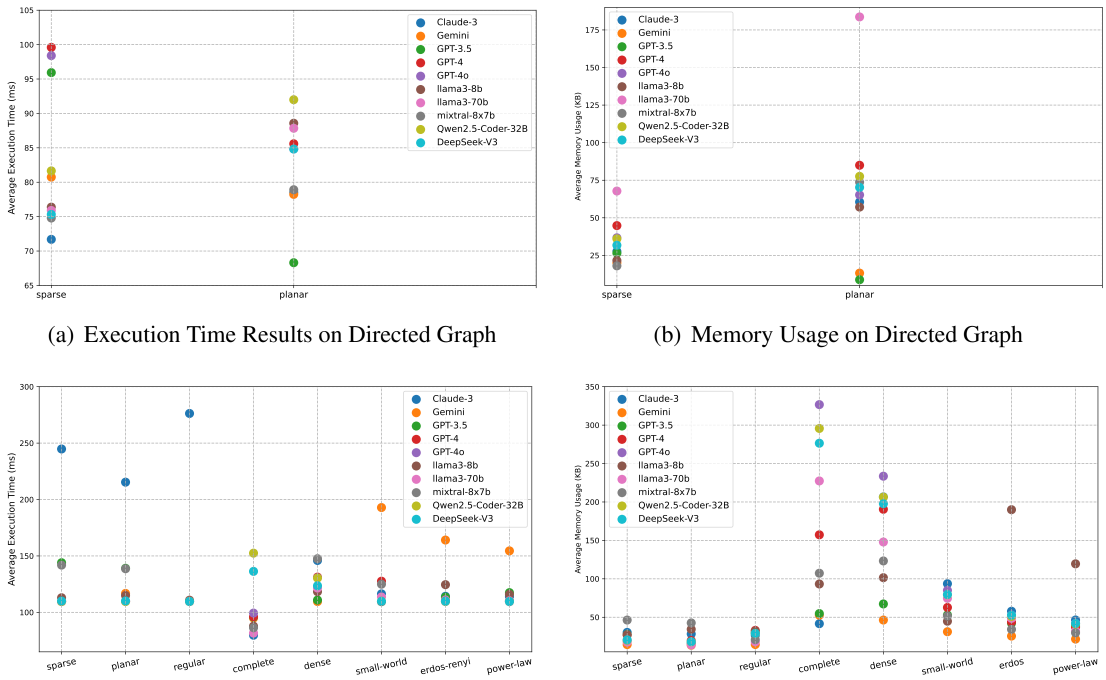
B.1 Results on Directed Graphs

For directed graphs, GPT-3.5 stands out as the most efficient model, demonstrating the lowest average execution times and memory usage across planar and sparse graphs. Claude-3-sonnet also performs well, particularly for sparse graphs, achieving the lowest execution time. Gemini-pro and Mixtral-8x7b show good memory efficiency, with Mixtral-8x7b being particularly efficient for sparse graphs. Conversely, Llama-3-70b consistently exhibits the highest memory usage, making it the least efficient in terms of memory management. Although GPT-4 is powerful, it has higher execution times and memory usage compared to GPT-3.5. Both Qwen2.5-Coder-32B and DeepSeek-V3 show lower usage of time and memory, compared to other LLMs. Overall, GPT-3.5 emerges as the most balanced and efficient model across the tested scenarios, whereas Llama-3-70B exhibits notable inefficiencies, particularly in memory consumption. However, this analysis is based solely on execution time and memory usage; in practice, researchers should also consider model accuracy to obtain a more comprehensive evaluation.

B.2 Results on Undirected Graphs

The performance of models on undirected graphs varies significantly across different graph types. Claude-3-sonnet displays notably high execution times for planar, sparse, and regular graphs but performs exceptionally well with complete graphs, showing the lowest execution time albeit with relatively higher memory usage. Gemini demonstrates consistently low memory usage and good execution times, particularly excelling with sparse graphs. GPT-3.5 and GPT-4 show balanced performance, with GPT-3.5 achieving better execution times for planar graphs and GPT-4 managing memory more efficiently except for complete graphs, where it uses significantly more memory. GPT-4o is similar to GPT-4 but with slightly improved times and memory efficiency. Llama-3-8b and Llama-3-70b exhibit higher memory usage, especially for complete graphs, with Llama-3-70b showing the highest memory usage for complete graphs. Mixtral-8x7b presents high execution times and memory usage across all graph types but performs moderately well with regular graphs. For low-density graphs, Qwen2.5-Coder-32B and DeepSeek-V3 exhibit performance comparable to other LLMs. However, when processing more complex structures, such as complete graphs, both models maintain high execution time and memory consumption. Overall, Gemini and GPT-3.5 emerge as the most efficient models in terms of execution time and memory usage across various graph types, while Claude-3-sonnet and GPT-4 show significant variations based on the graph type.

It is important to note that high efficiency in code execution does not always guarantee a high passing rate. Occasionally, code may produce incorrect answers, which can result in accelerated program execution.



(a) Execution Time Results on Directed Graph

(b) Memory Usage on Directed Graph

(c) Execution Time Results on Undirected Graph

(d) Memory Usage on Undirected Graph

Figure 12: Time and Memory usage results of LLMs on `GraphEval36K`.

C LLM Performance on Directed and Unidirected Graphs

In the experiment section, we found that the passing rate of directed graphs is generally higher than that of undirected graphs. However, it is difficult to make a definitive statement since the Leetcode problems on directed and undirected graphs may not be at a similar level of difficulty. Also, we did not find analogous problems involving undirected graphs and directed graphs within Leetcode and so it is difficult to compare the performance of LLMs on directed and undirected graphs. We compute the statistical significance of the gap between directed graphs and undirected graphs (assuming that the ensemble of problems had a similar level of difficulty). We found that across most categories and models, there were statistically significant differences in the performance outcomes.

The p-value results in Table 5 reveal notable differences in the performance of LLMs when solving graph-related problems on directed versus undirected graph samples. Among the eight models evaluated, Claude-3, Gemini, GPT-3.5, Llama-3-8b, and Mixtral-8x7b demonstrate statistically significant differences, with p-values well below the conventional threshold of 0.05. This suggests that these models exhibit distinct behavior or performance when handling directed and undirected graphs. In contrast, GPT-4, GPT-4o, and Llama-3-70b show no statistically significant difference, as indicated by their higher p-values (all above 0.05), implying that these models handle directed and undirected graph structures similarly. The variance in statistical significance across models may point to differing architectural strengths or limitations in their ability to generalize graph-based reasoning across different types of graph inputs. Overall, these results highlight the need for further investigation into model-specific behavior on varying graph structures, as some LLMs clearly struggle more than others with the distinction between directed and undirected graphs. For completeness, we also show examples of LLM reasoning on directed and undirected graphs in Section F.1.

C.1 Comparison with Traditional Graph Machine Learning (ML) methods

The objective of our work is to establish a benchmark that evaluates how well LLMs understand and reason about graph data structures, rather than comparing their performance with specialized graph machine learning models. Graph machine learning models (e.g., GNNs) are designed explicitly to solve graph-based problems and are optimized for such tasks. In contrast, our focus is on under-

Model	P-value	Statistical Significance
Claude-3-sonnet	3.23e-6	Significant
Gemini-pro	2.16e-2	Significant
GPT-3.5	7.56e-3	Significant
GPT-4	8.28e-1	Not Significant
GPT-4o	8.02e-1	Not Significant
Llama-3-8b	1.36e-2	Significant
Llama-3-70b	6.93e-1	Not Significant
Mixtral-8x7b	1.79e-3	Significant

Table 5: Statistical Results of Directed vs. Undirected Graphs for LLMs.

standing the extent to which LLMs, which are not specifically trained on graph problems, can generalize their capabilities to this domain. Our target is to assess their broader applicability in scenarios where graph reasoning is required but where graph-specific models might not be available or practical.

Our framework tests LLMs with graph problem statements and generates a program to solve them. The correctness of the program serves as a direct indicator of the LLMs' understanding and reasoning abilities. Our evaluation is based on this criterion, which is different from the evaluation of specialized graph machine learning models. We show the capabilities of LLMs on graph-structure problems in Table 2, and we also demonstrate the improvement of LLMs with SSD in Table 3 and Table 7. These results provide insights into the reasoning capabilities of LLMs on graph problems and show that our work serves as a complementary effort to existing graph ML research by exploring a different aspect of LLM capabilities. For any problem instance, our characterization is binary (either the produced code solved the problem or not) as opposed to the statistical evaluation in graph ML. Table 6 expands further on this comparison.

C.2 Statistical Analysis

We compute the p-value to evaluate whether there are statistically significant differences in the passing rates of eight LLMs when applied to three types of graph structures. Small-world, Erdos-Renyi, and Power-law. A p-value below 0.05 indicates that the difference in performance between the models in the compared data sets is statistically significant, meaning that such a difference is unlikely to be due to random variation.

The statistical analysis of the passing rate results reveals that the differences in model performance between Small-world and Erdos-Renyi graph samples (p-value = 0.833) as well as between Small-world and Power-law graph samples (p-value = 0.762) are not statistically significant. These p-values, being substantially greater than typical significance thresholds (e.g., 0.05), indicate that there is no compelling evidence to suggest a significant difference in the performance of LLMs across these graph types. This implies that, from the perspective of coding and reasoning tasks involving connected graphs, the LLMs evaluated demonstrate comparable abilities when presented with Small-world, Erdos-Renyi, and Power-law structures.

Such findings are indicative of the generalization capabilities of LLMs in handling different types of graph topologies, which are commonly seen in practical scenarios. Given the similarities in model performance across these graph types, it can be inferred that the LLMs possess a consistent level of competence in reasoning and generating code for graph problems, regardless of the specific structural properties represented by these categories. This consistency is a promising outcome, suggesting that LLMs are potentially versatile in solving graph-related challenges irrespective of the complexity or type of underlying graph distributions. However, further investigation is needed to determine whether this generalizability holds for more complex graph characteristics beyond connectivity or across more challenging graph-related problem domains.

D Evaluation on the LeetCode Platform

We selected a total of 17 graph coding 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 are tasked with generating a complete code based on this prompt. The generated code is subsequently tested on the LeetCode platform to assess its accuracy and performance. The summarized results of this assessment are presented in Figure 11:

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

Feature	Our Work	Traditional ML Methods (e.g., GNNs)
Input Type	Language-based queries	Pure graph-structure data
Main Process	1. Understand graph problem queries 2. Generate code to solve the problem 3. Evaluate graph reasoning via 36k test cases	1. Apply graph algorithms directly 2. Analyze graph structures
Understanding and Reasoning	Uses pre-trained knowledge and language understanding to interpret **graph-related queries**	Uses **graph topology** and properties for algorithmic reasoning
Key Strengths	**Code generative capabilities**, natural language queries, and unstructured inputs	High accuracy and efficiency in specific graph tasks
Typical Applications	Language-interactive graph data analysis, automatic **code generation**, interpreting complex graph data queries	Node classification, graph classification, link prediction

Table 6: Comparison between our work and traditional machine learning methods.

- 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.

E SSD Results on Difficulty Levels

We evaluate three versions of GPT models (GPT-3.5, GPT-4, and GPT-4o) on different difficulty levels of graph problems (same as the settings in Figure 11). We focus on the hard-level problems due to the near-perfect performance on easier tasks. We compare the performance of SSD method with the CoT-based method. The results are shown in Table 7. Our method outperforms the CoT-based method on three GPT models, improving the passing rate from 0% to 11.11% for GPT-3.5, and from 33.30% to 66.67% for GPT-4 and GPT-4o. The results confirm the effectiveness of SSD in enhancing LLMs' graph-solving capabilities. We observe a smaller improvement for GPT-3.5, likely due to its smaller model size and limited knowledge, making it less effective on hard-level graph tasks.

	GPT-3.5	GPT-4	GPT-4o
Easy-level (3)	100.00%	100.00%	100.00%
Medium-level (5)	80%	100.00%	100.00%
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%**

Table 7: 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.

F Complete Reasoning Example

Here, we provide further details on the SSD-enhanced graph reasoning and CoT-based graph reasoning: Figure 13 and Figure 14 ("SSD-enhanced GPT-4o's Graph Reasoning" and "CoT-based GPT-4o's Graph Reasoning").

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

<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: Each pair of vertices has at most one edge between them. No self-loops (edges where a vertex has an edge to itself).
<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>
Algorithm:

- **Breadth-First Search (BFS):**
  - Initialize an adjacency list from the edges.
  - For each vertex, perform a BFS to detect the shortest cycle starting from that vertex:
    - Keep track of visited nodes and the level of each node during the BFS using a queue.
    - When a visited node is encountered, calculate the potential cycle's length and update the shortest cycle length if this cycle is shorter than the previously found ones.
- **Handling Edge Cases:**
  - If no cycles are found after all vertices are processed, return -1.

<Program Implementation>
(the code is shown in Code (3)).

Figure 13: Graph Reasoning by GPT-4o with SSD.

✘ 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 Code (4), 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.

Figure 14: Graph Reasoning by GPT-4o with CoT.

The detailed code for SSD-enhanced Graph Reasoning and CoT-based Graph Reasoning is provided in Listings 1 and 2, respectively. Notably, the instances of hallucinations in the CoT-based Graph Reasoning are highlighted for clarity.

```
class Solution:
    def findShortestCycle(self, n:int, edges:List[List[int
      ]]) -> int:
        adjacency_list = [[] for _ in range(n)]
        for u, v in edges:
            adjacency_list[u].append(v)
            adjacency_list[v].append(u)

        shortest_cycle = float('inf')

        def bfs(start):
            nonlocal shortest_cycle
            queue = deque([(start, -1, 0)])
            visited = {start: 0}

            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)

        return -1 if shortest_cycle == float('inf') else
      shortest_cycle
```

Code 3: Code generated by the SSD-enhanced GPT-4o.

```
class Solution:
    def findShortestCycle(self, n:int, edges:List[List[int
      ]]) -> int:
        graph = defaultdict(list)
        for u, v in edges:
            graph[u].append(v)
            graph[v].append(u)

        def bfs(start):
            dist = [-1] * n
            dist[start] = 0
            queue = deque([(start, -1)])
            while queue:  # Hallucination Happens!
                current, parent = queue.popleft()
                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))

        return -1 if shortest_cycle == float('inf') else
      shortest_cycle
```

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

F.1 Reasoning Examples of LLM on Directed and Undirected Graphs

For completeness, we show below how the reasoning process proceeds for undirected and directed graphs with GPT-4o.

✔ GPT-4o Reasoning on **Undirected** Graphs

[Undirected Graph Example]
There is an infrastructure of n cities with some number of roads connecting these cities. Each roads[i] = [ai, bi] indicates that there is a bidirectional road between cities ai and bi. The network rank of two different cities is defined as the total number of directly connected roads to either city. If a road is directly connected to both cities, it is only counted once. The maximal network rank of the infrastructure is the maximum network rank of all pairs of different cities. Given the integer n and the array roads, return the maximal network rank of the entire infrastructure.
[GPT-4o's Reasoning Steps]

- Step 1: Initialize Data Structures
  - We need an array degree to store the number of roads connected to each city.
  - We also need a set direct_connection to keep track of whether two cities are directly connected.

- Step 2: Populate the Degree Array and Direct Connection Set
  - Iterate through each road in the roads array. For each road connecting cities ai and bi:
  a. Increment the degree of both ai and bi.
  b. Add the pair (ai, bi) (or (bi, ai)) to the direct_connection set.

- Step 3: Calculate the Maximal Network Rank
  - Iterate over all pairs of different cities (i, j):
  a. Calculate their combined network rank as degree[i] + degree[j].
  b. If cities i and j are directly connected, subtract 1 from their combined network rank (because we don't want to count the direct road twice).
  - Keep track of the maximum network rank encountered.

- Step 4: Return the Result
  - Return the maximum network rank found during the iteration.

Figure 15: Reasoning process of GPT-4o on undirected graphs.

✔ GPT-4o Reasoning on **Directed** Graphs

[Directed Graph Example]
There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course ai first if you want to take course bi. For example, the pair [0, 1] indicates that you have to take course 0 before you can take course 1. Prerequisites can also be indirect. If course a is a prerequisite of course b, and course b is a prerequisite of course c, then course a is a prerequisite of course c. You are also given an array queries where queries[j] = [uj, vj]. For the jth query, you should answer whether course uj is a prerequisite of course vj or not. Return a boolean array answer, where answer[j] is the answer to the jth query.
[GPT-4o's Reasoning Steps]

- Step 1: Model the Courses as a Graph
  - Each course can be represented as a node in a directed graph, and a prerequisite relationship between two courses can be represented as a directed edge.

- Step 2: Build the Graph
  - We'll use an adjacency list to represent the graph. The adjacency list for the graph is a dictionary where each key is a course, and its value is a list of courses that depend on it (i.e., courses that can be taken after completing the key course).

- Step 3: Precompute Reachability with Floyd-Warshall Algorithm
  - Initialize a 2D matrix reachable where reachable[i][j] is True if there's a path from course i to course j.
  - Iterate over each prerequisite to update the matrix: if there's a direct prerequisite relation, set the corresponding entry in the matrix to True.
  - Use the Floyd-Warshall algorithm to update the matrix:
  a. For each intermediate course k, check if reachable[i][k] and reachable[k][j] are both True for any pair of courses (i, j). If so, set reachable[i][j] to True.

- Step 4: Return the Results
  - For each query [uj, vj], return reachable[uj][vj] as the answer.

Figure 16: Reasoning process of GPT-4o on directed graphs.

G Sample Problems under Different Graph Concepts

In this section, we will present problem samples under each graph concept from our dataset. We have 9 concepts in total: Topological Sorting, Path Planning, Path Finding, Cycle Detection, Graph Traversal, Parent-Child Relations, Graph Construction, DAG Related Calculations and General Graph Problems. Note that we are only presenting the "problem statement" here.

Concept: *Topological Sorting*
Problem Statement:
You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given a 2D integer array relations where relations[j] = [prevCoursej, nextCoursej] denotes that course prevCoursej has to be completed before course nextCoursej (prerequisite relationship). Furthermore, you are given a 0-indexed integer array time where time[i] denotes how many months it takes to complete the (i+1)th course. You must find the minimum number of months needed to complete all the courses following these rules: You may start taking a course at any time if the prerequisites are met. Any number of courses can be taken at the same time.Return the minimum number of months needed to complete all the courses. Note: The test cases are generated such that it is possible to complete every course (i.e., the graph is a directed acyclic graph)

Concept: *Path Planning*
Problem Statement:
You are given a directed graph of n nodes numbered from 0 to n - 1, where each node has at most one outgoing edge. The graph is represented with a given 0-indexed array edges of size n, indicating that there is a directed edge from node i to node edges[i]. If there is no outgoing edge from node i, then edges[i] == -1. Return the length of the longest cycle in the graph. If no cycle exists, return -1. A cycle is a path that starts and ends at the same node.

Concept: *Path Finding*
Problem Statement:
There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1 (inclusive). The edges in the graph are represented as a 2D integer array edges, where each edges[i] = [ui, vi] denotes a bi-directional edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself. You want to determine if there is a valid path that exists from vertex source to vertex destination. Given edges and the integers n, source, and destination, return true if there is a valid path from source to destination, or false otherwise.

Concept: *Cycle Detection*
Problem Statement:
You are given a positive integer n representing the number of nodes in an undirected graph. The nodes are labeled from 1 to n. You are also given a 2D integer array edges, where edges[i] = [ai, bi] indicates that there is a bidirectional edge between nodes ai and bi. Notice that the given graph may be disconnected. Divide the nodes of the graph into m groups (1-indexed) such that: Each node in the graph belongs to exactly one group. For every pair of nodes in the graph that are connected by an edge [ai, bi], if ai belongs to the group with index x, and bi belongs to the group with index y, then |y - x| = 1. Return the maximum number of groups (i.e., maximum m) into which you can divide the nodes. Return -1 if it is impossible to group the nodes with the given conditions.

Concept: *Graph Traversal*
Problem Statement:
There is an undirected graph with n nodes, numbered from 0 to n - 1. You are given a 0-indexed integer array scores of length n where scores[i] denotes the score of node i. You are also given a 2D integer array edges where edges[i] = [ai, bi] denotes that there exists an undirected edge connecting nodes ai and bi. A node sequence is valid if it meets the following conditions: There is an edge connecting every pair of adjacent nodes in the sequence. No node appears more than once in the sequence. The score of a node sequence is defined as the sum of the scores of the nodes in the sequence. Return the maximum score of a valid node sequence with a length of 4. If no such sequence exists, return -1.

Concept: *Parent-Child Relations*
Problem Statement:
You are given an integer n denoting the number of cities in a country. The cities are numbered from 0 to n - 1. You are also given a 2D integer array roads where roads[i] = [ai, bi] denotes that there exists a bidirectional road connecting cities ai and bi. You need to assign each city with an integer value from 1 to n, where each value can only be used once. The importance of a road is then defined as the sum of the values of the two cities it connects. Return the maximum total importance of all roads possible after assigning the values optimally.

Concept: *Graph Construction*
Problem Statement:
You are given an array pairs, where pairs[i] = [xi, yi], and there are no duplicates. xi < yi. Let ways be the number of rooted trees that satisfy the following conditions: The tree consists of nodes whose values appeared in pairs. A pair [xi, yi] exists in pairs if and only if xi is an ancestor of yi or yi is an ancestor of xi. Note: the tree does not have to be a binary tree. Two ways are considered to be different if there is at least one node that has different parents in both ways. Return 0 if ways equal to 0. Return 1 if ways equal to 1. Return 2 if ways are larger than 1. A rooted tree is a tree that has a single root node, and all edges are oriented to be outgoing from the root. An ancestor of a node is any node on the path from the root to that node (excluding the node itself). The root has no ancestors.

Concept: *DAG Related Calculations*
Problem Statement:
You are given a directed graph of n nodes numbered from 0 to n - 1, where each node has at most one outgoing edge. The graph is represented with a given 0-indexed array edges of size n, indicating that there is a directed edge from node i to node edges[i]. If there is no outgoing edge from node i, then edges[i] == -1. Return the length of the longest cycle in the graph. If no cycle exists, return -1. A cycle is a path that starts and ends at the same node.

Concept: *General Graph Problems*
Problem Statement:
There is an undirected graph consisting of n nodes numbered from 1 to n. You are given the integer n and a 2D array edges where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi. The graph can be disconnected. You can add at most two additional edges (possibly none) to this graph so that there are no repeated edges and no self-loops. Return true if it is possible to make the degree of each node in the graph even, otherwise return false. The degree of a node is the number of edges connected to it.

H An Example of How LLMs Solve Difficult Problems

In this section, we will present how do LLMs solve a difficult concept problem. We choose Claude-3, Gemini, GPT-4 and GPT-4o to be verification models. First, we will present the chosen problem. And then, the reasoning and coding process of LLMs will be shown.

Graph Question - Concept: Path Finding

[Problem Statement]
You are given three positive integers n, x, and y. In a city, there exist houses numbered 1 to n connected by n streets. There is a street connecting the house numbered i with the house numbered i + 1 for all 1 <= i <= n - 1 . An additional street connects the house numbered x with the house numbered y. For each k, such that 1 <= k <= n, you need to find the number of pairs of houses (house1, house2) such that the minimum number of streets that need to be traveled to reach house2 from house1 is k. Return a 1-indexed array result of length n where result[k] represents the total number of pairs of houses such that the minimum streets required to reach one house from the other is k. Note that x and y can be equal.
[Data Examples]
Input: n = 3, x = 1, y = 3 Output: [6,0,0]
Input: n = 5, x = 2, y = 4 Output: [10,8,2,0,0]

GPT-4o Solution

Steps to solve:
1. Initialize result array: Start with an array of zeroes to store the count of pairs.
2. Iterate through pairs: For each pair (i, j), compute both the direct distance and the alternative path distance using the extra street between x and y.
3. Calculate min distance: Find the minimum of the two distances and update the corresponding index in the result array.
4. Return the result: Once all pairs are processed, return the result array.

```
class Solution:
    def countOfPairs(self, n: int, x: int, y: int) -> List[
      int]:
        # Initialize the result array with n zeroes
        result = [0] * n

        # Adjust x and y for 0-based indexing
        x -= 1
        y -= 1

        # Iterate over all possible pairs of houses (i, j)
        for i in range(n):
            for j in range(i + 1, n):
                # Direct distance between i and j
                direct_dist = j - i

                # Distance via the special street between x
      and y
                via_x_y = abs(i - x) + 1 + abs(j - y)
                via_y_x = abs(i - y) + 1 + abs(j - x)

                # Minimum distance between the direct path
      and the path via x and y
                min_dist = min(direct_dist, via_x_y, via_y_x
      )

                # Increment the count for the corresponding
      distance
                result[min_dist] += 1

        return result
```

Code 5: Code generated by the GPT-4o.

GPT-4 Solution

Steps to solve:
1. Construct the Graph: We'll use a simple adjacency list to represent the graph.
2. Shortest Path Calculation: For small values of n, a simple Floyd-Warshall algorithm can be considered since it computes shortest paths for all pairs. But the time complexity is too high. Given the specific graph structure (almost a path plus one edge), BFS from each node could be more efficient, leveraging the linear structure for faster computation.
3. Count the Pairs: After determining the shortest path distances, count the number of pairs for each distance.

```
class Solution:
    def countOfPairs(self, n: int, x: int, y: int) -> List[
      int]:
        # Create the graph
        graph = defaultdict(list)
        for i in range(1, n):
            graph[i].append(i + 1)
            graph[i + 1].append(i)
        if x != y:  # Add the additional connection if x and
     y are not the same
            graph[x].append(y)
            graph[y].append(x)

        # Function to calculate distances using BFS
        def bfs(start):
            distances = [-1] * (n + 1)
            queue = deque([start])
            distances[start] = 0
            while queue:
                current = queue.popleft()
                current_dist = distances[current]
                for neighbor in graph[current]:
                    if distances[neighbor] == -1:
                        distances[neighbor] = current_dist +
     1
                        queue.append(neighbor)
            return distances

        # Distance array where result[k] counts the pairs of
     distance k
        result = [0] * (n + 1)  # Using n+1 for convenience,
     will slice later

        # Calculate distances for each house and count the
    pairs
        for i in range(1, n + 1):
            distances = bfs(i)
            for j in range(i + 1, n + 1):  # Only count each
     pair once
                if distances[j] > 0:
                    result[distances[j]] += 1

        return result[1:n]  # We return the list from 1 to n
    -1 as specified
```

Code 6: Code generated by the GPT-4.

Gemini Solution

Steps to solve:
1. Graph Representation: Create an adjacency list to represent the graph, where each node represents a house and edges connect adjacent houses.
2. BFS for Distances: Perform breadth-first search (BFS) starting from house x to calculate the shortest distances to all other houses.
3. Distance-Based Counting: Iterate through all pairs of houses and count the number of pairs where the sum of their distances is less than n (to avoid counting pairs that wrap around the circle).
4. Result Array: Store the counts in the result array, where result[k] represents the number of pairs with a distance of k.

```
class Solution:
    def countOfPairs(self, n: int, x: int, y: int) -> List[
      int]:
        # Create a graph representation using an adjacency
    list
        graph = [[] for _ in range(n)]
        graph[x - 1].append(y - 1)
        graph[y - 1].append(x - 1)

        # Perform breadth-first search (BFS) to calculate
    distances
        queue = [(0, x - 1)]  # Start with distance 0 from
    house x
        visited = set()
        distances = [0] * n
        while queue:
```

```
            distance, node = queue.pop(0)
            if node not in visited:
                visited.add(node)
                distances[node] = distance
                for neighbor in graph[node]:
                    if neighbor not in visited:
                        queue.append((distance + 1, neighbor
    ))

        # Count pairs based on distances
        result = [0] * n
        for i in range(1, n):
            for j in range(i + 1, n):
                distance = distances[i - 1] + distances[j -
    1]
                if distance < n:
                    result[distance] += 1

        return result
```

Code 7: Code generated by the Gemini.

Claude-3 Solution

Steps to solve:
1. We start by initializing a distance matrix dist with dimensions n x n, setting all initial distances to infinity.
2. We set the distances for adjacent houses (i and i+1) to 1, and the distance from each house to itself to 0. We also set the special connection between houses x and y to 1.
3. We use the Floyd-Warshall algorithm to compute the shortest paths between all pairs of houses. This algorithm has a time complexity of O(n^3), but it's efficient for this problem size and gives us all the information we need.
4. After computing all shortest paths, we count the number of pairs for each distance. We iterate through all pairs of houses (i, j) where i < j, and increment the count for the corresponding distance. We increment by 2 to account for both (i, j) and (j, i) pairs.
5. Finally, we return the count array, which represents the number of pairs for each distance from 1 to n.

```
class Solution:
    def countOfPairs(self, n: int, x: int, y: int) -> List[
    int]:
        # Initialize the distance matrix
        dist = [[float('inf')] * n for _ in range(n)]

        # Set distances for adjacent houses and the special
    connection
        for i in range(n):
            if i > 0:
                dist[i][i-1] = dist[i-1][i] = 1
            dist[i][i] = 0
        dist[x-1][y-1] = dist[y-1][x-1] = 1

        # Floyd-Warshall algorithm to compute all-pairs
    shortest paths
        for k in range(n):
            for i in range(n):
                for j in range(n):
                    dist[i][j] = min(dist[i][j], dist[i][k]
    + dist[k][j])

        # Count pairs for each distance
        count = [0] * n
        for i in range(n):
            for j in range(i+1, n):
                d = dist[i][j]
                if d < n:
                    count[d-1] += 2  # Count both (i,j) and
    (j,i)

        return count
```

Code 8: Code generated by the Claude-3.