GTA: Graph Theory Agent and Benchmark for Algorithmic Graph Reasoning with LLMs

Zixiang Xu[1], Yanbo Wang[2], Chenxi Wang[3], Lang Gao[3], Zirui Song[3]
Yue Huang[4], Zhaorun Chen[5], Xiangliang Zhang[4], Xiuying Chen[3]*

[1] *University of Southern California* [2] *University of California, Los Angeles*
[3] *Mohamed bin Zayed University of Artificial Intelligence*
[4] *University of Notre Dame* [5] *University of Chicago*

Abstract

Large Language Models (LLMs) are increasingly asked to reason over structured data such as graphs, yet how reliably they can carry out multi-step graph algorithms in language remains unclear. Existing evaluations tend to use simple tasks on small graphs, to score code generation rather than reasoning over the graph itself, or to fix a single input format. We introduce **Graph Theory Bench (GT Bench)**, a benchmark covering 24 classical graph problems in 44 task–structure settings, with over 100,000 examples across four representations: natural language, structured language, adjacency list, and adjacency matrix. Evaluating eight LLMs on GT Bench shows that accuracy is strongly tied to the input representation, that the best representation shifts with graph density, size, and topology as well as with the model, and that this sensitivity persists, attenuated, in the strongest reasoning models. Building on these observations, we propose the **Graph Theory Agent (GTA)**, which pairs a preference-trained representation selector with plan-and-decompose scaffolding around a frozen executor LLM. GTA lifts Phi-4 from 53.5% to 69.1% on the benchmark's easy split and from 33.0% to 41.5% on its hard split, outperforming eight prompting and agent baselines, and transfers without retraining to GraCoRe and NLGraph. The benchmark, generation pipeline, and agent are open-source. Code: `https://github.com/xzx34/GTA`.

1 Introduction

Large Language Models (LLMs) have moved well beyond text processing into social-science applications (Huang et al., 2024b; Xu et al., 2026f), medicine (Zhou et al., 2024; Tian et al., 2024; Liu et al., 2025c; Chen et al., 2025), and robotics (Song et al., 2025). As their reach widens, they are increasingly expected to reason about structured data, including social networks (Wang et al., 2025b), interaction graphs for session search (Wu et al., 2024), and molecular structures (Ross et al., 2022). These applications connect entities through relations that can be represented as vertices and edges. Working with such data takes more than recognizing that a graph is present; the model must interpret its structure and sustain multi-step algorithmic reasoning over it.

Existing graph benchmarks evaluate several distinct capabilities. GraphEval36K (Wu et al., 2025) measures graph problem solving through generated code, whereas NLGraph (Wang et al., 2023a) and GraCoRe (Yuan et al., 2025) evaluate answers to structural and relational questions over graph inputs. GRBench (Jin et al., 2024) evaluates question answering with knowledge from text-attributed graphs. Representation studies further show that graph encoding can change LLM performance (Fatemi et al., 2024). These complementary settings motivate a controlled evaluation that combines diverse classical graph problems, equivalent input encodings, and language-only execution.

To support this evaluation, we introduce **Graph Theory Bench (GT Bench)**, a benchmark specifically designed to assess the multi-step algorithmic reasoning capabilities of LLMs on graph-theoretic problems. GT

*Corresponding author: Xiuying Chen (`xiuying.chen@mbzuai.ac.ae`).

Bench covers 24 classical graph problems in 44 task–structure settings, totaling more than 100,000 examples across four representations. The examples are stratified into *easy* and *hard* subsets covering tasks from basic connectivity checks to complex problems like minimum-cost maximum-flow (MCMF). Each instance is presented in four input modalities (natural language, structured language, adjacency list, adjacency matrix) to test model robustness and the impact of representation on reasoning. A parameterized generation pipeline supports flexible control of graph families (e.g., directed/undirected, weighted/unweighted, sparse/dense, tree-structured) and sizes, with node and edge counts calibrated to contemporary LLM context windows: large enough to require genuine multi-step reasoning yet compact enough to fit reliably within prompts. Unlike benchmarks focusing on code generation or rudimentary queries, GT Bench targets tasks requiring genuine algorithmic thinking and combinatorial reasoning. The dataset, generators, and evaluation scripts are fully open-source.

Our experiments on GT Bench expose a strong dependency between input representation and LLM performance. On a single task the choice of format can move accuracy by close to thirty points (Connectivity on sparse graphs spans 60.5% to 89.1% across the four formats), and the winning format varies systematically with graph structure: adjacency matrices excel on dense graphs, natural language on sparse graphs, and structured language on trees. Building on this finding, we propose the **Graph Theory Agent (GTA)**, a framework that improves LLM graph reasoning through adaptive input representation selection and systematic decomposition of the solution into manageable sub-steps. GTA operates under a *frozen executor* paradigm, keeping the base LLM unmodified while optimizing lightweight auxiliary components for representation selection and algorithmic decomposition.

Our focus on language-only reasoning is deliberate: it measures whether a model can interpret a graph, maintain intermediate algorithmic state, and carry out a sequence of dependent operations. GT Bench includes polynomial-time problems such as shortest path and minimum-cost maximum-flow, as well as NP-hard optimization problems such as Maximum Clique and Maximum Independent Set and NP-complete decision problems such as Hamiltonian Path and Circuit. Evaluating these tasks through language exposes the reasoning steps performed by the model itself. Code-augmented evaluation measures an additional capability: translating a problem into an executable algorithm and using its output.

Our main contributions are: (1) GT Bench, a benchmark for multi-step algorithmic reasoning of LLMs on graph-theoretic problems, with four equivalent graph representations; (2) an empirical characterization of representation sensitivity, showing how the best format depends on task, graph structure, and model, and that the effect survives in strong reasoning models; and (3) GTA, an agent framework that adaptively selects representations and decomposes problem solving around a frozen executor, improving accuracy across benchmarks and executor models.

2 Related Work

2.1 LLMs and Graph Reasoning Benchmarks

Graph benchmarks differ in their inputs and execution protocols. GraphEval36K (Wu et al., 2025) evaluates generated programs on graph problems. NLGraph (Wang et al., 2023a) covers eight tasks, including maximum flow and Hamiltonian path; GPT4Graph (Guo et al., 2023) evaluates structural and semantic understanding; and GraCoRe (Yuan et al., 2025) covers 19 tasks on pure and heterogeneous graphs. GRBench (Jin et al., 2024) focuses on question answering over text-attributed graphs, while ProofWriter (Tafjord et al., 2021) tests rule-based inference in natural language. HGB (Li et al., 2024d) provides hybrid-graph datasets for graph neural networks, and HiGPT (Tang et al., 2024) addresses heterogeneous graph learning. VisionGraph (Li et al., 2024c) evaluates multi-step graph-theoretic reasoning from visual inputs. GraphArena (Tang et al., 2025) covers ten polynomial-time and NP-complete tasks and distinguishes correct, suboptimal, hallucinatory, and missing answers. ProGraph (Li et al., 2024b) evaluates library-calling programs; GrAlgoBench (Zhang et al., 2026b) reports accuracy below 50% beyond 120 nodes in its evaluation of large reasoning models; LLM4Hypergraph (Feng et al., 2025) studies seven hypergraph languages; and LLM4DyG (Zhang et al., 2024) probes spatial-temporal reasoning on dynamic graphs. GT Bench contributes a controlled comparison of four standard encodings across 44 task–structure settings for language-only algorithmic reasoning.

A closely related and fast-growing thread studies how graph *representation* affects LLM performance. Format sensitivity is not unique to graphs: Sclar et al. (2024) show that semantically equivalent changes to prompt formatting can swing accuracy by tens of points on general tasks, and the effect survives scale and instruction tuning. Graphs make the phenomenon acute, because the same structure admits several equally standard encodings. "Talk Like a Graph" (Fatemi et al., 2024) demonstrated that the choice among them matters, and a series of recent studies has sharpened this picture. Firooz et al. (2024) show that accuracy drops as related facts are placed farther apart in the serialized context; GraphSOS (Chu et al., 2025) finds that merely shuffling node and edge order can swing performance between strong and near-random, and learns to select serialization orders and sample informative subgraphs; Ge et al. (2025) similarly show that the order in which edges are described shifts accuracy in task-dependent ways; and Herbst et al. (2025) demonstrate that LLM graph reasoners are not invariant to node relabeling, edge reordering, or syntax, and that fine-tuning can even amplify this sensitivity. Others design new encodings, such as the interpretable color-based scheme of Zangari et al. (2026), or compare textualization strategies for knowledge graphs (Markowitz et al., 2025). Two contemporaneous efforts go further and act on the choice: CFPO (Liu et al., 2025b) searches jointly over prompt content and format per task, and DynamicGTR (Wei et al., 2026b) selects among textual and visual graph topology representations for vision-language models on graph QA. These works establish representation as a first-order factor, but each probes robustness, advocates a single canonical format, or optimizes the choice outside the language-only algorithmic-reasoning setting studied here. GT Bench instead measures representation sensitivity systematically across 44 task–structure settings, four standard representations, and eight models, and GTA turns the choice into an *adaptive*, per-instance selection coupled with plan-decompose-execute, rather than seeking one fixed best format.

Contextual robustness extends beyond graph serialization. Adaptive Distraction uses automated tree search to construct contextual distractions intended to preserve the original question and answer (Wang et al., 2025f), while Cross-Lingual Pitfalls searches for bilingual question pairs that expose inconsistent performance across languages (Xu et al., 2025e). Babel Tower examines cross-lingual differences in forward and reverse causal reasoning (Wang et al., 2025c), and Word Form Matters studies semantic reconstruction under scrambled internal letter order (Wang et al., 2025a). These studies expose sensitivity to how information is presented. GT Bench isolates the effect of encoding a fixed graph and task in different formats.

The use of contextual evidence can also be checked during reasoning. ContextGuard organizes self-auditing around evidence and constraints (Jin et al., 2026a), and premise verification decomposes a question into checkable propositions before answering (Qin et al., 2026b). Related benchmarks examine complementary aspects of multi-step reasoning: FineLogic separates answer correctness from intermediate logical soundness (Zhou et al., 2025b), while Origami and JigShape test geometric planning and constraint satisfaction (Agarwal et al., 2026; Li et al., 2026b). GT Bench supplies explicit structural constraints and algorithmically verifiable targets for its graph tasks.

Benchmark construction and diagnostic resolution further affect what an evaluation reveals. ProbeLLM searches for failure-revealing inputs (Huang et al., 2026a); DataGen supports controllable synthetic dataset construction (Huang et al., 2025a); and RefineLab selects data-refinement operations under a token budget (Luo et al., 2026). For reasoning traces, AgentAuditor resolves disagreements through local verification within reasoning trees (Yang et al., 2026c). GT Bench combines parameterized generation with computed reference answers, while reporting the separate plan-alignment diagnostic in section 6.3.

The difficulty of graph tasks for transformers also has a theoretical basis. Sanford et al. (2024) place graph algorithms in a representational hierarchy, and Saparov et al. (2025) identify graph-search limitations tied to training distributions and graph size. Complementary work on formal verification examines errors in LLM-generated abstract-interpretation steps (Mitchell et al., 2025). These results motivate explicit evaluation of multi-step reasoning and scaling; the effect of GTA's decomposition is measured by our own ablations (Table 9).

What GT Bench adds. GT Bench targets multi-step algorithmic reasoning under realistic structural and computational constraints. It spans a broad set of classical graph problems and provides multiple input modalities (natural/structured language, adjacency list/matrix) to stress-test language-based reasoning

without relying on code execution or simple lookup. To make the positioning concrete, we summarize key differences with representative benchmarks:

As Table 1 shows, GT Bench combines classical problem coverage, equivalent graph encodings, and language-only answers. Its representation study is paired with an agent framework and cross-benchmark transfer evaluation.

Relation to GraphOmni and GraphAlgorithm. GraphOmni (Xu et al., 2026a) is closely related: both study representation sensitivity and adaptive selection. GraphOmni covers six core tasks with two additional NP-hard stress tests and jointly optimizes serialization and prompting. GT Bench covers 24 classical problems in 44 task–structure settings, and GTA combines representation selection with algorithmic plan–decompose–execute scaffolding. This decomposition adds 7.0 points on GT-E over the Selector-only configuration (Table 9), with further validation on GraCoRe and NLGraph. GraphAlgorithm (Hu et al., 2025d) instead evaluates algorithm design followed by code generation and execution; its Simple-RTC method reports 92.7–99.9% accuracy on five existing benchmarks. This protocol measures executable algorithmic solutions, while GT Bench evaluates answers produced in language. AlgoWorlds (Xu et al., 2026e) extends tool-mediated evaluation to globally optimal combinatorial decisions in partially observed environments, sharing algorithmically verifiable objectives with GT Bench while using a different observation and execution interface.

2.2 Attempts to Solve Graph-Theoretic Problems with LLMs

Several recent systems attempt to apply LLMs to graph tasks, but most do not directly target language-based algorithmic reasoning over graphs.

Graph Agent (Wang et al., 2023c) uses explicit inductive and deductive reasoning for node classification and link prediction. LLM-GOOD (Xu et al., 2025c) combines LLM-assisted annotation with a lightweight graph neural network for few-shot out-of-distribution detection on text-attributed graphs. These prediction tasks differ from executing classical graph algorithms in language.

Relational representation learning distinguishes constructing a useful graph representation from executing an algorithm on a given graph. NQE embeds multi-hop logical queries on n-ary knowledge graphs (Luo et al., 2023c); DHGE couples instance-level and ontology-level representations (Luo et al., 2023a); and HAHE combines global and local attention for hyper-relational knowledge graphs (Luo et al., 2023b). Text2NKG constructs n-ary knowledge graphs from text (Luo et al., 2024a). GT Bench starts from an explicitly specified graph and evaluates the requested computation across equivalent encodings.

Recent work on text-attributed graphs connects language models to data selection and distribution shifts. GOE-LLM generates outlier exposure data for graph out-of-distribution detection (Xu et al., 2025d), and LEGO-Learn studies label-efficient learning on open-set graphs (Xu et al., 2025a). TAG-AD evaluates contextual and structural anomalies and introduces retrieval-assisted zero-shot detection (Xu et al., 2025b). Together with LLM-GOOD, these prediction-oriented methods separate the contributions of semantic content, graph structure, and model choice, complementing the algorithmic setting of GT Bench.

Graph-learning robustness also distinguishes prediction stability from explanation stability. Small structural perturbations can change graph explanations even when predictions remain stable (Li et al., 2024a), motivating certification methods such as XGNNCert (Li et al., 2025a). These studies perturb the graph itself. GT Bench keeps the graph invariant, so differences across its encodings arise without changing the underlying problem.

Explicit intermediate structure is also relevant to reasoning fidelity. CDCR-SFT trains models to construct causal directed acyclic graphs and reason over them (Li et al., 2026c). Work on cooperative rationalization shows that an accurate predictor can still exploit spurious correlations introduced by rationale selection, including on graph-classification data (Liu et al., 2025a). In classical program analysis, incremental algebraic program analysis reuses and updates symbolic path summaries after program changes (Zhou et al., 2025a). These studies connect structured intermediate states to causal reasoning, explanation fidelity, and path computation, respectively. They motivate evaluating the stages of a reasoning pipeline separately rather than inferring their reliability from final accuracy alone.

Benchmark	Task coverage	Reported scale	Input and execution protocol
GT Bench (Ours)	24 problems; 44 settings	105,600 encoded examples	Four equivalent graph encodings; language-only answers; easy/hard subsets.
GraCoRe	19 tasks	5,140 graphs	Pure and heterogeneous graphs; direct answers; node-order and semantic-feature analyses.
NLGraph	8 tasks	29,370 problems	Natural-language graphs; direct answers; task-specific difficulty subsets and prompting studies.
GraphOmni	6 core tasks; 2 extensions	241,726 core queries	Seven serializations and nine prompt schemes; three difficulty levels; adaptive serialization/prompt selection.
GraphAlgorithm	239 problems	3,041 test instances	Competition problem descriptions; algorithm reasoning followed by generated and executed code.

Table 1: Reported scope and evaluation protocols of graph benchmarks: GraCoRe (Yuan et al., 2025), NLGraph (Wang et al., 2023a), GraphOmni (Xu et al., 2026a), and GraphAlgorithm (Hu et al., 2025d). Scale units differ across sources and are stated explicitly. NLGraph's count is its extended set; GraphOmni's core count excludes its two additional NP-hard stress tests.

GraphAgent-Reasoner (Hu et al., 2025e) distributes node-centric computation across an agent network, with a master LLM coordinating the algorithm and summarizing final states. Its distributed execution protocol differs from GTA's single frozen executor.

GraphTeam (Li et al., 2025b) orchestrates multiple agents in a workflow (Original Question → Question Agent → Search Agent → Coding Agent), with a fallback Reasoning Agent when code fails. This *code-first* pipeline is effective for analysis workflows, but it *bypasses* intrinsic language-based graph reasoning by delegating problem solving to generated code and execution. Our objective is orthogonal: evaluate and enhance an LLM's *intrinsic* algorithmic reasoning over graph structure without relying on external code execution.

Beyond these, training- or alignment-centric approaches such as **GUNDAM** (Ouyang et al., 2024), **Graph-Thought** (Huang et al., 2025b), **GraphWiz** (Chen et al., 2024), and **GCoder** (Zhang et al., 2025d) explore instruction tuning, preference alignment, or thought-generation for graph tasks, and **GraphInstruct** (Luo et al., 2024b) contributes an instruction corpus with intermediate reasoning steps across 21 graph tasks. This line remains active in the reasoning-model era: **G1** (Guo et al., 2025b) applies reinforcement learning on a synthetic graph corpus so that a small model surpasses far larger ones, **NPG-Muse** (Wang et al., 2025i) elicits long chain-of-thought reasoning by training on NP-hard graph problems, Zhang et al. (2025c) continue pretraining on a 10.9B-token graph-problem corpus and report transfer to mathematical and logical reasoning, Zhang et al. (2026c) compare solution- and process-based post-training rewards on synthetic graph problems and evaluate transfer to real-world tasks, **GraphTool-Instruction** (Wang et al., 2025d) decomposes reasoning into subtasks that call external graph tools, and further work fine-tunes LLMs as end-to-end combinatorial-optimization solvers (Jiang et al., 2025) or as reasoning-then-coding agents (Hu et al., 2025d). These are promising directions, yet they typically modify or fine-tune the *executor* model itself or rely on tool/code execution. In contrast, our GTA explicitly (i) keeps the executor *frozen* and (ii) compels the model to *solve by reasoning* through an adaptive representation selector and a lightweight plan–decompose–execute pipeline. This separation isolates the benefit of better *reasoning and representation choices*, rather than changes in model capacity.

A further alternative changes the graph interface. GraphLLM (Chai et al., 2023) learns graph-enhanced prefixes for a frozen LLM, GraphToken (Perozzi et al., 2024) learns continuous soft-prompt encodings, and G-Retriever (He et al., 2024) couples a graph encoder with retrieval for textual graph question answering. These methods require embedding-space access or an auxiliary encoder. ReMindRAG (Hu et al., 2025c) instead guides knowledge-graph traversal with an LLM and reuses traversal experience for efficient retrieval. GTA selects among textual encodings of the complete problem graph and supplies algorithmic decomposition to a frozen executor.

2.3 LLM-Based Agent Frameworks

The view of LLMs as agents that plan, decompose, and act has gained momentum. Chain-of-Thought prompting (Wei et al., 2022) exposes intermediate reasoning steps, ReAct (Yao et al., 2023) interleaves reasoning with actions, and Toolformer (Schick et al., 2023) learns when to invoke tools. For graphs, Graph-CoT (Jin et al., 2024) alternates reasoning, graph interaction, and graph execution. AFlow (Zhang et al., 2025b) searches over workflows of LLM calls, whereas DyFlow (Wang et al., 2025g) uses a designer–executor architecture to revise subgoal plans in response to intermediate execution feedback.

Adaptive controllers can also alter how reasoning is carried out. AdaReasoner learns task-dependent reasoning configurations through an auxiliary policy (Wang et al., 2025e); InfoQA combines capacity-aware decomposition with pruning of earlier reasoning traces for multi-hop question answering (Wan et al., 2026); and Learning to Deliberate learns policies over persist, refine, and concede actions in multi-agent reasoning (Yang & Thomason, 2026). CoAct studies parallel agent execution through contrastive task allocation (Peng et al., 2026). GTA specializes adaptation to graph encoding and algorithmic decomposition. A distinct agents-as-topology setting is studied by AgentsNet (Grötschla et al., 2025), where node-level LLM agents coordinate on distributed graph problems and performance degrades as the communication network grows.

Workflow adaptation can be learned at several levels. SWIFT transfers structural priors and workflow demonstrations across tasks to synthesize workflows with a single generation pass (Du et al., 2026). OASES jointly trains search and state-evaluation policies to align intermediate search feedback with final outcomes (Zhang et al., 2026a). Unity-MAS converts workflows into role-conditioned trajectories for multi-agent reinforcement learning (Chen et al., 2026b), while MAESTRO trains execution and synthesis agents with conditional policy optimization (Yang et al., 2025c). Self-Compressing MAS learns to reduce redundant reasoning content (Chen et al., 2026a). These approaches optimize workflow structure, credit assignment, or communication. GTA uses a fixed high-level workflow and learns the representation choice and decomposition for its graph tasks.

Decomposition makes intermediate interfaces explicit across several agent settings. CogWriter combines hierarchical planning, parallel drafting, and review for long-form writing (Wan et al., 2025); AD-AGENT organizes data analysis, model selection, and code execution for anomaly detection (Yang et al., 2025a); and QUITE uses a finite-state multi-agent workflow for SQL rewriting with semantic and execution checks (Song et al., 2026a). These examples connect task decomposition to the checks available at each stage, whereas GTA executes the resulting graph-reasoning steps in language.

Control policies further determine how plans respond to observations. ASTRO learns planning strategies for non-cooperative dialogue (Hu et al., 2025b). Three-Step Nav combines global planning, local subgoals, and retrospective checking (Zheng et al., 2026); Ponder separates GUI-action interpretation from action localization (Wang et al., 2025h); and ManipLVM-R1 uses affordance information and verifiable trajectory rewards for manipulation (Song et al., 2026b). These systems illustrate the separation of high-level planning, intermediate representations, and execution in interactive tasks.

Reusable skills and memory offer another form of adaptation. Recipes for Agents describes skills as reusable procedural knowledge and examines their construction, composition, and evaluation (Xing et al., 2026). HyperSkill represents skills and trajectories in a hypergraph for retrieval and reuse (Xu et al., 2026c); SkillGen derives reusable skills from contrasting successful and unsuccessful trajectories (Ma et al., 2026). RaMem preserves the contextual conditions needed to interpret retrieved memories as evidence (Yang et al., 2026b). MemoHarness learns an external controller around a frozen model (Huang et al., 2026b), and MetaCollab combines autonomous execution, selective human intervention, and continual improvement (Yang et al., 2026a). These systems distinguish persistent experience from the current task state. GTA's cross-benchmark evaluation instead tests transfer of its learned auxiliary components without benchmark-specific retraining.

Selection matters even when the available components are fixed. MetaTool evaluates whether an LLM should use a tool and which one it should choose (Huang et al., 2024a), while AD-LLM evaluates model selection in anomaly detection (Yang et al., 2025b). MetaOOD and M3OOD use historical performance and task descriptors to select out-of-distribution detectors (Qin et al., 2025; 2026a); FlexRouter studies complementary

model sets for flexible LLM routing (Wei et al., 2026a). GTA fixes the executor and learns which graph encoding is most effective for the current instance.

Auxiliary components can also differ in the supervision used to train them. CoAct combines self-rewarding with active preference annotation (Xu et al., 2026b); it is distinct from the contrastive task-allocation framework of the same name (Peng et al., 2026). TerminalTraj generates verifiable terminal-agent trajectories (Wu et al., 2026), and Context-CoT constructs context-grounded reasoning traces for downstream training (Jin et al., 2026b). RMO reshapes reward-margin distributions (Ru et al., 2026), while DOG-DPO selects preference data through geometric coverage (Nian et al., 2026). These methods address the source or quality of supervision, which is relevant to GTA's preference-trained selector and decomposer.

Execution structure provides a complementary basis for diagnosis. GRADE represents dependencies and execution structure in agent runs (Zhao, 2026), and RiskLab varies interaction topology, protocols, agents, and tasks to study multi-agent failures (Jiang et al., 2026). GTA evaluates the effects of representation selection and decomposition through component ablations within its specified workflow.

Our Graph Theory Agent specializes this broader agent-design space to algorithmic graph reasoning. It combines an adaptive graph representation selector with plan–decompose–execute scaffolding around a frozen executor. The resulting design makes representation choice and decomposition directly testable through component ablations and transfer to graph benchmarks with different task distributions.

3 GT Bench: Graph Theory Benchmark

To rigorously evaluate LLM algorithmic reasoning on graphs, we constructed GT Bench to probe multi-step inference across diverse graph structures and graph-theoretic problems. Detailed task definitions and ground-truth algorithms are provided in Appendix A.

3.1 Graph Characteristics

The graphs within GT Bench are primarily categorized by edge density into sparse graphs, where the number of edges E scales linearly with the number of vertices V (i.e., $E = O(V)$), and dense graphs, where E approaches the quadratic maximum ($E = O(V^2)$). A third category comprises trees, defined as connected graphs satisfying $E = V - 1$. Depending on specific task requirements, graphs are generated with varying properties, such as weighted or unweighted edges, and may be connected or disconnected. Specific topological structures like star graphs are also implicitly generated within these categories.

3.2 Input Representations

A distinctive feature of GT Bench is that every problem instance is provided in four input modalities, enabling a controlled study of how representation choice affects reasoning performance. The four modalities are: **Natural Language (NL)**, a descriptive prose format detailing vertices, edges, and weights where applicable; **Structured Language (SL)**, which employs a more templated approach using keywords and indentation to articulate graph structure; the **Adjacency Matrix (AM)**, a matrix-based representation where entries denote edge existence or their associated weights; and the **Adjacency List (AL)**, which lists the neighbors and corresponding weights for each vertex. Figure 1 offers a visual comparison of these four representations applied to an exemplary small graph.

3.3 Task Coverage and Dataset Composition

GT Bench encompasses 24 classical graph problems, ranging from connectivity checks to optimization problems such as MCMF. Twenty problem types are evaluated on both sparse and dense graphs, and four are tree-specific, yielding 44 task–structure settings. Graph sizes are governed by a prompt-fit budget: node ranges are calibrated per task family so that every instance, under any of the four representations, fits comfortably within the context windows of contemporary LLMs, with tighter ranges for dense families whose edge count grows quadratically. The graphs remain large enough that answers require genuine multi-step

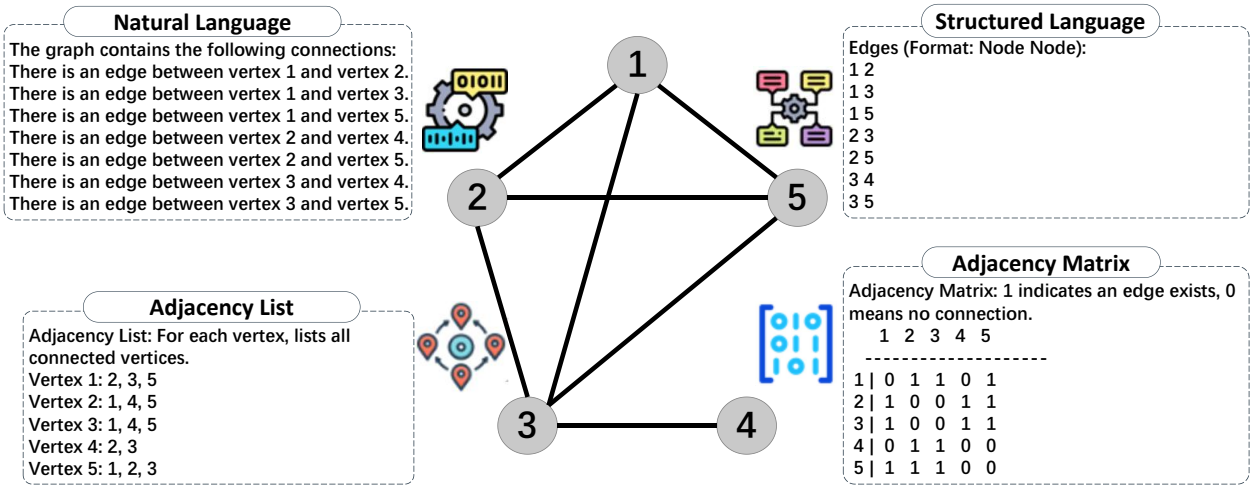

Figure 1: Example of an undirected graph represented using the four input modalities evaluated in this work: Natural Language, Structured Language, Adjacency List, and Adjacency Matrix.

reasoning rather than lookup. Table 2 details the resulting node counts and instance numbers for each problem type. Collectively, the dataset comprises 105,600 examples across the four representations, which are further stratified into *easy* (GT-E) and *hard* (GT-H) subsets to accommodate varying levels of difficulty, where details can be found in Appendix A.

The generation of the entire dataset, including graph structures, problem formulations, and solution verifications, is fully automated through algorithmic procedures. This generation algorithm also supports adjusting graph scales to produce new data instances; the evaluated size ranges and scope are detailed in subsection B.4. Ground truth solutions are systematically derived using canonical algorithms specific to each task, thereby ensuring correctness and eliminating the need for manual annotation or labeling. Both the GT Bench dataset and the configurable generation pipeline have been made open-source.

4 Benchmarking LLMs on GT Bench

We evaluate eight LLMs on GT Bench: three proprietary models, `GPT-4o-mini` (OpenAI, 2024b), `GPT-4o` (OpenAI, 2024a), and `o3-mini` (OpenAI, 2025), and five open-source models, `Llama-3.1-8B` (Meta, 2024b), `Llama-3.3-70B` (Meta, 2024a), `Phi-4` (Abdin et al., 2024), `QwQ-32B` (Qwen Team, 2025), and `DeepSeek-R1` (Guo et al., 2025a). Each model is evaluated on 10,000 instances across the four representations, using one inference run per configuration at temperature 0.01. Model versions and settings appear in Appendix B, with evaluation scope and statistical interpretation in subsection B.4.

4.1 Representation Sensitivity

Different models exhibit distinct preferences for input representations. As shown in Table 4, no single input format is universally optimal. For example, `llama-3.1-8B` favors AM (29.4%), while `phi-4` performs best with SL (44.9%). Larger models like `gpt-4o` achieve peak accuracy with AL (47.6%), whereas `deepseek-r1` prefers AM (82.5%). The preference is model-specific, so any evaluation or deployment that fixes one format across models will flatter some and shortchange others.

Table 2: Overview of the 24 problem types in GT Bench. Example counts include the four input representations; sparse/dense variants and tree tasks together form 44 task–structure settings.

Task	Description	Nodes	Examples
Connectivity	Determine whether two vertices are connected	15–40	4800
Bipartiteness Check	Determine whether the graph is bipartite	15–40	4800
Minimum Cycle Length	Find the length of the smallest cycle in the graph	15–40	4800
Maximum Clique Size	Compute the size of the largest clique	15–40	4800
Maximum Independent Set	Compute the size of the largest independent set	15–40	4800
Eulerian Path	Determine whether the graph contains an Eulerian path	15–40	4800
Eulerian Circuit	Determine whether the graph contains an Eulerian circuit	15–40	4800
Hamiltonian Path	Determine whether the graph contains a Hamiltonian path	15–40	4800
Hamiltonian Circuit	Determine whether the graph contains a Hamiltonian circuit	15–40	4800
Biconnected Components	Count the number of biconnected components	11–20	4800
Bridge Count	Count the number of bridges (cut edges)	11–20	4800
Triangle Count	Count the number of triangles (3-node cycles)	11–13	4800
Cycle Count	Count the total number of cycles in the graph	6–10	4800
Spanning Tree Count	Count the number of possible spanning trees	6–10	4800
Shortest Path Length	Compute the shortest path between two nodes	14–20	4800
Minimum Spanning Tree	Compute the total weight of the minimum spanning tree	14–20	4800
Second Minimum Spanning Tree	Compute the weight of the second-best minimum spanning tree	11–20	4800
Tree Diameter	Compute the diameter (longest shortest path) of the tree	29–31	2400
Tree Centroid	Find the centroid node with the smallest index	29–60	2400
Lowest Common Ancestor	Find the lowest common ancestor (root at node 1)	29–60	2400
Tree Max Independent Set	Compute the size of the maximum independent set in a tree	29–60	2400
Maximum Flow	Compute the maximum flow from source to sink	11–20	4800
Minimum Cut	Compute the capacity of the minimum cut (source to sink)	11–20	4800
Min-Cost Max-Flow	Compute the minimum-cost maximum flow	9–15	4800

Table 3: Accuracy (%) on GT Bench across selected tasks and input representations, averaged across models. Average and Range summarize the four representations and are computed before rounding.

Task	Graph Type	NL	SL	AM	AL	Average	Range
Bipartite Check	Sparse	**89.9**	83.6	75.9	77.0	81.6	14.0
Tree Centroid	Tree	56.3	**60.8**	42.6	51.1	52.7	18.3
Maximum Clique	Dense	24.7	22.1	**38.6**	34.9	30.1	16.5
Hamiltonian Circuit	Dense	81.4	81.0	83.0	**88.3**	83.4	7.3

The optimal input representation is highly dependent on the specific task and graph structure. Table 3 illustrates this dependency with selected tasks from GT Bench. For instance, NL representation (89.9%) is most effective for Bipartite Check on sparse graphs, whereas AM (38.6%) is superior for Maximum Clique on dense graphs. Tree-centric tasks like Tree Centroid benefit from SL (60.8%), and path-oriented problems such as Hamiltonian Circuit on dense graphs achieve highest accuracy with AL (88.3%). Task-level results for these examples appear in Appendix C and form the empirical basis of GTA's heuristic selector.

Table 4: Accuracy (%) of different models on GT Bench under four input representations.

Model	NL	SL	AM	AL
Llama-3.1-8B	29.0	27.4	**29.4**	28.5
Phi-4	43.7	**44.9**	38.8	42.7
GPT-4o-mini	36.2	33.4	37.8	**40.7**
Llama-3.3-70B	**44.9**	41.1	40.7	44.1
GPT-4o	45.5	43.7	45.3	**47.6**
QwQ-32B	63.1	63.1	62.1	**71.3**
DeepSeek-R1	80.9	80.6	**82.5**	80.9
o3-mini	87.4	89.9	89.0	**90.2**

Which representation wins is not arbitrary. Table 19 in the appendix tabulates the empirically best format for every task–graph-type combination, and four regularities emerge:

NL suits tasks on sparse graphs, where little structural parsing is required and the descriptive prose plays to the model's linguistic strengths.

SL dominates on trees; its templated, indented format mirrors the hierarchy that tree algorithms traverse.

AM wins on dense graphs, where the matrix lays out every potential pairwise connection at once, making global properties easier to read off.

AL is strongest for pathfinding and neighbor-iteration tasks at any density, since it lists exactly the edges an algorithm would walk.

Representation sensitivity persists in the strongest evaluated reasoning models, as shown in Table 4. The gap between a model's best and worst format is 2.0 points for `Llama-3.1-8B`, which sits near floor under every format, widens to 6.1 for `Phi-4`, 7.3 for `GPT-4o-mini`, and 9.2 for `QwQ-32B`, and only then narrows to 2.8 for `o3-mini` and 1.9 for `DeepSeek-R1`. Sensitivity is thus largest in the capable-but-unsaturated regime where benchmark performance is most informative, and it shrinks, but does not vanish, for the strongest models. Appendix D complements these aggregates with paired case studies in which the same model, given the identical instance, fails under one representation and succeeds under another.

4.2 Performance across Difficulty Levels

Aggregate accuracy hides how sharply GT Bench separates models once difficulty is taken into account. Instances are stratified as follows: four intrinsically demanding problem types (Min-Cost Max-Flow, Cycle Count, Spanning Tree Count, and Biconnected Components) are always Hard, two elementary ones (Connectivity and Bipartiteness) are always Easy, and for every other task, instances on sparse graphs or trees count as Easy while instances on dense graphs count as Hard. Table 5 reports per-model accuracy under this split.

Two aspects of the split stand out. All eight models drop sharply from Easy to Hard; `QwQ-32B`, for example, falls by 47 points, from 85.62% to 38.46%. The model ordering also changes: `GPT-4o` edges out `Llama-3.3-70B` on Easy instances (60.54% versus 58.19%) yet trails it on Hard ones (28.94% versus 30.40%), so rankings

Table 5: Accuracy (%) of different LLMs on the Easy and Hard subsets of GT Bench. Easy and Hard subsets are defined based on task type and graph structure as detailed in subsection 4.2.

Model	Easy Task Accuracy (%)	Hard Task Accuracy (%)
`Llama-3.1-8B`	38.13	19.05
`GPT-4o-mini`	44.15	27.47
`Phi-4`	53.51	32.97
`Llama-3.3-70B`	58.19	30.40
`GPT-4o`	60.54	28.94
`QwQ-32B`	85.62	38.46
`DeepSeek-r1`	96.32	64.10
`o3-mini`	98.66	75.09

drawn from easy tasks alone can mislead. Even the strongest reasoning models remain well below ceiling on the Hard split, and this is the headroom that the methods evaluated in section 6 target.

4.3 Does Providing Every Representation Help?

To test whether providing all four representations simultaneously improves performance, we concatenated all formats into a single prompt. As Table 6 shows, this consistently degrades accuracy relative to the best single format for each model; `GPT-4o`, for instance, drops from 47.6% to 40.3%. Two effects compound. Concatenation inflates the prompt roughly three to four times, which risks context-window overflow on larger graphs, and the structural diversity of four simultaneous formats introduces noise that the model must reconcile rather than exploit. Adaptive selection of a single, well-matched representation is therefore preferable to naively supplying every available encoding.

Table 6: Accuracy (%) under individual representations vs. providing all four representations simultaneously. Best single-representation result for each model is **bolded**.

Model	NL	SL	AM	AL	All
Llama-3.1-8B	29.0	27.4	**29.4**	28.5	22.4
Phi-4	43.7	**44.9**	38.8	42.7	34.6
GPT-4o	45.5	43.7	45.3	**47.6**	40.3

4.4 Relationship to General Capabilities

To understand whether graph-theoretic reasoning tests capabilities distinct from those measured by existing benchmarks, we compare model performance on GT Bench (Hard subset) with their Elo ratings on Text Arena[1] across four capability dimensions: Coding, MATH, Instruction Following, and Longer Query (Table 7). Three patterns emerge. First, graph reasoning diverges sharply from general language capabilities. The most telling example is `QwQ-32B`: despite having the *lowest* scores in Instruction Following (1149) and Longer Query (1176), even lower than the 8B-parameter `Llama-3.1-8B`, it ranks 3rd on GT-H, substantially outperforming the much larger `GPT-4o` and `Llama-3.3-70B`. Second, the top performers on GT-H (`o3-mini`, `DeepSeek-R1`, `QwQ-32B`) are consistently the leaders in Coding and MATH, suggesting that graph-theoretic reasoning is rooted in precise, multi-step algorithmic execution, a capability shared with formal reasoning but distinct from natural language understanding. Third, model scale is not the primary driver of success: smaller, reasoning-optimized models consistently outperform larger generalist models. GT Bench thus probes an axis of capability that general leaderboards do not capture. That this axis is both distinctive and transferable is corroborated by Zhang et al. (2025c), who find that continued pretraining on graph problems improves mathematical and logical reasoning downstream.

Across these analyses a consistent picture emerges: accuracy swings with the input representation by up to nine points per model and far more on individual tasks; the winning format depends jointly on task, graph structure, and model; supplying every format at once backfires; and the Hard split remains far from saturated even for strong reasoning models. A remedy should therefore choose a suitable representation per instance

[1] https://lmarena.ai/leaderboard/text

Table 7: GT Bench (Hard) accuracy vs. Text Arena Elo ratings across capability dimensions. Models sorted by GT-H performance.

Model	GT-H	Coding	MATH	Instr. Follow	Longer Query
Llama-3.1-8B	19.1%	1258	1194	1187	1220
GPT-4o-mini	27.5%	1347	1279	1288	1322
GPT-4o	28.9%	1367	1308	1319	1328
Llama-3.3-70B	30.4%	1343	1299	1287	1309
Phi-4	33.0%	1304	1268	1239	1263
QwQ-32B	38.5%	1385	1366	1149	1176
DeepSeek-R1	64.1%	1443	1398	1390	1396
o3-mini	75.1%	1413	1387	1339	1358

rather than fix one globally, should support the executor through the multi-step algorithmic work on which unaided models fail, and should do both without retraining the executor, so that improvements reflect better problem presentation rather than added capacity. The agent introduced next is built around exactly these requirements.

5 Graph Theory Agent Framework

The **Graph Theory Agent (GTA)** turns the diagnosis of section 4 into a design. It combines two mechanisms: adaptive selection of the input representation for each instance, targeting the sensitivity documented in subsection 4.1, and systematic decomposition of the solution process, targeting the multi-step failures behind the Easy–Hard gap. Crucially, GTA operates under a *frozen executor* paradigm: the base LLM is never trained or fine-tuned on GT Bench data. Only lightweight auxiliary modules (the Selector and Decomposer) are optimized, ensuring that performance gains are attributable to the framework's reasoning strategy rather than changes in the executor's intrinsic capabilities. This distinguishes GTA from fine-tuning approaches such as GraphWiz, which modify the executor itself. GTA tackles a graph problem instance $P = (G, T)$ (where G is the graph and T the task) via a structured pipeline:

$$P \xrightarrow{\text{Selector}} P' \xrightarrow{\text{Generator}} \mathcal{A} \xrightarrow{\text{Decomposer}} \{s_1, ..., s_k\} \xrightarrow{\text{Executor}} \text{Solution} \tag{1}$$

The pipeline begins with the **Input Representation Selector** choosing an optimal format R^* for graph G and task T, yielding $P' = (G_{R^*}, T)$. The **Algorithm Generator** then formulates a high-level strategy $\mathcal{A}$, which the **Algorithm Decomposer** refines into a sequence of manageable sub-steps $\{s_1, ..., s_k\}$. Finally, an **Executor** LLM works through these sub-steps to compute the solution.

5.1 Input Representation Selector

The Input Representation Selector is the first component in the GTA pipeline. Its role is to choose the most effective format from the available options $\mathcal{R} = \{\text{NL, SL, AM, AL}\}$ to present the input graph G for a given task T. The objective is to select the representation R^* that maximizes the likelihood of successful downstream reasoning by the Executor LLM. This selection can be formalized as $\mathcal{S} : (G, T) \mapsto R^*$, where $R^* \in \mathcal{R}$ is determined based on features of G (e.g., size, density) and the requirements of T. We explore two implementations for this adaptive selector:

1. Heuristic-Based Selector: This approach employs a rule-based function derived from our empirical findings (Table 19). It maps observable graph and task characteristics to the representation that demonstrated the strongest average performance. Its primary advantage is computational efficiency.

2. Learned Selector: Alternatively, this selector is a dedicated language model, fine-tuned for representation selection. This allows optimization for a *specific* downstream Executor LLM, trained to predict the representation most likely to yield successful task completion by that target model using historical data of problem instances, representations, and outcomes. The learned selector's advantage over the heuristic is

most pronounced in two scenarios: (i) when the executor LLM deviates from general statistical trends (for example, reasoning-specialized models like `QwQ-32B` may exhibit different parsing preferences than generalist models), and (ii) for graphs in ambiguous structural regimes (e.g., medium density with complex community structures), where the optimal representation depends on finer-grained correlations between task requirements and the executor's specific capabilities. Henceforth, unless otherwise specified, references to GTA will imply the use of the Learned Selector.

By tailoring the input format per instance, the Selector confronts the representation sensitivity documented in section 4 and hands the downstream components a graph encoding the executor is well placed to reason over.

The Learned Selector is also the component that carries GTA's generalization, along three axes. Along the *model* axis, a single Selector trained on the Phi-4 pipeline transfers to unseen Executors without retraining (Table 14). Along the *task* axis, it extends to new problem families by fine-tuning on a small set of (task, graph, representation, outcome) preference pairs through the same procedure of subsection 5.4. Along the *representation* axis, the candidate set $\mathcal{R}$ can be enlarged with additional formats without any architectural change, since the Selector consumes only (G, T) features and returns a choice from $\mathcal{R}$. The static Heuristic-Based Selector, by contrast, is a lookup derived from GT Bench statistics and is best viewed as an economical fallback that would need re-derivation outside its coverage.

5.2 Algorithm Generator

Following the selection of an optimal representation R^*, the **Algorithm Generator** formulates a *high-level* strategic plan $\mathcal{A}$. This component, an off-the-shelf LLM, is prompted with the task description T and the graph G_{R^*} in its chosen format. Its role is conceptual: to establish the overarching algorithmic approach and outline key reasoning phases. It explicitly avoids generating detailed instructions or executable code. The resultant plan $\mathcal{A}$ serves as a strategic blueprint, ensuring a logically sound methodology and providing high-level guidance for the subsequent Algorithm Decomposer. This component operates without task-specific fine-tuning for its abstract plan generation stage.

5.3 Algorithm Decomposer

Bridging the gap between the high-level plan $\mathcal{A}$ generated previously and the final execution phase, the **Algorithm Decomposer** refines the abstract strategy into a concise sequence of concrete, manageable sub-steps $\{s_1, s_2, ..., s_k\}$. These sub-steps are specifically crafted for sequential processing by the LLM Executor. Breaking the problem down in this way reduces the load on the Executor, walking its attention incrementally through the algorithmic logic. The transformation performed by the decomposer can be represented as:

$$\text{Decomposer} : (\mathcal{A}, G_{R^*}) \mapsto \{s_1, s_2, ..., s_k\} \tag{2}$$

Each resulting sub-step s_i constitutes a focused natural language instruction that directs the Executor to perform a specific part of the overall algorithm. To effectively translate conceptual plans into actionable steps, the Algorithm Decomposer is implemented as a dedicated LLM. The specific fine-tuning process designed to optimize its decomposition capabilities is detailed in subsection 5.4.

5.4 Training and Execution Workflow

The GTA framework's components are trained sequentially. The **Heuristic-Based Selector** is a parameter-free function based on empirical results, and the **Algorithm Generator** employs a base LLM without specific fine-tuning for its role. The core training focuses on the **Algorithm Decomposer** and subsequently the **Learned Selector**.

Algorithm Decomposer Training: The Algorithm Decomposer LLM is trained in two stages, using the Heuristic-Based Selector for input graph representation ($G_{R^*_{\text{heuristic}}}$). *1. Supervised Fine-Tuning (SFT):* To establish foundational decomposition skills, the Algorithm Decomposer LLM is initially fine-tuned on expert-like decompositions D_{expert} (obtained via distillation from a stronger model) for given plans $\mathcal{A}$ and

graphs $G_{R^*_{\text{heuristic}}}$. This stage minimizes the SFT loss:

$$\mathcal{L}_{\text{SFT-Decomp}} = -\mathbb{E}_{(\mathcal{A},G_{R^*_{\text{heuristic}}}),D_{\text{expert}}}\left[\log P_{\text{Decomp}}(D_{\text{expert}}|\mathcal{A}, G_{R^*_{\text{heuristic}}})\right], \tag{3}$$

where P_{Decomp} denotes the probability assigned by the Algorithm Decomposer LLM. *2. Direct Preference Optimization (DPO):* Following Rafailov et al. (2023), the SFT-tuned model is then further refined. Multiple decomposition candidates $\{D_j\}$ are sampled for each problem. After execution by the Executor, these are labeled as "winning" (D_w, leading to a correct solution) or "losing" (D_l, leading to an incorrect solution). DPO then optimizes the Algorithm Decomposer LLM using these preference pairs (D_w, D_l) to maximize the likelihood of generating effective decompositions for the Executor. This directly links the Decomposer's output to successful problem execution.

Learned Selector Training: With the Algorithm Decomposer trained and fixed, the Learned Selector LLM is trained. For each problem instance $P = (G, T)$, the complete GTA pipeline (using the Algorithm Generator, Algorithm Decomposer, and Executor) is executed with each of the four input representations $R \in \{\text{NL, SL, AM, AL}\}$. This yields preferred representations R_w (leading to correct solutions) and dispreferred ones R_l. The Learned Selector is then fine-tuned via DPO using these preference pairs (R_w, R_l) for each problem P. This optimizes the Selector to choose the representation that maximizes the end-to-end success probability of the GTA pipeline.

Execution Phase: During inference, the chosen Selector (either Heuristic-Based or Learned Selector) determines the input representation R^*. The Algorithm Generator formulates the high-level plan $\mathcal{A}$. The Algorithm Decomposer breaks $\mathcal{A}$ into sub-steps $\{s_i\}$, which are then processed by the base LLM acting as the Executor to derive the final solution.

6 Evaluating GTA

We evaluate GTA against prompting and agent baselines (subsection 6.2), isolate where its gains come from (subsection 6.3, subsection 6.4), and then examine the practical axes that matter for deployment: behavior on larger graphs, transfer to other executors, monetary cost, and failure modes.

6.1 Setup

Datasets. GTA and all baselines are compared on GT Bench and, to test transfer beyond the training distribution, on the Graph Understanding subset of GraCoRe (Yuan et al., 2025) and the complete NLGraph (Wang et al., 2023a) dataset. GraCoRe and NLGraph instances are randomly converted to one of the four input modalities used in GT Bench, and each method is evaluated on 1,000 instances per dataset.

Baselines. Across all datasets, we compare our proposed GTA against several established methods. These encompass prompting-based techniques: Vanilla prompting, Chain-of-Thought (CoT) (Wei et al., 2022), LLM-Debate (Du et al., 2024), and Self-Refine (Madaan et al., 2023); as well as automated agent frameworks: ADAS (Hu et al., 2025a) , AFlow (Zhang et al., 2025b) and MaAS (Zhang et al., 2025a). Throughout our experiments, "Vanilla prompting" refers to using the base LLM executor to solve graph problems directly in a zero-shot setting, without any component of the GTA agentic framework or additional prompting strategies. To ensure a fair comparison, all baseline methods also utilize the `Phi-4` model as their executor LLM. Importantly, under all configurations (including GTA), the executor LLM remains frozen and unmodified; GTA's auxiliary components learn to *format and structure* problems for the executor, not the task solutions themselves. This ensures a fair comparison: all methods use the same base model, and only the external reasoning strategy differs.

Implementation Details. GTA is trained solely on GT Bench and evaluated for generalization on GraCoRe and NLGraph. We use `Phi-4` for the Learned Selector, Algorithm Generator, Algorithm Decomposer, and Executor to balance performance and computational cost, and distill expert demonstrations for the Decomposer from `GPT-4o`. A consistent temperature of 0.01 is used for all model inferences to ensure stability. Further details of the experimental configuration are provided in Appendix B, and task-level representation results are provided in Appendix C.

6.2 Comparative Performance of GTA

We first compare GTA with the prompting strategies and automated agent frameworks introduced above. Table 8 reports average accuracy on GT Bench (Easy and Hard), GraCoRe, and NLGraph.

GTA achieves the highest accuracy on every benchmark, reaching 69.1% on GT-E and 41.5% on GT-H, ahead of the next best methods, MaAS (63.2%/37.6%) and AFlow (62.0%/37.2%). Notably, the search-based agent frameworks (ADAS, AFlow, MaAS) optimize the workflow around the same frozen executor, so the comparison isolates what is gained by making the input representation and the algorithmic breakdown explicit targets of optimization rather than searching over generic prompt-and-call structures.

Since GTA is trained solely on GT Bench, its accuracy on GraCoRe (80.5%) and NLGraph (89.4%), two benchmarks whose task distributions are not subsets of GT Bench, indicates that adaptive representation selection and structured decomposition transfer to related graph reasoning settings rather than overfitting the training benchmark.

Table 8: Average accuracy (%) of different methods on multiple graph reasoning benchmarks. GT-E = GT Bench (Easy), GT-H = GT Bench (Hard), GCR = GraCoRe, NLG = NLGraph.

Method	GT-E	GT-H	GCR	NLG
Vanilla	53.5	33.0	66.8	80.2
CoT	52.1	34.2	67.2	81.0
LLM-Debate	58.9	35.5	70.3	84.4
Self-Refine	56.8	34.6	68.0	83.1
GraphTeam	50.1	27.2	51.6	63.2
ADAS	52.8	32.7	67.5	80.5
AFlow	62.0	37.2	75.4	86.2
MaAS	63.2	37.6	78.8	86.0
GTA	**69.1**	**41.5**	**80.5**	**89.4**

6.3 Ablation Study of GTA Components

To verify the contribution of GTA's core modules, we compare the full system against a **Vanilla** baseline (Phi-4 with vanilla prompting) and several simplified variants: **GTA-NoSel**: GTA without any Input Representation Selector (fixed default representation). **GTA-NoDec**: GTA without the Algorithm Generator and Decomposer (direct execution). **GTA-HeuSel**: GTA using the Heuristic-Based Selector instead of the Learned Selector. Performance is reported in Table 9.

Table 9: Accuracy (%) on GT Bench, GraCoRe (GCR), and NLGraph (NLG). Configurations: Vanilla (Vanilla Phi-4), GTA-NoSel (w/o Selector), GTA-HeuSel (w/ Heuristic Selector), GTA-NoDec (w/o Generator & Decomposer), **GTA** (Our proposed method).

Config.	GT-E	GT-H	GCR	NLG
Vanilla	53.5	33.0	66.8	80.2
GTA-NoSel	58.2	36.1	71.6	83.5
GTA-NoDec	62.1	37.7	75.3	85.0
GTA-HeuSel	64.5	38.8	76.5	85.8
GTA	**69.1**	**41.5**	**80.5**	**89.4**

Both modules matter, and neither alone accounts for the gains. Removing representation selection is the single most damaging change: GT-E falls from 69.1% to 58.2%, confirming that the input format is not a cosmetic detail but a central factor in whether the executor succeeds. Replacing the Learned Selector with the heuristic lookup costs 4.6 points on GT-E (64.5% versus 69.1%), which quantifies how much executor- and task-specific signal the learned policy captures beyond coarse empirical rules. Removing planning and decomposition costs 7.0 points (62.1% versus 69.1%), showing that a structured multi-step breakdown helps the executor well beyond direct monolithic solving. The same ordering of configurations holds on GT-H, GraCoRe, and NLGraph, so the two mechanisms contribute complementary improvements that are not tied to a single dataset.

Per-component ablation: Generator versus Decomposer. The plan–decompose module instantiates the plan–decompose–execute paradigm widely adopted in prior agent work (Wang et al., 2023b; Zhou et al., 2023; Khot et al., 2023; Shen et al., 2023), in which a strategic Generator and a step-level Decomposer play distinct but complementary roles. To isolate their individual contributions, we evaluate two finer-grained

variants, averaged over three runs: **GTA-NoDecomposer** keeps the Generator but removes the Decomposer, and **GTA-NoGen** keeps the Decomposer but removes the Generator (Table 10).

Table 10: Per-component ablation of the plan–decompose module (mean ± std over 3 runs). GTA-NoDec removes both modules; GTA-NoDecomposer keeps only the Generator; GTA-NoGen keeps only the Decomposer.

Configuration	GT-E (%)	GT-H (%)	GCR (%)	NLG (%)
GTA-NoDec (Selector only)	62.1 ± 0.5	37.7 ± 0.4	75.3 ± 0.6	85.0 ± 0.4
GTA-NoDecomposer (Generator only)	64.3 ± 0.6	38.5 ± 0.5	76.8 ± 0.5	86.2 ± 0.5
GTA-NoGen (Decomposer only)	66.8 ± 0.5	39.9 ± 0.4	78.1 ± 0.5	87.5 ± 0.4
GTA (full)	$\mathbf{69.1 \pm 0.4}$	$\mathbf{41.5 \pm 0.5}$	$\mathbf{80.5 \pm 0.3}$	$\mathbf{89.4 \pm 0.4}$

Removing the Decomposer (a 4.8-point drop on GT-E) hurts more than removing the Generator (a 2.3-point drop), consistent with the Decomposer supplying the actionable step-level signal that the Executor consumes while the Generator contributes the complementary high-level algorithmic frame. The joint removal (7.0 points below full GTA) is close to the sum of the two independent removals (7.1 points), indicating that the modules play distinct, near-additive roles rather than redundant ones.

Selector training for direct execution. GTA-NoDec reuses the Learned Selector trained with the full pipeline. To measure the effect of this training configuration, we retrain a Selector specifically for the direct Phi-4 Executor, with no Generator or Decomposer (Table 11). The comparison evaluates how well the Selector's $(G, T) \mapsto R^*$ mapping transfers from the full pipeline to direct execution.

Table 11: A Selector retrained specifically for the direct Executor (Selector-Direct) versus GTA-NoDec, which reuses the full-pipeline Selector (mean ± std over 3 runs).

Configuration	GT-E (%)	GT-H (%)	GCR (%)	NLG (%)
GTA-NoDec (Selector from full pipeline)	62.1 ± 0.5	37.7 ± 0.4	75.3 ± 0.6	85.0 ± 0.4
Selector-Direct (Selector retrained for direct Executor)	63.8 ± 0.6	38.4 ± 0.5	76.2 ± 0.5	85.6 ± 0.5
GTA (full)	$\mathbf{69.1 \pm 0.4}$	$\mathbf{41.5 \pm 0.5}$	$\mathbf{80.5 \pm 0.3}$	$\mathbf{89.4 \pm 0.4}$

Retraining the Selector for the direct Executor improves accuracy by 1.7 points on GT-E and 0.7 on GT-H over GTA-NoDec. Full GTA still outperforms the retrained Selector-only system by 5.3 points on GT-E and 3.1 on GT-H. Representation selection thus transfers across execution configurations, while the plan–decompose module provides additional gains beyond a Selector optimized for direct execution.

Reliability of the training-free Generator. Because the Algorithm Generator is a prompted, off-the-shelf LLM rather than a fine-tuned module, we measured its stability on a 1,000-instance subset spanning all 44 task–structure settings, averaged over three runs. Plan self-consistency, measured by Jaccard overlap over the enumerated reasoning phases at temperature 0.7, is $94.7 \pm 1.2\%$, and algorithmic alignment with the canonical reference algorithms in Appendix A, judged by GPT-4o, is $91.5 \pm 1.8\%$. The GPT-4o alignment score is an auxiliary judge-based diagnostic; LLM-judge bias has been studied mechanistically by Xu et al. (2026d). Other evaluations document systematic judgment biases (Ye et al., 2025), preference leakage when generators and judges are related (Li et al., 2026a), and susceptibility to optimized prompt injection (Shi et al., 2024). GT Bench answer accuracy is instead evaluated against algorithmically verified ground truth. These diagnostics indicate that the Generator produces stable, well-aligned high-level plans for these classical graph problems, which justifies keeping the Generator training-free and concentrating optimization on the Decomposer and the Selector.

6.4 SFT–DPO Training Dynamics of the Learned Components

We employ a two-stage pipeline for the Algorithm Decomposer and the Learned Selector: (i) **SFT** on expert demonstrations to establish a broad, modality-agnostic foundation, followed by (ii) **DPO** to directly optimize for end-to-end task success beyond imitation-style learning.

Selector choice during Decomposer training. We compare DPO training with the Heuristic-Based Selector against a *Random Selector* that samples all input modalities uniformly. This comparison measures the effect of the training-time representation policy on the resulting Decomposer. Both configurations improve over the Vanilla baseline, with the heuristic configuration achieving higher mean accuracy across the four evaluation sets:

Table 12: Effect of selector choice during DPO on Decomposer performance (mean ± std).

Decomposer trained with	**GT-E (%)**	**GT-H (%)**	**GCR (%)**	**NLG (%)**
Vanilla Baseline	53.5 ± 0.2	33.0 ± 0.3	66.8 ± 0.1	80.2 ± 0.2
Heuristic Selector (DPO)	$\mathbf{69.1 \pm 0.4}$	$\mathbf{41.5 \pm 0.5}$	$\mathbf{80.5 \pm 0.3}$	$\mathbf{89.4 \pm 0.4}$
Random Selector (DPO)	68.7 ± 0.7	40.9 ± 0.6	79.8 ± 0.9	87.0 ± 0.5

Module-specific DPO gains. We apply DPO to two modules:

1) *Learned Selector.* DPO enables the selector to prefer the input representation that most likely yields a correct answer. As shown in Table 9, the full GTA (with the Learned Selector) substantially outperforms the heuristic variant (GTA-HeuSel), confirming the value of DPO-guided representation selection.

2) *Algorithm Decomposer.* To quantify the decomposer-side gains, we compare SFT-only versus SFT+DPO. Averaged over 3 runs:

Table 13: Decomposer ablation: SFT-only vs. SFT+DPO (mean ± std).

Config. (Decomposer trained with...)	**GT-E (%)**	**GT-H (%)**	**GCR (%)**	**NLG (%)**
SFT Only	65.1 ± 1.1	37.2 ± 0.8	71.5 ± 0.9	80.3 ± 0.7
SFT + DPO (Ours)	$\mathbf{69.1 \pm 0.4}$	$\mathbf{41.5 \pm 0.5}$	$\mathbf{80.5 \pm 0.3}$	$\mathbf{89.4 \pm 0.4}$

The SFT stage supplies demonstrations across tasks and modalities, while DPO optimizes for task success. DPO-trained Decomposers improve over the Vanilla baseline under both training-time selector policies (Table 12), and SFT+DPO outperforms SFT-only training (Table 13). These results complement the Learned Selector ablation (Table 9) and identify the contributions of the two trained components to GTA's end-to-end performance.

6.5 Scalability Analysis with Increasing Graph Size

A key advantage of our automated generation process for GT Bench is the flexibility to adjust graph size (number of nodes and edges), enabling the creation of customized datasets to study LLM scalability on graph reasoning tasks. To investigate the impact of graph scale on performance, we generated variants of GT Bench tasks with increasing numbers of nodes. Figure 2 illustrates the performance trends of selected models (`Phi-4`, `GPT-4o-mini`, `QwQ-32B`) and our GTA framework (using `Phi-4` as the base) as graph size increases, separately for Easy (GT-E) and Hard (GT-H) subsets.

On the GT-E subset, the accuracy of the smaller base models, `Phi-4` and `GPT-4o-mini`, declines steadily as the node count grows, while GTA stays comparatively flat across the tested sizes. For these simpler tasks, adaptive representation plus decomposition absorbs most of the difficulty that scale adds.

The picture changes on GT-H. As graphs grow, `Phi-4` and `GPT-4o-mini` fall off sharply, and GTA declines as well: its drop is steeper than that of a strong standalone reasoner such as `QwQ-32B`, but far gentler than that of its own base model, which it outperforms at every size tested. The framework mitigates, rather than eliminates, the difficulty of scaling hard combinatorial tasks; on the largest hard instances the executor's intrinsic capability reasserts itself as the binding constraint.

In short, GTA does not remove the difficulty of large hard instances, but for the same base model it consistently turns a steep degradation curve into a gentler one.

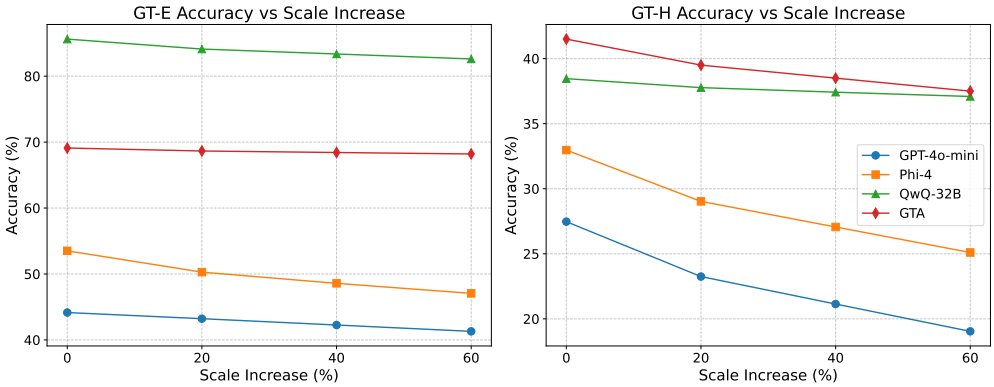

Figure 2: Performance comparison of different models (Phi-4, GPT-4o-mini, QwQ-32B) and the GTA framework on GT Bench Easy (GT-E) and Hard (GT-H) subsets as the number of graph scale increases.

6.6 Generality across Executor Models

While the core GTA components (Learned Selector, Algorithm Decomposer) were fine-tuned using `Phi-4`, it is important to understand how the framework performs when a different LLM acts as the final Executor. To investigate this, we evaluated the performance of GTA using `GPT-4o-mini`, `GPT-4o`, and `DeepSeek-R1` as executors, comparing their performance within the GTA framework against their respective vanilla (direct prompting) results. The outcomes are presented in Table 14.

Table 14: Accuracy (%) comparison of different models with Vanilla prompting vs. integrated into the GTA framework as the Executor. GTA components (Selector, Decomposer) were trained using Phi-4.

Model Configuration	**GT-E (%)**	**GT-H (%)**	**GCR (%)**	**NLG (%)**
GPT-4o-mini (Vanilla)	44.2	27.5	56.2	70.3
GPT-4o-mini (GTA Executor)	**53.1**	**34.2**	**63.5**	**76.6**
Phi-4 (Vanilla)	53.5	33.0	66.8	80.2
Phi-4 (GTA Executor)	**69.1**	**41.5**	**80.5**	**89.4**
GPT-4o (Vanilla)	60.5	28.9	70.2	81.4
GPT-4o (GTA Executor)	**68.8**	**36.9**	**78.8**	**88.2**
DeepSeek-R1 (Vanilla)	96.3	64.1	97.6	99.0
DeepSeek-R1 (GTA Executor)	96.4	**72.2**	97.6	98.8

GTA transfers across executors despite its components having been tuned with `Phi-4`. `GPT-4o-mini` and `GPT-4o` gain between 6 and 9 points on every benchmark; `GPT-4o`, for instance, rises from 60.5% to 68.8% on GT-E and from 28.9% to 36.9% on GT-H. The `DeepSeek-R1` row shows how these gains behave at the top of the capability range: where the executor is already near ceiling (GT-E, GraCoRe, NLGraph), GTA neither helps nor hurts, with differences within noise, while on the unsaturated GT-H split it still adds 8.1 points (64.1% to 72.2%). The framework's benefit thus concentrates where headroom remains, and the scaffolding does not degrade executors that no longer need it.

6.7 Inference Cost Analysis

Beyond accuracy, inference cost decides whether a reasoning framework is practical to deploy. To compare the efficiency of GTA with the baselines, we measured token consumption and calculated the associated costs when running each method on a representative sample of 600 instances drawn from the GT Bench dataset (combining both Easy and Hard tasks). For all methods, `Phi-4` was used as the base LLM to ensure a fair comparison of the overhead introduced by each framework. Costs are calculated from the reported token

counts at rates of $0.07 per million input tokens and $0.14 per million output tokens. Table 15 summarizes these findings.

Table 15: Inference Cost and Token Consumption (in Millions) for running 600 instances (mixed Easy/Hard) from GT Bench using Phi-4 as the base LLM. Pricing: $0.07/M input, $0.14/M output.

Method	Input Tokens (M)	Output Tokens (M)	Calculated Cost (USD)
Phi-4 (Vanilla)	1.75	0.53	$0.20
LLM-Debate	64.51	19.81	$7.29
Self-Refine	34.87	10.71	$3.94
AFlow	17.44	5.65	$2.01
MaAS	8.11	2.49	$0.92
GTA (ours)	9.85	3.02	$1.11

Cost differences across methods are large. Multi-round methods are the most expensive: LLM-Debate consumes over thirty times the tokens of vanilla prompting, and Self-Refine and AFlow also add substantial overhead. GTA's overhead is instead bounded by construction. Each instance passes through the Selector, Generator, Decomposer, and Executor once, so token usage grows by a small constant factor over vanilla prompting (roughly 5–6× here) rather than with the number of interaction rounds. GTA therefore delivers the highest accuracy in Table 8 at $1.11 per 600 instances, below every method in Table 15 except MaAS and vanilla prompting itself.

One-time training and data-generation cost. The inference figures above exclude the one-time overhead of building GTA, which we itemize in Table 16. Distilling decomposition demonstrations from `GPT-4o` for SFT consumes roughly 5M tokens (about $28), and DPO fine-tuning together with Selector preference-pair generation runs for 40 hours on two A6000 GPUs (about $64), for a total of roughly $92.

Table 16: One-time training and data-generation cost of GTA. This overhead is incurred once and amortizes across all subsequent inferences.

Component	Quantity	Subtotal (USD)
GPT-4o distillation for SFT (1,000 instances)	∼5M tokens (3M in / 2M out)	∼$28
DPO fine-tuning + Selector preference-pair generation (2×A6000, 40h)	80 GPU-hours	∼$64
Total one-time cost		**≈$92**

The $92 one-time cost adds $0.0092 per instance when spread over 10,000 inferences and $0.00092 per instance over 100,000 inferences. For comparison, the token counts in Table 15 yield an inference cost of approximately $0.00185 per instance, or $1.85 per 1,000 instances. Total cost therefore depends on evaluation volume as well as inference token consumption.

6.8 Failure-Mode and Truncation Analysis

We measure context overflow, output truncation, and answer-extraction failures to assess their contribution to the reported error rates. All evaluations use a maximum generation length of 8,192 tokens, consistent across models. Table 17 reports these statistics across all 10,000 evaluation instances, averaged over three runs.

Table 17: Input-overflow, output-truncation, and answer-extraction rates across all 10,000 evaluation instances (mean ± std over 3 runs). Input overflow is eliminated by design through the prompt-fit budget of GT Bench.

Metric	Overall	GT-E (Easy)	GT-H (Hard)
Input exceeds context window	0.0%	0.0%	0.0%
Output truncated at max length	$1.2 \pm 0.3\%$	$0.5 \pm 0.2\%$	$2.1 \pm 0.4\%$
Answer extraction failure	$0.8 \pm 0.2\%$	$0.5 \pm 0.1\%$	$1.2 \pm 0.3\%$

The 0.0% input-overflow rate is by design: the prompt-fit budget of section 3 calibrates node ranges so that all four representations fit within typical context windows, with tighter ranges for dense graphs where the edge count grows as $O(V^2)$. Output truncation is low (1.2% overall, 2.1% on Hard) and cannot account for the observed accuracy gaps: the gap between GTA-NoDec and full GTA alone is 7.0 points on GT-E and 3.8 on GT-H, far larger than the 1–2% truncation rate. Answer-extraction failures are similarly rare (0.8% overall). The performance differences we report therefore reflect reasoning ability rather than formatting or truncation artifacts.

7 Conclusion

We introduced GT Bench, a benchmark of 24 graph problems in 44 task–structure settings with over 100,000 examples across four input representations, and used it to characterize how LLMs reason over graphs in language. The evaluation yields a consistent picture: accuracy depends on the input representation in ways that track task, graph structure, and model; the sensitivity persists, attenuated, in the strongest reasoning models; and hard multi-step tasks remain far from solved by scale alone. The Graph Theory Agent turns these observations into a method. A preference-trained Selector picks the representation per instance and a Generator–Decomposer pair converts the task into steps a frozen executor can follow, yielding consistent gains over prompting and agent baselines at bounded cost, transfer to benchmarks and executors unseen in training, and a gentler degradation curve as graphs grow (subsection 6.5). Together, GT Bench and GTA provide a benchmark and an adaptive reasoning framework for measuring and improving language-based algorithmic graph reasoning.

References

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 Technical Report, 2024. URL https://arxiv.org/abs/2412.08905. arXiv preprint arXiv:2412.08905.

Naaisha Agarwal, Yihan Wu, Yichang Jian, Yikuan Hu, Nishad Mansoor, Mohan Li, Yifei Peng, Wang-Zhou Dai, Yao-Xiang Ding, and Emanuele Sansone. OrigamiBench: An Interactive Environment to Synthesize Flat-Foldable Origamis. *arXiv preprint arXiv:2603.13856*, 2026. URL https://arxiv.org/abs/2603.13856.

Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiao Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. GraphLLM: Boosting Graph Reasoning Ability of Large Language Model, 2023. URL https://arxiv.org/abs/2310.05845.

Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. GraphWiz: An Instruction-Following Language Model for Graph Computational Problems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 353–364, 2024. doi: 10.1145/3637528.3672010. URL https://doi.org/10.1145/3637528.3672010.

Xiuying Chen, Tairan Wang, Juexiao Zhou, Zirui Song, Xin Gao, and Xiangliang Zhang. Evaluating and mitigating bias in AI-based medical text generation. *Nature Computational Science*, 5(5):388–396, 2025. doi: 10.1038/s43588-025-00789-7. URL https://doi.org/10.1038/s43588-025-00789-7.

Yiqun Chen, Jinyuan Feng, Wei Yang, Meizhi Zhong, Zhengliang Shi, Rui Li, Xiaochi Wei, Yan Gao, Yi Wu, Yao Hu, Zhiqiang Pu, and Jiaxin Mao. Self-Compression of Chain-of-Thought via Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2601.21919*, 2026a. URL https://arxiv.org/abs/2601.21919.

Yiqun Chen, Wei Yang, Erhan Zhang, Shijie Wang, Qi Liu, Zechun Niu, Bin Zhang, Haitao Li, Rui Li, Lingyong Yan, Jinyuan Feng, Biqing Qi, Xiaochi Wei, Yan Gao, Yi Wu, Yao Hu, and Jiaxin Mao. UnityMAS-O: A General RL Optimization Framework for LLM-Based Multi-Agent Systems. *arXiv preprint arXiv:2605.26646*, 2026b. URL https://arxiv.org/abs/2605.26646.

Xu Chu, Hanlin Xue, Zhijie Tan, Bingce Wang, Tong Mo, and Weiping Li. GraphSOS: Graph Sampling and Order Selection to Help LLMs Understand Graphs Better. *arXiv preprint arXiv:2501.14427*, 2025. URL https://arxiv.org/abs/2501.14427.

Shiyi Du, Jiayuan Liu, Weihua Du, Yue Huang, Jiayi Li, Yingtao Luo, Xiangliang Zhang, Vincent Conitzer, and Carl Kingsford. Why Search When You Can Transfer? Amortized Agentic Workflow Design from Structural Priors. *arXiv preprint arXiv:2604.25012*, 2026. URL https://arxiv.org/abs/2604.25012.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving Factuality and Reasoning in Language Models through Multiagent Debate. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 2024. URL https://proceedings.mlr.press/v235/du24e.html.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a Graph: Encoding Graphs for Large Language Models. In *International Conference on Learning Representations*, volume 2024, pp. 43909–43934, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/bf72f65f30eedf5d48da6980ee02b589-Paper-Conference.pdf.

Yifan Feng, Chengwu Yang, Xingliang Hou, Shaoyi Du, Shihui Ying, Zongze Wu, and Yue Gao. Beyond Graphs: Can Large Language Models Comprehend Hypergraphs? In *International Conference on Learning Representations*, pp. 42468–42495, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/690fc970014e4ecebc8068bbc03b35e6-Paper-Conference.pdf.

Hamed Firooz, Maziar Sanjabi, Wenlong Jiang, and Xiaoling Zhai. Lost-in-Distance: Impact of Contextual Proximity on LLM Performance in Graph Tasks. *arXiv preprint arXiv:2410.01985*, 2024. URL https://arxiv.org/abs/2410.01985.

Yuyao Ge, Shenghua Liu, Baolong Bi, Yiwei Wang, Lingrui Mei, Wenjie Feng, Lizhe Chen, and Xueqi Cheng. Can Graph Descriptive Order Affect Solving Graph Problems with LLMs? In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6404–6420. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.321. URL https://aclanthology.org/2025.acl-long.321/.

Florian Grötschla, Luis Müller, Jan Tönshoff, Mikhail Galkin, and Bryan Perozzi. AgentsNet: Coordination and Collaborative Reasoning in Multi-Agent LLMs. *arXiv preprint arXiv:2507.08616*, 2025. URL https://arxiv.org/abs/2507.08616.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025a. doi: 10.1038/s41586-025-09422-z. URL https://doi.org/10.1038/s41586-025-09422-z.

Jiayan Guo, Lun Du, Hengyu Liu, Mengyu Zhou, Xinyi He, and Shi Han. GPT4Graph: Can Large Language Models Understand Graph Structured Data? An Empirical Evaluation and Benchmarking. *arXiv preprint arXiv:2305.15066*, 2023. URL https://arxiv.org/abs/2305.15066.

Xiaojun Guo, Ang Li, Yifei Wang, Stefanie Jegelka, and Yisen Wang. G1: Teaching LLMs to Reason on Graphs with Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 52958–53001, 2025b. doi: 10.52202/085713-1767. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/4cd171bc72cdfec310eb6e9e78f7bd90-Paper-Conference.pdf.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 132876–132907, 2024. doi: 10.52202/079017-4224. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/efaf1c9726648c8ba363a5c927440529-Paper-Conference.pdf.

Daniel Herbst, Lea Karbevska, Divyanshu Kumar, Akanksha Ahuja, Fatemeh Gholamzadeh Nasrabadi, and Fabrizio Frasca. Lost in Serialization: Invariance and Generalization of LLM Graph Reasoners. *arXiv preprint arXiv:2511.10234*, 2025. URL https://arxiv.org/abs/2511.10234.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/36b7acf6f6010652b3f2a433774a66fe-Paper-Conference.pdf.

Yikuan Hu, Chen Huang, and Wenqiang Lei. ASTRO: Automatic Strategy Optimization For Non-Cooperative Dialogues. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 388–408, 2025b. doi: 10.18653/v1/2025.findings-acl.22. URL https://aclanthology.org/2025.findings-acl.22/.

Yikuan Hu, Jifeng Zhu, Lanrui Tang, and Chen Huang. ReMindRAG: Low-Cost LLM-Guided Knowledge Graph Traversal for Efficient RAG. In *Advances in Neural Information Processing Systems*, volume 38, pp. 53757–53798, 2025c. doi: 10.52202/085713-1793. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/4d880ba08f6b115bb8d685159bb167eb-Abstract-Conference.html.

Yuwei Hu, Xinyi Huang, Zhewei Wei, Yongchao Liu, and Chuntao Hong. Rethinking and Benchmarking Large Language Models for Graph Reasoning. *arXiv preprint arXiv:2509.24260*, 2025d. URL https://arxiv.org/abs/2509.24260.

Yuwei Hu, Runlin Lei, Xinyi Huang, Zhewei Wei, and Yongchao Liu. Scalable and Accurate Graph Reasoning with LLM-based Multi-Agents, 2025e. URL https://arxiv.org/abs/2410.05130.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. MetaTool Benchmark for Large Language Models: Deciding Whether to Use Tools and Which to Use. In *International Conference on Learning Representations*, 2024a. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/bc12914d66b41b6bfc2d3a5decdb498b-Abstract-Conference.html.

Yue Huang, Kai Shu, Philip S. Yu, and Lichao Sun. From Creation to Clarification: ChatGPT's Journey Through the Fake News Quagmire. In *Companion Proceedings of the ACM Web Conference 2024*, pp. 513–516, 2024b. doi: 10.1145/3589335.3651509. URL https://doi.org/10.1145/3589335.3651509.

Yue Huang, Siyuan Wu, Chujie Gao, Dongping Chen, Qihui Zhang, Yao Wan, Tianyi Zhou, Chaowei Xiao, Jianfeng Gao, Lichao Sun, and Xiangliang Zhang. DataGen: Unified Synthetic Dataset Generation via Large Language Models. In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a01e69aa9c3c61fcb40ea378e71fc780-Abstract-Conference.html.

Yue Huang, Zhengzhe Jiang, Yuchen Ma, Yu Jiang, Xiangqi Wang, Yujun Zhou, Yuexing Hao, Kehan Guo, Pin-Yu Chen, Stefan Feuerriegel, and Xiangliang Zhang. ProbeLLM: Automating Principled Diagnosis of LLM Failures. *arXiv preprint arXiv:2602.12966*, 2026a. URL https://arxiv.org/abs/2602.12966.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. MemoHarness: Agent Harnesses That Learn from Experience. *arXiv preprint arXiv:2607.14159*, 2026b. URL https://arxiv.org/abs/2607.14159.

Zixiao Huang, Lifeng Guo, Wenhao Li, Junjie Sheng, Chuyun Shen, Haosheng Chen, Bo Jin, Changhong Lu, and Xiangfeng Wang. GraphThought: Graph Combinatorial Optimization with Thought Generation, 2025b. URL https://arxiv.org/abs/2502.11607.

Xia Jiang, Yaoxin Wu, Minshuo Li, Zhiguang Cao, and Yingqian Zhang. Large Language Models as End-to-end Combinatorial Optimization Solvers. In *Advances in Neural Information Processing Systems*, volume 38, pp. 164787–164826, 2025. doi: 10.52202/085713-5495. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f0c6c93e1dbff84daf6082fef7e1a094-Paper-Conference.pdf.

Yu Jiang, Wenjie Wang, Yue Huang, Yanbo Wang, Zhenhong Zhou, Xiuying Chen, Yang Liu, Pin-Yu Chen, Wei Wang, and Xiangliang Zhang. RiskLab: A Controlled Toolkit for Probing Emergent Risks in LLM-Based Multi-Agent Systems. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 167–177, 2026. doi: 10.18653/v1/2026.acl-demo.17. URL https://aclanthology.org/2026.acl-demo.17/.

Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Zheng Li, Ruirui Li, Xianfeng Tang, Suhang Wang, Yu Meng, and Jiawei Han. Graph Chain-of-Thought: Augmenting Large Language Models by Reasoning on Graphs. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 163–184. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.findings-acl.11. URL https://aclanthology.org/2024.findings-acl.11/.

Hongbo Jin, Chi Wang, Haoran Tang, Zhongjing Du, Xu Jiang, Jingqi Tian, Qiaoman Zhang, and Jiayu Ding. ContextGuard: Structured Self-Auditing for Context Learning in Language Models. *arXiv preprint arXiv:2605.26827*, 2026a. URL https://arxiv.org/abs/2605.26827.

Hongbo Jin, Mingnan Zhu, Jingqi Tian, Xu Jiang, Zhongjing Du, Haoran Tang, Siyi Xie, Qiaoman Zhang, and Jiayu Ding. Context-CoT: Enhancing Context Learning via High-Quality Reasoning Synthesis. *arXiv preprint arXiv:2605.25354*, 2026b. URL https://arxiv.org/abs/2605.25354.

Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed Prompting: A Modular Approach for Solving Complex Tasks. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://iclr.cc/virtual/2023/poster/10871.

Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. Preference Leakage: A Contamination Problem in LLM-as-a-judge. In *International Conference on Learning Representations*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/6297baf3f5c98146b62dd3a1bffe068e-Abstract-Conference.html.

Jiate Li, Meng Pang, Yun Dong, Jinyuan Jia, and Binghui Wang. Graph Neural Network Explanations are Fragile. In *International Conference on Machine Learning*, pp. 28551–28567, 2024a. URL https://proceedings.mlr.press/v235/li24bd.html.

Jiate Li, Meng Pang, Yun Dong, Jinyuan Jia, and Binghui Wang. Provably Robust Explainable Graph Neural Networks against Graph Perturbation Attacks. In *International Conference on Learning Representations*, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e889993cf825668a2dcedcff6e97e0df-Abstract-Conference.html.

Shawn Li, Wei Yang, Jike Zhong, Jiate Li, Jiawei Yang, You Qin, Ryan Rossi, Franck Dernoncourt, Roger Zimmermann, Yue Wang, Zhengzhong Tu, Vicente Ordonez, Mohit Bansal, and Yue Zhao. JigShape: Evaluating Visual-Geometric Reasoning in VLMs through Jigsaw Puzzles. *arXiv preprint arXiv:2607.27670*, 2026b. URL https://arxiv.org/abs/2607.27670.

Xin Li, Weize Chen, Qizhi Chu, Haopeng Li, Zhaojun Sun, Ran Li, Chen Qian, Yiwei Wei, Zhiyuan Liu, Chuan Shi, Maosong Sun, and Cheng Yang. Can Large Language Models Analyze Graphs like Professionals? A Benchmark, Datasets and Models. In *Advances in Neural Information Processing Systems*, volume 37, pp. 141045–141070, 2024b. doi: 10.52202/079017-4478. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/ff417c3993894694e88ffc4d3f53d28b-Paper-Datasets_and_Benchmarks_Track.pdf.

Xin Li, Qizhi Chu, Yubin Chen, Yang Liu, Yaoqi Liu, Zekai Yu, Weize Chen, Chen Qian, Chuan Shi, and Cheng Yang. GraphTeam: Facilitating Large Language Model-based Graph Analysis via Multi-Agent Collaboration, 2025b. URL https://arxiv.org/abs/2410.18032.

Yuangang Li, Yiqing Shen, Yi Nian, Jiechao Gao, Ziyi Wang, Chenxiao Yu, Li Li, Jie Wang, Xiyang Hu, and Yue Zhao. Mitigating Hallucinations in Large Language Models via Causal Reasoning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31852–31860, 2026c. doi: 10.1609/aaai.v40i38.40454. URL https://ojs.aaai.org/index.php/AAAI/article/view/40454.

Yunxin Li, Baotian Hu, Haoyuan Shi, Wei Wang, Longyue Wang, and Min Zhang. VisionGraph: Leveraging Large Multimodal Models for Graph Theory Problems in Visual Context. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 27903–27919. PMLR, 2024c. URL https://proceedings.mlr.press/v235/li24ab.html.

Zehui Li, Xiangyu Zhao, Mingzhu Shen, Guy-Bart Stan, Pietro Liò, and Yiren Zhao. Hybrid Graph: A Unified Graph Representation with Datasets and Benchmarks for Complex Graphs. *arXiv preprint arXiv:2306.05108*, 2024d. URL https://arxiv.org/abs/2306.05108.

Wei Liu, Zhongyu Niu, Lang Gao, Zhiying Deng, Jun Wang, Haozhao Wang, and Ruixuan Li. Adversarial Cooperative Rationalization: The Risk of Spurious Correlations in Even Clean Datasets. In *International Conference on Machine Learning*, pp. 39126–39146, 2025a. URL https://proceedings.mlr.press/v267/liu25av.html.

Yuanye Liu, Jiahang Xu, Li Lyna Zhang, Qi Chen, Xuan Feng, Yang Chen, Zhongxin Guo, Yuqing Yang, and Peng Cheng. Beyond Prompt Content: Enhancing LLM Performance via Content-Format Integrated Prompt Optimization. *arXiv preprint arXiv:2502.04295*, 2025b. URL https://arxiv.org/abs/2502.04295.

Zhengliang Liu, Yue Huang, Xiaowei Yu, Lu Zhang, Zihao Wu, Chao Cao, Haixing Dai, Lin Zhao, Yiwei Li, Peng Shu, Fang Zeng, Lichao Sun, Wei Liu, Dinggang Shen, Quanzheng Li, Tianming Liu, Dajiang Zhu, and Xiang Li. DeID-GPT: Zero-shot Medical Text De-Identification by GPT-4. *arXiv preprint arXiv:2303.11032*, 2025c. URL https://arxiv.org/abs/2303.11032.

Haoran Luo, Haihong E, Ling Tan, Gengxian Zhou, Tianyu Yao, and Kaiyang Wan. DHGE: Dual-View Hyper-Relational Knowledge Graph Embedding for Link Prediction and Entity Typing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 6467–6474, 2023a. doi: 10.1609/aaai.v37i5.25795. URL https://ojs.aaai.org/index.php/AAAI/article/view/25795.

Haoran Luo, Haihong E, Yuhao Yang, Yikai Guo, Mingzhi Sun, Tianyu Yao, Zichen Tang, Kaiyang Wan, Meina Song, and Wei Lin. HAHE: Hierarchical Attention for Hyper-Relational Knowledge Graphs in Global and Local Level. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8095–8107, 2023b. doi: 10.18653/v1/2023.acl-long.450. URL https://aclanthology.org/2023.acl-long.450/.

Haoran Luo, Haihong E, Yuhao Yang, Gengxian Zhou, Yikai Guo, Tianyu Yao, Zichen Tang, Xueyuan Lin, and Kaiyang Wan. NQE: N-ary Query Embedding for Complex Query Answering over Hyper-Relational Knowledge Graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 4543–4551, 2023c. doi: 10.1609/aaai.v37i4.25576. URL https://ojs.aaai.org/index.php/AAAI/article/view/25576.

Haoran Luo, Haihong E, Yuhao Yang, Tianyu Yao, Yikai Guo, Zichen Tang, Wentai Zhang, Shiyao Peng, Kaiyang Wan, Meina Song, Wei Lin, Yifan Zhu, and Luu A. Tuan. Text2NKG: Fine-Grained N-ary Relation Extraction for N-ary relational Knowledge Graph Construction. In *Advances in Neural Information Processing Systems*, volume 37, pp. 27417–27439, 2024a. doi: 10.52202/079017-0861. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/305b2288122d46bf0641bdd86c9a7921-Abstract-Conference.html.

Xiaonan Luo, Yue Huang, Ping He, and Xiangliang Zhang. Better Datasets Start from RefineLab: Automatic Optimization for High-Quality Dataset Refinement. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 32374–32382, 2026. doi: 10.1609/aaai.v40i38.40512. URL https://ojs.aaai.org/index.php/AAAI/article/view/40512.

Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, Xing Xie, and Hai Jin. GraphInstruct: Empowering Large Language Models with Graph Understanding and Reasoning Capability. *arXiv preprint arXiv:2403.04483*, 2024b. URL https://arxiv.org/abs/2403.04483.

Yuchen Ma, Yue Huang, Han Bao, Haomin Zhuang, Swadheen Shukla, Michel Galley, Xiangliang Zhang, and Stefan Feuerriegel. SkillGen: Verified Inference-Time Agent Skill Synthesis. *arXiv preprint arXiv:2605.10999*, 2026. URL https://arxiv.org/abs/2605.10999.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. In *Advances in Neural Information Processing Systems*, volume 36, pp. 46534–46594. Curran Associates, Inc., 2023. doi: 10.52202/075280-2019. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/91edff07232fb1b55a505a9e9f6c0ff3-Paper-Conference.pdf.

Elan Markowitz, Krupa Galiya, Greg Ver Steeg, and Aram Galstyan. KG-LLM-Bench: A Scalable Benchmark for Evaluating LLM Reasoning on Textualized Knowledge Graphs. In *Fourth Workshop on Knowledge-Augmented Natural Language Processing at NAACL-HLT*, 2025. URL https://knowledge-nlp.github.io/naacl2025/papers/39.pdf.

Meta. Llama 3.3-70B-Instruct, 2024a. URL https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct.

Meta. Llama 3.1-8B-Instruct, 2024b. URL https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct.

Jacqueline L. Mitchell, Brian Hyeongseok Kim, Chenyu Zhou, and Chao Wang. Understanding Formal Reasoning Failures in LLMs as Abstract Interpreters. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*. Association for Computing Machinery, 2025. doi: 10.1145/3759425.3763389. URL https://arxiv.org/abs/2503.12686.

Yi Nian, Tiankai Yang, Yudi Zhang, Qi Pan, Zelong Xu, Shenzhe Zhu, Qingqing Luan, Yue Huang, Xiangliang Zhang, and Yue Zhao. DOG-DPO:Dynamic Optimization in Geometry for Safety Alignment. *arXiv preprint arXiv:2606.07678*, 2026. URL https://arxiv.org/abs/2606.07678.

OpenAI. GPT-4o System Card. *arXiv preprint arXiv:2410.21276*, 2024a. URL https://arxiv.org/abs/2410.21276.

OpenAI. GPT-4o mini: Advancing Cost-Efficient Intelligence, 2024b. URL https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

OpenAI. o3-mini, 2025. URL https://openai.com/index/openai-o3-mini/.

Sheng Ouyang, Yulan Hu, Ge Chen, and Yong Liu. GUNDAM: Aligning Large Language Models with Graph Understanding, 2024. URL https://arxiv.org/abs/2409.20053.

Yuyang Peng, Yanling Xu, Shuyi Wang, Xiaofei Liao, and Qinbin Li. Parallelizing LLM Agent Execution with Contrastive Task Allocation. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 3967–3977, 2026. doi: 10.1145/3770855.3817872. URL https://dl.acm.org/doi/10.1145/3770855.3817872.

Bryan Perozzi, Bahare Fatemi, Dustin Zelle, Anton Tsitsulin, Mehran Kazemi, Rami Al-Rfou, and Jonathan Halcrow. Let Your Graph Do the Talking: Encoding Structured Data for LLMs. *arXiv preprint arXiv:2402.05862*, 2024. URL https://arxiv.org/abs/2402.05862.

Yuehan Qin, Yichi Zhang, Yi Nian, Xueying Ding, and Yue Zhao. MetaOOD: Automatic Selection of OOD Detection Models. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/3a2e5889b4bbef997ddb13b55d5acf77-Abstract-Conference.html.

Yuehan Qin, Li Li, Defu Cao, Tiankai Yang, Jiate Li, and Yue Zhao. M3OOD: Automatic Selection of Multimodal OOD Detectors. *arXiv preprint arXiv:2508.11936*, 2026a. URL https://arxiv.org/abs/2508.11936.

Yuehan Qin, Shawn Li, Yi Nian, Xinyan Velocity Yu, Yue Zhao, and Xuezhe Ma. Don't Let It Hallucinate: Premise Verification via Retrieval-Augmented Logical Reasoning. *arXiv preprint arXiv:2504.06438*, 2026b. URL https://arxiv.org/abs/2504.06438.

Qwen Team. QwQ-32B: Embracing the Power of Reinforcement Learning, 2025. URL https://qwenlm.github.io/blog/qwq-32b/.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *Advances in Neural Information Processing Systems*, volume 36, pp. 53728–53741. Curran Associates, Inc., 2023. doi: 10.52202/075280-2338. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/a85b405ed65c6477a4fe8302b5e06ce7-Paper-Conference.pdf.

Jerret Ross, Brian Belgodere, Vijil Chenthamarakshan, Inkit Padhi, Youssef Mroueh, and Payel Das. Large-scale chemical language representations capture molecular structure and properties. *Nature Machine Intelligence*, 4(12):1256–1264, 2022. doi: 10.1038/s42256-022-00580-7. URL https://doi.org/10.1038/s42256-022-00580-7.

Yanchi Ru, Yue Huang, and Xiangliang Zhang. RMO: Towards Better LLM Alignment via Reshaping Reward Margin Distributions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 32851–32859, 2026. doi: 10.1609/aaai.v40i39.40565. URL https://ojs.aaai.org/index.php/AAAI/article/view/40565.

Clayton Sanford, Bahare Fatemi, Ethan Hall, Anton Tsitsulin, Mehran Kazemi, Jonathan Halcrow, Bryan Perozzi, and Vahab Mirrokni. Understanding Transformer Reasoning Capabilities via Graph Algorithms. In *Advances in Neural Information Processing Systems*, volume 37, pp. 78320–78370, 2024.

doi: 10.52202/079017-2490. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/8f395480c04ac6dfb2c2326a639df88e-Paper-Conference.pdf.

Abulhair Saparov, Srushti Ajay Pawar, Shreyas Pimpalgaonkar, Nitish Joshi, Richard Yuanzhe Pang, Vishakh Padmakumar, Seyed Mehran Kazemi, Najoung Kim, and He He. Transformers Struggle to Learn to Search. In *International Conference on Learning Representations*, pp. 16028–16053, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/28d43a64147bf94921041599efdf791d-Paper-Conference.pdf.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *Advances in Neural Information Processing Systems*, volume 36, pp. 68539–68551. Curran Associates, Inc., 2023. doi: 10.52202/075280-2997. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/d842425e4bf79ba039352da0f658a906-Paper-Conference.pdf.

Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting. In *International Conference on Learning Representations*, pp. 25055–25083, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/6c0e99d736da621403018ca7b32b1a4d-Paper-Conference.pdf.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. In *Advances in Neural Information Processing Systems*, volume 36, pp. 38154–38180. Curran Associates, Inc., 2023. doi: 10.52202/075280-1657. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/77c33e6a367922d003ff102ffb92b658-Paper-Conference.pdf.

Jiawen Shi, Zenghui Yuan, Yinuo Liu, Yue Huang, Pan Zhou, Lichao Sun, and Neil Zhenqiang Gong. Optimization-based Prompt Injection Attack to LLM-as-a-Judge. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 660–674, 2024. doi: 10.1145/3658644.3690291. URL https://doi.org/10.1145/3658644.3690291.

Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. *arXiv preprint arXiv:2506.07675*, 2026a. URL https://arxiv.org/abs/2506.07675.

Zirui Song, Guangxian Ouyang, Meng Fang, Hongbin Na, Zijing Shi, Zhenhao Chen, Yujie Fu, Zeyu Zhang, Shiyu Jiang, Miao Fang, Ling Chen, and Xiuying Chen. Hazards in Daily Life? Enabling Robots to Proactively Detect and Resolve Anomalies. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 7399–7415. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.379. URL https://aclanthology.org/2025.naacl-long.379/.

Zirui Song, Guangxian Ouyang, Mingzhe Li, Yuheng Ji, Chenxi Wang, Zixiang Xu, Zeyu Zhang, Xiaoqing Zhang, Qian Jiang, Fengxian Ji, Zhenhao Chen, Zhongzhi Li, and Xiuying Chen. ManipLVM-R1: Reinforcement Learning for Reasoning in Embodied Manipulation with Large Vision-Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 18558–18566, 2026b. doi: 10.1609/aaai.v40i22.38922. URL https://ojs.aaai.org/index.php/AAAI/article/view/38922.

Oyvind Tafjord, Bhavana Dalvi Mishra, and Peter Clark. ProofWriter: Generating Implications, Proofs, and Abductive Statements over Natural Language. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pp. 3621–3634. Association for Computational Linguistics, 2021. doi: 10.18653/v1/2021.findings-acl.317. URL https://aclanthology.org/2021.findings-acl.317/.

Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Long Xia, Dawei Yin, and Chao Huang. HiGPT: Heterogeneous Graph Language Model. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 2842–2853, 2024. doi: 10.1145/3637528.3671987. URL https://doi.org/10.1145/3637528.3671987.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. GraphArena: Evaluating and Exploring Large Language Models on Graph Computation. In *International Conference on Learning Representations*, pp. 48118–48145, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/77fa8253adfc8b33209639f3e9985741-Paper-Conference.pdf.

Shubo Tian, Qiao Jin, Lana Yeganova, Po-Ting Lai, Qingqing Zhu, Xiuying Chen, Yifan Yang, Qingyu Chen, Won Kim, Donald C Comeau, Rezarta Islamaj, Aadit Kapoor, Xin Gao, and Zhiyong Lu. Opportunities and challenges for ChatGPT and large language models in biomedicine and health. *Briefings in Bioinformatics*, 25(1):bbad493, 2024. doi: 10.1093/bib/bbad493. URL https://doi.org/10.1093/bib/bbad493.

Kaiyang Wan, Honglin Mu, Rui Hao, Haoran Luo, Tianle Gu, and Xiuying Chen. A Cognitive Writing Perspective for Constrained Long-Form Text Generation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 9832–9844, 2025. doi: 10.18653/v1/2025.findings-acl.511. URL https://aclanthology.org/2025.findings-acl.511/.

Kaiyang Wan, Lang Gao, Honglin Mu, Preslav Nakov, Yuxia Wang, and Xiuying Chen. A Fano-Style Accuracy Upper Bound for LLM Single-Pass Reasoning in Multi-Hop QA. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2509.21199.

Chenxi Wang, Tianle Gu, Zhongyu Wei, Lang Gao, Zirui Song, and Xiuying Chen. Word Form Matters: LLMs' Semantic Reconstruction under Typoglycemia. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 16870–16885, 2025a. doi: 10.18653/v1/2025.findings-acl.866. URL https://aclanthology.org/2025.findings-acl.866/.

Chenxi Wang, Zongfang Liu, Dequan Yang, and Xiuying Chen. Decoding Echo Chambers: LLM-Powered Simulations Revealing Polarization in Social Networks. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 3913–3923. Association for Computational Linguistics, 2025b. URL https://aclanthology.org/2025.coling-main.264/.

Chenxi Wang, Yixuan Zhang, Lang Gao, Zixiang Xu, Zirui Song, Yanbo Wang, and Xiuying Chen. Under the Shadow of Babel: How Language Shapes Reasoning in LLMs. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 24327–24344, 2025c. doi: 10.18653/v1/2025.findings-emnlp.1321. URL https://aclanthology.org/2025.findings-emnlp.1321/.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can Language Models Solve Graph Problems in Natural Language? In *Advances in Neural Information Processing Systems*, volume 36, pp. 30840–30861. Curran Associates, Inc., 2023a. doi: 10.52202/075280-1345. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/622afc4edf2824a1b6aaf5afe153fa93-Paper-Conference.pdf.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634. Association for Computational Linguistics, 2023b. doi: 10.18653/v1/2023.acl-long.147. URL https://aclanthology.org/2023.acl-long.147/.

Qinyong Wang, Zhenxiang Gao, and Rong Xu. Graph Agent: Explicit Reasoning Agent for Graphs. *arXiv preprint arXiv:2310.16421*, 2023c. URL https://arxiv.org/abs/2310.16421.

Rongzheng Wang, Shuang Liang, Qizhi Chen, Jiasheng Zhang, and Ke Qin. GraphTool-Instruction: Revolutionizing Graph Reasoning in LLMs through Decomposed Subtask Instruction. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, pp. 1492–1503. Association for Computing Machinery, 2025d. doi: 10.1145/3690624.3709238. URL https://doi.org/10.1145/3690624.3709238.

Xiangqi Wang, Yue Huang, Yanbo Wang, Xiaonan Luo, Kehan Guo, Yujun Zhou, and Xiangliang Zhang. AdaReasoner: Adaptive Reasoning Enables More Flexible Thinking. In *Advances in Neural Information Processing Systems*, volume 38, pp. 75737–75764, 2025e. doi: 10.52202/085713-2545. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/6dc129f0383f9fa270347bb09f0030d4-Abstract-Conference.html.

Yanbo Wang, Zixiang Xu, Yue Huang, Chujie Gao, Siyuan Wu, Jiayi Ye, Pin-Yu Chen, Xiuying Chen, and Xiangliang Zhang. Adaptive Distraction: Probing LLM Contextual Robustness with Automated Tree Search. In *Advances in Neural Information Processing Systems*, volume 38, pp. 165093–165131. Curran Associates, Inc., 2025f. doi: 10.52202/085713-5504. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/f13d74c6666087aa6eea3d17820e6a23-Paper-Conference.pdf.

Yanbo Wang, Zixiang Xu, Yue Huang, Xiangqi Wang, Zirui Song, Lang Gao, Chenxi Wang, Xiangru Tang, Yue Zhao, Arman Cohan, Xiangliang Zhang, and Xiuying Chen. DyFlow: Dynamic Workflow Framework for Agentic Reasoning. In *Advances in Neural Information Processing Systems*, volume 38, pp. 174148–174181. Curran Associates, Inc., 2025g. doi: 10.52202/085713-5793. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/fe9910d2b03324faeb5371a9658277bb-Paper-Conference.pdf.

Yiqin Wang, Haoji Zhang, Jingqi Tian, and Yansong Tang. Ponder & Press: Advancing Visual GUI Agent towards General Computer Control. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1461–1473, 2025h. doi: 10.18653/v1/2025.findings-acl.76. URL https://aclanthology.org/2025.findings-acl.76/.

Yuyao Wang, Bowen Liu, Jianheng Tang, Nuo Chen, Yuhan Li, Qifan Zhang, Chenyi Zi, Chen Zhang, and Jia Li. NPG-Muse: Scaling Long Chain-of-Thought Reasoning with NP-Hard Graph Problems. *arXiv preprint arXiv:2508.20373*, 2025i. URL https://arxiv.org/abs/2508.20373.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022. doi: 10.52202/068431-1800. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

Wang Wei, Hongzheng Yang, Tiankai Yang, Samyadeep Basu, Hongjie Chen, Yue Zhao, Franck Dernoncourt, Ryan A. Rossi, and Hoda Eldardiry. FlexRouter: Learning Complementary Model Sets for Flexible LLM Routing. Adobe Research, 2026a. URL https://research.adobe.com/publication/flexrouter-learning-complementary-model-sets-for-flexible-llm-routing/.

Yanbin Wei, Jiangyue Yan, Chun Kang, Yang Chen, Hua Liu, James Kwok, and Yu Zhang. DynamicGTR: Leveraging Graph Topology Representation Preferences to Boost VLM Capabilities on Graph QAs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2026b. URL https://arxiv.org/abs/2602.21864.

Qiming Wu, Zichen Chen, Will Corcoran, Misha Sra, and Ambuj Singh. GraphEval36K: Benchmarking Coding and Reasoning Capabilities of Large Language Models on Graph Datasets. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 8110–8132. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.findings-naacl.452. URL https://aclanthology.org/2025.findings-naacl.452/.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-Scale Terminal Agentic Trajectory Generation from Dockerized Environments. *arXiv preprint arXiv:2602.01244*, 2026. URL https://arxiv.org/abs/2602.01244.

Songhao Wu, Quan Tu, Hong Liu, Jia Xu, Zhongyi Liu, Guannan Zhang, Ran Wang, Xiuying Chen, and Rui Yan. Unify Graph Learning with Text: Unleashing LLM Potentials for Session Search. In *Proceedings of the ACM Web Conference 2024*, pp. 1509–1518, 2024. doi: 10.1145/3589334.3645574. URL https://doi.org/10.1145/3589334.3645574.

Hanwen Xing, Haomin Zhuang, Xuandong Zhao, Yue Huang, Zhenheng Tang, and Xiangliang Zhang. Recipes for Agents: Understanding Skills and Their Open Questions. In *Proceedings of the 32nd ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 13245–13251, 2026. doi: 10.1145/3770855.3818652. URL https://doi.org/10.1145/3770855.3818652.

Hao Xu, Xiangru Jian, Xinjian Zhao, Wei Pang, Chao Zhang, Suyuchen Wang, Qixin Zhang, Zhengyuan Dong, Joao Monteiro, Bang Liu, Qiuzhuang Sun, and Tianshu Yu. GraphOmni: A Comprehensive and Extensible Benchmark Framework for Large Language Models on Graph-theoretic Tasks. In *International Conference on Learning Representations (ICLR)*, 2026a. URL https://arxiv.org/abs/2504.12764.

Haoyan Xu, Kay Liu, Zhengtao Yao, Philip S. Yu, Mengyuan Li, Kaize Ding, and Yue Zhao. LEGO-Learn: Label-Efficient Graph Open-Set Learning. *arXiv preprint arXiv:2410.16386*, 2025a. URL https://arxiv.org/abs/2410.16386.

Haoyan Xu, Ruizhi Qian, Zhengtao Yao, Ziyi Liu, Li Li, Yuqi Li, Yanshu Li, Wenqing Zheng, Daniele Rosa, Daniel Barcklow, Senthil Kumar, Jieyu Zhao, and Yue Zhao. LLM-Powered Text-Attributed Graph Anomaly Detection via Retrieval-Augmented Reasoning. *arXiv preprint arXiv:2511.17584*, 2025b. URL https://arxiv.org/abs/2511.17584.

Haoyan Xu, Zhengtao Yao, Yushun Dong, Ziyi Wang, Ryan Rossi, Mengyuan Li, and Yue Zhao. Few-Shot Graph Out-of-Distribution Detection with LLMs. In *Machine Learning and Knowledge Discovery in Databases. Research Track*, volume 16016 of *Lecture Notes in Computer Science*, pp. 307–324. Springer Nature Switzerland, 2025c. doi: 10.1007/978-3-032-06078-5_18. URL https://link.springer.com/chapter/10.1007/978-3-032-06078-5_18.

Haoyan Xu, Zhengtao Yao, Ziyi Wang, Zhan Cheng, Xiyang Hu, Mengyuan Li, and Yue Zhao. Graph Synthetic Out-of-Distribution Exposure with Large Language Models. *arXiv preprint arXiv:2504.21198*, 2025d. URL https://arxiv.org/abs/2504.21198.

Ruiyao Xu, Mihir Parmar, Tiankai Yang, Zhengyu Hu, Yue Zhao, and Kaize Ding. CoAct: Co-Active LLM Preference Learning with Human-AI Synergy. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 16508–16525, 2026b. doi: 10.18653/v1/2026.acl-long.751. URL https://aclanthology.org/2026.acl-long.751/.

Ruiyao Xu, Tiankai Yang, and Wei-Chieh Huang. HyperSkill: Self-Evolving LLM Agents via Hypergraph-Structured Skill Memory. *arXiv preprint arXiv:2608.16114*, 2026c. URL https://arxiv.org/abs/2608.16114.

Zixiang Xu, Yanbo Wang, Yue Huang, Xiuying Chen, Jieyu Zhao, Meng Jiang, and Xiangliang Zhang. Cross-Lingual Pitfalls: Automatic Probing Cross-Lingual Weakness of Multilingual Large Language Models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8254–8284. Association for Computational Linguistics, 2025e. doi: 10.18653/v1/2025.acl-long.404. URL https://aclanthology.org/2025.acl-long.404/.

Zixiang Xu, Sixian Li, Huaxing Liu, Xiang Wang, Shuai Li, Zirui Song, and Xiuying Chen. Inside the Unfair Judge: A Mechanistic Interpretability Account of LLM-as-Judge Bias, 2026d. URL https://arxiv.org/abs/2607.11871.

Zixiang Xu, Jiaan Wang, and Fandong Meng. AlgoWorlds: Benchmarking Tool Use for Global Optimization in Algorithmic Worlds, 2026e. URL https://arxiv.org/abs/2608.29397.

Zixiang Xu, Yanbo Wang, Yue Huang, Haomin Zhuang, Yujun Zhou, Jiayi Ye, Sixian Li, Zirui Song, Lang Gao, Chenxi Wang, Zhaorun Chen, Wang Pan, Yue Zhao, Jieyu Zhao, Xiangliang Zhang, and Xiuying Chen. SocialMaze: A Benchmark for Evaluating and Enhancing Social Reasoning in Large Language Models in Complex Social Environments, 2026f. URL https://arxiv.org/abs/2505.23713v2.

Tiankai Yang, Junjun Liu, Michael Siu, Jiahang Wang, Zhuangzhuang Qian, Chanjuan Song, Cheng Cheng, Xiyang Hu, and Yue Zhao. AD-AGENT: A Multi-agent Framework for End-to-end Anomaly Detection. In *Proceedings of the 14th International Joint Conference on Natural Language Processing and the 4th Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics*, pp. 191–205, 2025a. doi: 10.18653/v1/2025.findings-ijcnlp.11. URL https://aclanthology.org/2025.findings-ijcnlp.11/.

Tiankai Yang, Yi Nian, Li Li, Ruiyao Xu, Yuangang Li, Jiaqi Li, Zhuo Xiao, Xiyang Hu, Ryan A. Rossi, Kaize Ding, Xia Hu, and Yue Zhao. AD-LLM: Benchmarking Large Language Models for Anomaly Detection. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 1524–1547, 2025b. doi: 10.18653/v1/2025.findings-acl.79. URL https://aclanthology.org/2025.findings-acl.79/.

Wei Yang and Jesse Thomason. Learning to Deliberate: Meta-policy Collaboration for Agentic LLMs with Multi-agent Reinforcement Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40 (35):29820–29828, 2026. doi: 10.1609/aaai.v40i35.40228. URL https://ojs.aaai.org/index.php/AAAI/article/view/40228.

Wei Yang, Jiacheng Pang, Shixuan Li, Paul Bogdan, Stephen Tu, and Jesse Thomason. Maestro: Learning to Collaborate via Conditional Listwise Policy Optimization for Multi-Agent LLMs. *arXiv preprint arXiv:2511.06134*, 2025c. URL https://arxiv.org/abs/2511.06134.

Wei Yang, Defu Cao, Jiacheng Pang, Muyan Weng, and Yan Liu. Adaptive Collaboration with Humans: Metacognitive Policy Optimization for Multi-Agent LLMs with Continual Learning. In *International Conference on Learning Representations*, 2026a. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/688964ef60e83fb3c3efebdffe2f06f7-Abstract-Conference.html.

Wei Yang, Bryce Kan, Shixuan Li, Li Li, Yuehan Qin, Jiate Li, Paul Bogdan, and Jesse Thomason. RaMem: Contextual Reinstatement for Long-term Agentic Memory. *arXiv preprint arXiv:2606.22844*, 2026b. URL https://arxiv.org/abs/2606.22844.

Wei Yang, Shixuan Li, Heng Ping, Peiyu Zhang, Paul Bogdan, and Jesse Thomason. Auditing Multi-Agent LLM Reasoning Trees Outperforms Majority Vote and LLM-as-Judge. *arXiv preprint arXiv:2602.09341*, 2026c. URL https://arxiv.org/abs/2602.09341.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://iclr.cc/virtual/2023/poster/11003.

Jiayi Ye, Yanbo Wang, Yue Huang, Dongping Chen, Qihui Zhang, Nuno Moniz, Tian Gao, Werner Geyer, Chao Huang, Pin-Yu Chen, Nitesh V. Chawla, and Xiangliang Zhang. Justice or Prejudice? Quantifying Biases in LLM-as-a-Judge. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/fdca08d371e4b6c031397909e20043bd-Abstract-Conference.html.

Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. GraCoRe: Benchmarking Graph Comprehension and Complex Reasoning in Large Language Models. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7925–7948. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.531/.

Angelo Zangari, Peyman Baghershahi, and Sourav Medya. Colorful Talks with Graphs: Human-Interpretable Graph Encodings for Large Language Models. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 41194–41215. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.2049. URL https://aclanthology.org/2026.findings-acl.2049/.

Erhan Zhang, Yiqun Chen, Zechun Niu, Wei Yang, Xiaochi Wei, Yan Gao, Yi Wu, Yao Hu, and Jiaxin Mao. OASES: Outcome-Aligned Search-Evaluation Co-Training for Agentic Search. *arXiv preprint arXiv:2604.03675*, 2026a. URL https://arxiv.org/abs/2604.03675.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent Architecture Search via Agentic Supernet. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 75834–75852. PMLR, 2025a. URL https://proceedings.mlr.press/v267/zhang25bi.html.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, XiongHui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow:

Automating Agentic Workflow Generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/5492ecbce4439401798dcd2c90be94cd-Paper-Conference.pdf.

Qifan Zhang, Nuo Chen, Zehua Li, Miao Peng, Jing Tang, and Jia Li. Improving LLMs' Generalized Reasoning Abilities by Graph Problems. In *Conference on Language Modeling (COLM)*, 2025c. URL https://arxiv.org/abs/2507.17168.

Qifan Zhang, Xiaobin Hong, Jianheng Tang, Nuo Chen, Yuhan Li, Wenzhong Li, Jing Tang, and Jia Li. GCoder: Improving Large Language Model for Generalized Graph Reasoning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*, pp. 4149–4159, 2025d. doi: 10.1145/3746252.3761066. URL https://doi.org/10.1145/3746252.3761066.

Qifan Zhang, Jianhao Ruan, Aochuan Chen, Kang Zeng, Nuo Chen, Jing Tang, and Jia Li. Exposing Weaknesses of Large Reasoning Models through Graph Algorithm Problems. *arXiv preprint arXiv:2602.06319*, 2026b. URL https://arxiv.org/abs/2602.06319.

Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xinyun Liu, and Yulia Tsvetkov. Generalizable LLM Learning of Graph Synthetic Data with Post-training Alignment. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 12070–12091. Association for Computational Linguistics, 2026c. doi: 10.18653/v1/2026.findings-acl.586. URL https://aclanthology.org/2026.findings-acl.586/.

Zeyang Zhang, Xin Wang, Ziwei Zhang, Haoyang Li, Yijian Qin, and Wenwu Zhu. LLM4DyG: Can Large Language Models Solve Spatial-Temporal Problems on Dynamic Graphs? In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024. URL https://arxiv.org/abs/2310.17110.

Yue Zhao. GRADE: Graph Representation of LLM Agent Dependency and Execution. *arXiv preprint arXiv:2606.22741*, 2026. URL https://arxiv.org/abs/2606.22741.

Wanrong Zheng, Yunhao Ge, and Laurent Itti. Three-Step Nav: A Hierarchical Global-Local Planner for Zero-Shot Vision-and-Language Navigation. *arXiv preprint arXiv:2604.26946*, 2026. URL https://arxiv.org/abs/2604.26946.

Chenyu Zhou, Yuzhou Fang, Jingbo Wang, and Chao Wang. An Incremental Algorithm for Algebraic Program Analysis. *Proceedings of the ACM on Programming Languages*, 9(POPL):1934–1961, 2025a. doi: 10.1145/3704901. URL https://doi.org/10.1145/3704901.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H Chi. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://iclr.cc/virtual/2023/poster/12263.

Juexiao Zhou, Xiaonan He, Liyuan Sun, Jiannan Xu, Xiuying Chen, Yuetan Chu, Longxi Zhou, Xingyu Liao, Bin Zhang, Shawn Afvari, and Xin Gao. Pre-trained multimodal large language model enhances dermatological diagnosis using SkinGPT-4. *Nature Communications*, 15(1):5649, 2024. doi: 10.1038/s41467-024-50043-3. URL https://doi.org/10.1038/s41467-024-50043-3.

Yujun Zhou, Jiayi Ye, Zipeng Ling, Yufei Han, Yue Huang, Haomin Zhuang, Zhenwen Liang, Kehan Guo, Taicheng Guo, Xiangqi Wang, and Xiangliang Zhang. Dissecting Logical Reasoning in LLMs: A Fine-Grained Evaluation and Supervision Study. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 17075–17098, 2025b. doi: 10.18653/v1/2025.findings-emnlp.926. URL https://aclanthology.org/2025.findings-emnlp.926/.

A Task Definitions and Solution Algorithms in GT Bench

This section provides detailed definitions for the graph-theoretic tasks included in GT Bench, along with the canonical algorithms used to generate their ground-truth solutions. We begin by defining fundamental graph-theoretic concepts used throughout this section. A **graph** G is represented as a pair (V, E), where V is a set of **vertices** (also called nodes) and E is a set of **edges** connecting pairs of vertices. Graphs can be **directed** (edges have a direction from one vertex to another) or **undirected** (edges have no intrinsic direction). Edges may also have associated **weights** or **capacities**. A **path** is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. A **simple path** is a path where all vertices are distinct. A **cycle** is a path that starts and ends at the same vertex. A **simple cycle** is a cycle where all intermediate vertices are distinct.

Connectivity. Definition: This task requires determining whether a path exists between two specified vertices, say u and v, in a given graph $G = (V, E)$. Solution Algorithm: The ground truth is established by performing a Breadth-First Search (BFS) starting from vertex u. If vertex v is visited during the traversal, the vertices are deemed connected; otherwise, they are not.

Bipartiteness Check. Definition: This task aims to determine if the vertices of a graph $G = (V, E)$ can be partitioned into two disjoint and independent sets, U and W, such that every edge in E connects a vertex in U to one in W. Solution Algorithm: A graph coloring approach using two colors is implemented via BFS. Starting from an arbitrary uncolored vertex and assigning it a color, its neighbors are assigned the other color, and this process continues. If an edge is found to connect two vertices of the same color, the graph is not bipartite; otherwise, it is.

Minimum Cycle Length. Definition: For an unweighted graph $G = (V, E)$, this task is to find the length (i.e., the number of edges) of the shortest simple cycle. Solution Algorithm: The length of the shortest cycle is found by iterating through each vertex $s \in V$. For each s, a BFS is initiated. If the BFS encounters a previously visited vertex t (other than the immediate parent of the current vertex in the BFS tree for s), a cycle is found. The length of this cycle is $\text{depth}(s_{\text{current}}) + \text{depth}(t) + 1$. The minimum such length over all starting vertices s and all detected cycles forms the minimum cycle length.

Maximum Clique Size. Definition: A clique in an undirected graph $G = (V, E)$ is a subset of vertices such that every two distinct vertices in the clique are adjacent. This task requires finding the number of vertices in the largest such subset. Solution Algorithm: Given that finding the maximum clique is NP-hard, a backtracking-based brute-force search algorithm is employed. This algorithm systematically explores all possible subsets of vertices, checking if each forms a clique, and keeps track of the largest clique found.

Maximum Independent Set Size. Definition: An independent set in an undirected graph $G = (V, E)$ is a subset of vertices such that no two vertices in the subset are adjacent. This task asks for the size of the largest such set. Solution Algorithm: This problem is NP-hard. Similar to the maximum clique problem, a backtracking-based brute-force search algorithm is used. It explores all subsets of vertices, verifies the independent set property, and records the size of the largest valid independent set encountered.

Eulerian Path. Definition: An Eulerian path in a graph is a trail that visits every edge exactly once. This task determines whether such a path exists. Solution Algorithm: For an undirected graph G, an Eulerian path exists if and only if G is connected (considering only non-isolated vertices) and has exactly zero or two vertices of odd degree. Connectivity is checked using BFS. The degrees of all vertices are computed, and these conditions are verified. For a directed graph, similar conditions regarding in-degrees and out-degrees and strong connectivity are checked.

Eulerian Circuit. Definition: An Eulerian circuit is an Eulerian path that starts and ends on the same vertex. This task determines if such a circuit exists. Solution Algorithm: For an undirected graph G, an Eulerian circuit exists if and only if G is connected (verified by BFS) and all vertices have an even

degree. For a directed graph, it exists if and only if the graph is strongly connected and every vertex v has in-degree(v) = out-degree(v).

Hamiltonian Path. Definition: A Hamiltonian path is a path in an undirected or directed graph that visits each vertex exactly once. This task is to determine if such a path exists. Solution Algorithm: This problem is NP-complete. A backtracking-based brute-force search algorithm is employed. The algorithm attempts to build a path vertex by vertex, ensuring each vertex is visited exactly once, exploring all valid sequences.

Hamiltonian Circuit. Definition: A Hamiltonian circuit is a Hamiltonian path that is a cycle. Solution Algorithm: This problem is NP-complete. A backtracking-based brute-force search, similar to that for Hamiltonian paths, is used, with the additional constraint that the path must form a cycle by connecting the last vertex in the path back to the first.

Biconnected Components Count. Definition: A biconnected component of a graph is a maximal biconnected subgraph. A graph is biconnected if it remains connected if any single vertex is removed. This task counts the number of biconnected components. Solution Algorithm: The ground truth is found using a Depth-First Search (DFS)-based algorithm that identifies articulation points and bridges. Edges are then grouped into biconnected components based on these findings.

Bridge Count. Definition: A bridge in an undirected graph is an edge whose removal increases the number of connected components. This task requires counting the number of such bridges. Solution Algorithm: A DFS-based algorithm is used. During the DFS, discovery times ($disc[u]$) and low-link values ($low[u]$) are maintained. An edge (u, v) where v is a child of u in the DFS tree is a bridge if $low[v] > disc[u]$.

Triangle Count. Definition: This task is to count the number of simple cycles of length 3 (triangles) in an undirected graph. Solution Algorithm: The number of triangles is computed by iterating through all unique triples of vertices (u, v, w) and checking if edges (u, v), (v, w), and (w, u) all exist in the graph

Cycle Count. Definition: This task requires counting the total number of simple cycles in the graph. Solution Algorithm: For the small graphs specified for this task in GT Bench (6-8 nodes), a DFS-based backtracking search is implemented to explicitly enumerate and count all elementary circuits.

Spanning Tree Count. Definition: This task is to count the number of distinct spanning trees in a given connected undirected graph. Solution Algorithm: Kirchhoff's Matrix Tree Theorem is applied. The Laplacian matrix L of the graph is constructed ($L = D - A$, where D is the degree matrix and A is the adjacency matrix). Any cofactor of L gives the number of spanning trees.

Shortest Path Length. Definition: This task involves computing the length of a shortest path between two specified vertices s and t. Solution Algorithm: For unweighted graphs, Breadth-First Search (BFS) starting from s is used. For weighted graphs with non-negative edge weights, Dijkstra's algorithm is employed.

Minimum Spanning Tree (MST) Weight. Definition: Given a connected, undirected, and edge-weighted graph, this task requires computing the total weight of a Minimum Spanning Tree (MST). Solution Algorithm: Kruskal's algorithm is used to find an MST. The sum of the weights of the edges in the found MST is the result.

Second Minimum Spanning Tree (SMST) Weight. Definition: This task is to find the total weight of a spanning tree whose weight is the smallest among all spanning trees that are different from an MST. Solution Algorithm: First, an MST T is found using Kruskal's algorithm. Then, for each edge (u, v) in the original graph G not in T, adding (u, v) to T creates a cycle. On this cycle, the edge (x, y) (from T, different from (u, v)) with the maximum weight is identified. Replacing (x, y) with (u, v) forms a new spanning tree T'. The SMST weight is the minimum weight among all such T' that is strictly greater than the MST weight.

Tree Diameter. Definition: The diameter of a tree is the length of the longest shortest path between any pair of nodes. Solution Algorithm: This is found using two BFS traversals: 1. Start a BFS from an arbitrary node s to find the node u farthest from s. 2. Start another BFS from u to find the node v farthest from u. The distance between u and v is the diameter.

Tree Centroid. Definition: A centroid of a tree is a node c such that removing c divides the tree into components each with at most $|V|/2$ nodes. The task is to find such a centroid (smallest index if multiple exist). Solution Algorithm: A DFS traversal computes subtree sizes. A node v is a centroid if the size of each subtree rooted at a child of v, and the size of the remaining tree if v is removed, are all $\leq |V|/2$. The centroid with the smallest index is selected.

Lowest Common Ancestor (LCA). Definition: Given a rooted tree (node 1 as root) and two nodes u and v, the LCA is the deepest node that is an ancestor of both u and v. Solution Algorithm: Depths and parent pointers are computed via DFS. To find LCA(u, v): 1. Bring u and v to the same depth by moving the deeper node up. 2. If $u = v$, it is the LCA. 3. Else, move both u and v upwards simultaneously until they meet. This meeting point is the LCA.

Tree Maximum Independent Set Size. Definition: For a given tree, find the size of a maximum independent set. Solution Algorithm: Dynamic programming on trees is used. For each node u, $dp[u][1]$ (size of max independent set in subtree of u, including u) and $dp[u][0]$ (size, excluding u) are computed in a post-order traversal. $dp[u][1] = 1 + \sum_{c \in \text{children}(u)} dp[c][0]$ $dp[u][0] = \sum_{c \in \text{children}(u)} \max(dp[c][1], dp[c][0])$ The answer is $\max(dp[\text{root}][1], dp[\text{root}][0])$.

Maximum Flow. Definition: Given a directed graph with edge capacities, a source s, and a sink t, find the maximum flow from s to t. Solution Algorithm: The Edmonds-Karp algorithm is used. It iteratively finds augmenting paths from s to t in the residual graph using BFS and pushes flow along these paths until no more exist.

Minimum Cut Capacity. Definition: In a flow network, an $s - t$ cut partitions vertices into S (containing s) and T (containing t). The cut capacity is the sum of capacities of edges from S to T. This task asks for the minimum such capacity. Solution Algorithm: By the Max-Flow Min-Cut Theorem, this value is equal to the maximum flow from s to t. Thus, the Edmonds-Karp algorithm is used to compute the maximum flow, which gives the minimum cut capacity.

Minimum-Cost Maximum-Flow Value. Definition: In a flow network with edge capacities and costs per unit of flow, find a flow that maximizes total flow and, among such flows, minimizes total cost. The task asks for this minimum cost. Solution Algorithm: A successive shortest path algorithm using costs as edge lengths in the residual graph is employed. It repeatedly finds the shortest (minimum cost) augmenting path from s to t (using Bellman-Ford due to potential negative costs in residual graphs after flow augmentation) and pushes flow. This continues until no more augmenting paths exist or a pre-calculated maximum flow value is reached. The total cost is accumulated.

B Experimental Details

This section provides further details on the experimental setup, including the models utilized, configurations for baseline methods, and specific parameters for training and evaluation.

B.1 Models

A variety of LLMs were employed in our experiments, both for direct evaluation and as components within the GTA framework and baseline methods. Table 18 lists these models with their versions, originating organizations, licenses, and roles in our study (evaluation or fine-tuning).

Table 18: Models used in our experiments along with their versions, organizations, licenses, and purposes. *Eval*: Model used for evaluation; *FT*: Model used for fine-tuning.

Model	Version	Organization	License	Eval	FT
Phi-4	Phi-4	Microsoft	MIT	✓	✓
GPT-4o-mini	gpt-4o-mini-2024-07-18	OpenAI	Proprietary	✓	
GPT-4o	gpt-4o-2024-08-06	OpenAI	Proprietary	✓	
Llama-3.1-8B	Meta-Llama-3.1-8B-Instruct	Meta	Llama 3.1 Community	✓	
Llama-3.3-70B	Meta-Llama-3.3-70B-Instruct	Meta	Llama-3.3	✓	
QwQ	QwQ-32B	Alibaba	Apache 2.0	✓	
o3-mini	o3-mini-2025-01-31	OpenAI	Proprietary	✓	
Deepseek-R1	DeepSeek-R1	DeepSeek	MIT	✓	

When these models were evaluated directly (i.e., not as part of GTA or other agentic frameworks), we employed Vanilla prompting in a zero-shot setting to assess their baseline graph reasoning capabilities.

B.2 Baselines

For all evaluated baseline methods (Vanilla prompting, Chain-of-Thought (CoT) (Wei et al., 2022), LLM-Debate (Du et al., 2024), Self-Refine (Madaan et al., 2023), ADAS (Hu et al., 2025a), AFlow (Zhang et al., 2025b), and MaAS (Zhang et al., 2025a)), we standardized the executor LLM to be `Phi-4`. This was done to ensure a fair comparison of the reasoning strategies or frameworks themselves, isolating their impact from variations in the underlying model's raw capabilities. For the automated agent frameworks (ADAS, AFlow, MaAS), the respective planner or optimizer components were configured following the specifications and recommendations detailed in their original publications.

In addition, we include two graph-specific baselines for context:

GraphTeam (Reasoning-Only). Following Li et al. (2025b), we ablate the original multi-agent pipeline to a language-only setting by disabling search/coding/tools and external code execution. The executor remains `Phi-4` for consistency with other agentic baselines, so any improvement reflects coordination/reasoning rather than tool use.

GraphWiz-DPO. We evaluate the released *GraphWiz-DPO* checkpoint (Chen et al., 2024) in its prescribed instruction format and report results alongside a vanilla prompt and our GTA:

Config	GT-E (%)	GT-H (%)
Vanilla	30.2	15.4
GraphWiz-DPO	32.5	16.6
GTA (LLaMA 2-13B)	50.8	25.7

Other graph systems use different task or execution interfaces. **Graph Agent** (Wang et al., 2023c) targets node classification and link prediction, while **GraphAgent-Reasoner** (Hu et al., 2025e) distributes computation across node-level agents. **GraphLLM** (Chai et al., 2023) learns graph-enhanced prefixes for a frozen LLM. **GraphThought** (Huang et al., 2025b) and **GUNDAM** (Ouyang et al., 2024) train graph-specialized models, and **GCoder** (Zhang et al., 2025d) solves graph problems through generated code. These differences define the scope of the comparisons above; they do not establish a general ranking of these systems.

Overall, these choices align with our objective: GTA improves performance *by reasoning* (via adaptive representation selection and a lightweight plan–decompose–execute pipeline) without modifying the executor or relying on code execution.

B.3 Training and Evaluation Parameters

GTA Component Training: The fine-tuning of GTA's components involved specific datasets and hyperparameters. For the **Algorithm Decomposer**, the Supervised Fine-Tuning (SFT) stage utilized a dataset of 1,000 instances where decomposition steps were distilled from `GPT-4o`. Subsequently, for Direct Preference Optimization (DPO), 2,000 preference pairs (winning vs. losing decompositions) were used. It is important to note that, given the performance characteristics of the `Phi-4` model used as the base, we constrained the number of decomposed sub-tasks, k, for the Algorithm Decomposer to be no more than 3. Our empirical observations indicated that when k exceeded 3, the computational overhead increased while the overall accuracy of GTA tended to decrease. For training the **Learned Selector**, an initial SFT phase was conducted using 1,000 data instances. This was followed by DPO, where preference pairs were generated based on the end-to-end success of the GTA pipeline with different input representations.

Common hyperparameters for all fine-tuning experiments were as follows: models were trained for 2 epochs with a learning rate of 5.0e-6. We employed a cosine learning rate scheduler with a warmup ratio of 0.1. The per-device training batch size was set to 1, with gradient accumulation performed over 8 steps. Training was conducted using bf16 precision to optimize memory and speed. When generating data for SFT or sampling candidates for DPO, a temperature of 1.0 was used to encourage diversity. All training experiments were conducted on 2 NVIDIA A6000 GPUs over a period of 40 hours.

Evaluation Settings: During all evaluation phases, including direct model assessments and runs of the baseline methods and GTA, the inference temperature for the LLMs (particularly the executor component) was consistently set to 0.01. This low temperature promotes deterministic and more replicable outputs.

For direct model evaluations on **GT Bench**, each model listed in Table 18 was tested on 10,000 instances. For the evaluation of all baseline methods and our GTA framework, each method was tested on 1,000 instances from each respective dataset (GT Bench, GraCoRe, and NLGraph).

B.4 Evaluation Scope and Statistical Protocol

Synthetic, domain-independent tasks. GT Bench consists of synthetically generated instances of classical graph problems with algorithmically verified ground truth. This design isolates structural reasoning from application-specific knowledge. The transfer results on GraCoRe and NLGraph (subsection 6.2) demonstrate gains beyond the training benchmark. The evaluation does not establish performance on domain-attributed graphs or tasks that require social, molecular, or other application-specific knowledge.

Graph scale. GT Bench instances range from 6 to 60 nodes, with per-family ranges set by the prompt-fit budget of section 3. Two constraints jointly fix it. First, a representation-controlled comparison is only meaningful while every instance fits the context window under all four formats, and the token cost of dense graphs grows as $O(V^2)$: a 40-node dense graph already carries up to 780 edges, which is why dense families use tighter node ranges. Second, difficulty is governed by the size of the search space rather than by the node count itself: a 40-node Hamiltonian instance spans a search space beyond 10^{47} orderings, and node ranges are calibrated inversely to algorithmic complexity, from 6–10 nodes for combinatorial-counting tasks such as Cycle Count and Spanning Tree Count up to 60 nodes for tree tasks with linear-time algorithms. The benchmark therefore offers a controlled difficulty gradient rather than a uniform cap, and the regime is demonstrably far from saturated: the best evaluated model reaches 75.1% on GT-H, and model-averaged accuracy remains below 1% under each of the four representations on dense Min-Cost Max-Flow. The scalability study in subsection 6.5 evaluates larger graphs produced by the same generation pipeline; the reported accuracy results remain specific to the evaluated size ranges.

Statistical protocol. Scores in the model-representation sweeps of section 4 come from a single inference run per configuration, decoded at temperature 0.01. These sweeps characterize representation effects across task–representation–model configurations, but do not quantify run-to-run uncertainty for individual cells. The targeted analyses in section 6, including the component ablations, selector checks, and truncation statistics, are averaged over three runs with reported standard deviations. Prompts, outputs, and evaluation scripts are released for independent verification.

C Experimental Results

C.1 Best-Performing Representation Per Task

Table 19 lists the best-performing input representation for each task–graph-type combination in GT Bench, based on average accuracy across all evaluated models. These results informed the design of the Heuristic-Based Selector in GTA.

Table 19: Best-performing input representation format (Best Repr.) per task and graph type, based on average accuracy across all evaluated models (task-level averages in subsection C.2).

Task & Graph Type	Best Repr.	Task & Graph Type	Best Repr.
Biconnected Components (Dense)	AM	Maximum Independent Set (Dense)	AM
Biconnected Components (Sparse)	SL	Maximum Independent Set (Sparse)	AM
Bipartite (Dense)	AM	Min Cost Max Flow (Dense)	AM
Bipartite (Sparse)	NL	Min Cost Max Flow (Sparse)	AL
Bridge Count (Dense)	AL	Minimum Cut (Dense)	AM
Bridge Count (Sparse)	NL	Minimum Cut (Sparse)	AL
Connectivity (Dense)	AM	Minimum Cycle (Dense)	AM
Connectivity (Sparse)	AL	Minimum Cycle (Sparse)	NL
Cycle Count (Dense)	AL	Minimum Spanning Tree (Dense)	AL
Cycle Count (Sparse)	AM	Minimum Spanning Tree (Sparse)	SL
Eulerian Circuit (Dense)	AM	Second MST (Dense)	SL
Eulerian Circuit (Sparse)	SL	Second MST (Sparse)	SL
Eulerian Path (Dense)	AM	Shortest Path (Dense)	AL
Eulerian Path (Sparse)	AM	Shortest Path (Sparse)	AL
Hamiltonian Circuit (Dense)	AL	Spanning Tree Count (Dense)	AM
Hamiltonian Circuit (Sparse)	AL	Spanning Tree Count (Sparse)	AL
Hamiltonian Path (Dense)	AL	Tree Centroid (Tree)	SL
Hamiltonian Path (Sparse)	AL	Tree Diameter (Tree)	SL
Maximum Clique (Dense)	AM	Tree LCA (Tree)	SL
Maximum Clique (Sparse)	NL	Tree Max Independent Set (Tree)	SL
Maximum Flow (Dense)	AM	Triangle Count (Dense)	AM
Maximum Flow (Sparse)	AL	Triangle Count (Sparse)	NL

C.2 Task-Specific Performance Across Input Representations

To further investigate the impact of input graph representation on LLM performance, this section details the average accuracy achieved by the evaluated models for 43 reported task–structure settings in GT Bench, broken down by the four input modalities: Natural Language (NL), Structured Language (SL), Adjacency Matrix (AM), and Adjacency List (AL). These results underpin the representation-sensitivity analysis of subsection 4.1 and the design of the heuristic selector in the GTA framework.

Table 20 tabulates the average accuracy for these task and graph type combinations, ordered alphabetically by task name. The accuracy figures represent the mean performance across all LLMs tested on that particular configuration. For each task–graph-type row, the highest accuracy is shown in bold.

The results in Table 20 show that no single input representation is universally optimal.

Natural Language (NL) achieves the highest average accuracy on sparse Bipartite Check, Bridge Count, Minimum Cycle, Maximum Clique, and Triangle Count.

Structured Language (SL) performs best on Tree Centroid, Tree Diameter, and Tree Max Independent Set. It also leads on sparse Minimum Spanning Tree and on Second MST at both densities.

Adjacency Matrix (AM) performs best on sparse Cycle Count, Eulerian Path at both densities, dense Maximum Clique, and Maximum Independent Set at both densities.

Table 20: Model-averaged accuracy (%) for 43 task–structure settings in GT Bench under four input representations (NL, SL, AM, AL). Bold values indicate the best representation within each row.

Task & Graph Type	**NL (%)**	**SL (%)**	**AM (%)**	**AL (%)**	**Best Repr.**
Biconnected Components (Dense)	67.61	59.75	**69.83**	69.03	AM
Biconnected Components (Sparse)	17.85	**23.31**	18.90	22.98	SL
Bipartite (Dense)	95.63	95.81	**97.64**	96.45	AM
Bipartite (Sparse)	**89.92**	83.60	75.91	77.02	NL
Bridge Count (Dense)	83.01	71.93	84.55	**88.23**	AL
Bridge Count (Sparse)	**42.91**	36.55	35.14	40.88	NL
Connectivity (Dense)	88.79	89.01	**90.65**	87.89	AM
Connectivity (Sparse)	87.33	87.95	60.52	**89.07**	AL
Cycle Count (Dense)	12.67	10.98	11.25	**13.01**	AL
Cycle Count (Sparse)	29.78	24.03	**31.84**	26.35	AM
Eulerian Circuit (Dense)	83.05	75.01	**95.03**	92.99	AM
Eulerian Circuit (Sparse)	99.75	**100.00**	98.98	99.81	SL
Eulerian Path (Dense)	71.10	75.05	**96.75**	93.88	AM
Eulerian Path (Sparse)	98.99	98.15	**100.00**	96.42	AM
Hamiltonian Circuit (Dense)	81.35	80.99	83.03	**88.27**	AL
Hamiltonian Circuit (Sparse)	83.01	86.18	82.75	**87.34**	AL
Hamiltonian Path (Dense)	84.48	81.89	84.71	**85.65**	AL
Hamiltonian Path (Sparse)	67.68	68.51	69.03	**71.97**	AL
Maximum Clique (Dense)	24.65	22.08	**38.61**	34.87	AM
Maximum Clique (Sparse)	**50.63**	43.72	46.88	43.41	NL
Maximum Flow (Dense)	15.21	16.95	**24.17**	22.93	AM
Maximum Flow (Sparse)	34.88	28.03	33.45	**39.56**	AL
Maximum Independent Set (Dense)	27.19	32.61	**36.93**	33.17	AM
Maximum Independent Set (Sparse)	36.03	35.75	**40.31**	31.48	AM
Min Cost Max Flow (Dense)	0.13	0.00	**0.98**	0.07	AM
Min Cost Max Flow (Sparse)	21.51	21.20	23.01	**23.45**	AL
Minimum Cut (Dense)	14.70	15.22	**19.03**	18.65	AM
Minimum Cut (Sparse)	40.28	36.88	39.15	**40.67**	AL
Minimum Cycle (Dense)	86.99	83.07	**87.35**	85.33	AM
Minimum Cycle (Sparse)	**81.42**	72.49	80.98	75.03	NL
Minimum Spanning Tree (Dense)	14.39	16.92	17.22	**19.81**	AL
Minimum Spanning Tree (Sparse)	29.77	**31.85**	24.91	29.79	SL
Second MST (Dense)	14.68	**19.83**	11.79	18.09	SL
Second MST (Sparse)	23.77	**24.15**	22.96	23.70	SL
Shortest Path (Dense)	49.41	43.73	51.99	**53.17**	AL
Shortest Path (Sparse)	71.92	69.95	63.97	**72.83**	AL
Spanning Tree Count (Dense)	12.11	16.05	**16.47**	12.69	AM
Spanning Tree Count (Sparse)	18.66	18.08	19.50	**19.85**	AL
Tree Centroid (Tree)	56.25	**60.84**	42.59	51.09	SL
Tree Diameter (Tree)	37.73	**42.05**	34.01	36.03	SL
Tree Max Independent Set (Tree)	33.99	**37.01**	24.92	36.58	SL
Triangle Count (Dense)	7.82	16.08	**23.29**	18.99	AM
Triangle Count (Sparse)	**40.33**	39.17	37.45	37.77	NL

Adjacency List (AL) leads on sparse Connectivity, Hamiltonian Path and Circuit at both densities, and Shortest Path at both densities. It also achieves the highest average accuracy on sparse Maximum Flow, Min Cost Max Flow, and Minimum Cut.

The spread between the best and worst representation frequently exceeds ten points within a single row, and the winning format changes from row to row; this per-task granularity underlies the aggregated analysis in subsection 4.1.

D Case Studies

To further illustrate the practical implications of input representation sensitivity and motivate the adaptive approach of GTA, we present several case studies. These examples show cases where the same LLM produces different outcomes on the identical graph problem instance, based solely on the input format used to represent the graph structure.

Our first comparison involves `GPT-4o-mini` tasked with determining connectivity on a sparse graph. As shown in Figure 3, when presented with the Natural Language (NL) description of the graph, the model incorrectly concludes the connectivity status between the specified nodes.

However, when the exact same graph instance and query are provided using the Adjacency List (AL) format (Figure 4), `GPT-4o-mini` successfully determines the correct connectivity. This suggests that the explicit listing of neighbors provided by AL facilitates the path-finding reasoning inherent in connectivity checks, which the model struggles to extract reliably from the more verbose NL description in this sparse setting.

Next, we examine `Llama-3.3-70B` attempting to find the centroid of a tree structure. Using the AL representation (Figure 5), the model fails to identify the correct centroid node(s).

In contrast, presenting the tree structure via Structured Language (SL), as depicted in Figure 6, enables `Llama-3.3-70B` to accurately compute the tree centroid. The templated, potentially hierarchical nature of the SL format seems better suited for representing tree structures in a way that facilitates reasoning about subtree sizes and properties, which are crucial for the centroid calculation. The flat neighbor listing provided by AL appears less conducive to this type of structural reasoning for tree-specific algorithms.

Our final case study features the highly capable `o3-mini` model tasked with counting cycles within a dense graph. When processing the NL description (Figure 7), the model makes an error in the cycle count.

Yet, when provided with the Adjacency Matrix (AM) representation for the same dense graph (Figure 8), `o3-mini` accurately calculates the number of cycles. For dense graphs, where the number of potential edges is high, the AM offers a complete and explicit view of all connections and non-connections. Having every entry explicit appears to help the combinatorial bookkeeping that cycle counting requires, sidestepping the parsing ambiguities of the NL description.

In none of these cases is the failure a lack of capability in the abstract: the same model solves the same instance once the structure arrives in a form matched to the task. This is the instance-level counterpart of the aggregate sensitivity documented in section 4, and it is precisely the behavior that GTA's Selector is trained to exploit.

Connectivity Sparse NL

Input: Please provide the reasoning process and the final answer directly to the question. Given the graph, determine if vertex 14 and vertex 3 are connected. This is an undirected graph with 17 vertices and 17 edges. The graph contains the following connections: There is an edge between vertex 13 and vertex 11. There is an edge between vertex 12 and vertex 11. There is an edge between vertex 7 and vertex 15. There is an edge between vertex 10 and vertex 12. There is an edge between vertex 6 and vertex 12. There is an edge between vertex 13 and vertex 10. There is an edge between vertex 2 and vertex 13. There is an edge between vertex 3 and vertex 1. There is an edge between vertex 1 and vertex 12. There is an edge between vertex 9 and vertex 5. There is an edge between vertex 9 and vertex 12. There is an edge between vertex 4 and vertex 14. There is an edge between vertex 5 and vertex 15. There is an edge between vertex 8 and vertex 3. There is an edge between vertex 2 and vertex 7. There is an edge between vertex 11 and vertex 16. There is an edge between vertex 8 and vertex 4

.

Response: To determine if vertex 14 and vertex 3 are connected in the given undirected graph, we will analyze the connections provided. 1. List the connections: - 13 - 11 - 12 - 11 - 7 - 15 - 10 - 12 - 6 - 12 - 13 - 10 - 2 - 13 - 3 - 1 - 1 - 12 - 9 - 5 - 9 - 12 - 4 - 14 - 5 - 15 - 8 - 3 - 2 - 7 - 11 - 16 - 8 - 4 2. Identify the vertices connected to vertex 3: - Vertex 3 is connected to vertex 1 (3 - 1). - Vertex 3 is also connected to vertex 8 (8 - 3). 3. Identify the vertices connected to vertex 14: - Vertex 14 is connected to vertex 4 (4 - 14). 4. Check for a path between vertex 3 and vertex 14: - From vertex 3, we can go to vertex 1. - From vertex 1, we can go to vertex 12 (1 - 12). - From vertex 12, we can go to vertex 10 (12 - 10). - From vertex 10, we can go to vertex 13 (10 - 13). - From vertex 13, we can go to vertex 2 (13 - 2). - From vertex 2, we can go to vertex 7 (2 - 7). - From vertex 7, we can go to vertex 15 (7 - 15). - From vertex 15, we can go to vertex 5 (15 - 5). - From vertex 5, we can go to vertex 9 (5 - 9). - From vertex 9, we can go to vertex 12 (9 - 12) - but this does not help us reach vertex 14. - We cannot find a direct or indirect path from vertex 3 to vertex 14 through the connections listed. 5. Conclusion: Since there is no path connecting vertex 3 to vertex 14 through the available edges, we conclude that vertex 14 and vertex 3 are not connected. Final Answer: Vertex 14 and vertex 3 are not connected.

Response by GPT-4o-mini 

Figure 3: GPT-4o-mini incorrectly solves Connectivity (Sparse) using NL input.

Connectivity Sparse AL

Input: Please provide the reasoning process and the final answer directly to the question. Given the graph, determine if vertex 14 and vertex 3 are connected. This is an undirected graph with 17 vertices and 17 edges. - Adjacency List: For each vertex, lists all connected vertices. Adjacency List: Vertex 1: 3, 12 Vertex 2: 13, 7 Vertex 3: 1, 8 Vertex 4: 14, 8 Vertex 5: 9, 15 Vertex 6: 12 Vertex 7: 15, 2 Vertex 8: 3, 4 Vertex 9: 5, 12 Vertex 10: 12, 13 Vertex 11: 13, 12, 16 Vertex 12: 11, 10, 6, 1, 9 Vertex 13: 11, 10, 2 Vertex 14: 4 Vertex 15: 7, 5 Vertex 16: 11 Vertex 17: No connections .

Response: To determine if vertex 14 and vertex 3 are connected in the given undirected graph, we can analyze the adjacency list provided. 1. Identify Connections: - Vertex 14 is connected to vertex 4. - Vertex 3 is connected to vertices 1 and 8. 2. Check for Path: - Since vertex 14 is connected to vertex 4, we need to check if vertex 4 can connect to vertex 3 through any other vertices. - Vertex 4 is connected to vertex 14 and vertex 8. - Vertex 8 is connected to vertex 3 and vertex 4. 3. Establishing Connection: - From vertex 14, we can go to vertex 4. - From vertex 4, we can go to vertex 8. - From vertex 8, we can go to vertex 3. Thus, there is a path from vertex 14 to vertex 3 through the vertices 4 and 8. Final Answer: Yes, vertex 14 and vertex 3 are connected.

Response by GPT-4o-mini 

Figure 4: GPT-4o-mini correctly solves the same Connectivity (Sparse) problem using AL input.

Tree Centroid Tree AL

Input: Please provide the reasoning process and the final answer directly to the question. Find the centroid of the given tree with the minimum index. A centroid is a node whose removal results in subtrees of size at most n/2. This is an undirected graph with 30 vertices and 29 edges. - Adjacency List: For each vertex, lists all connected vertices. Adjacency List: Vertex 1: 29, 7, 2, 25, 13, 24, 10, 8 Vertex 2: 1, 26 Vertex 3: 7, 28, 21, 12 Vertex 4: 28, 27 Vertex 5: 7, 22 Vertex 6: 7 Vertex 7: 1, 5, 3, 6, 30 Vertex 8: 1, 23 Vertex 9: 25, 16 Vertex 10: 1 Vertex 11: 21 Vertex 12: 3 Vertex 13: 1, 15, 14 Vertex 14: 13 Vertex 15: 13, 17 Vertex 16: 9 Vertex 17: 15 Vertex 18: 22 Vertex 19: 22 Vertex 20: 23 Vertex 21: 3, 11 Vertex 22: 5, 18, 19 Vertex 23: 8, 20 Vertex 24: 1 Vertex 25: 1, 9 Vertex 26: 2 Vertex 27: 4 Vertex 28: 3, 4 Vertex 29: 1 Vertex 30: 7

Response: [Omit part of the Response]Step 8: Evaluate node 1 as a potential centroid,Node 1 is connected to many other nodes, but its removal would indeed result in subtrees of varying sizes. The key is whether any of these subtrees exceed 15 nodes.,,## Step 9: Evaluate node 7 as a potential centroid,Node 7, being central and connected to nodes that themselves have multiple connections (like node 1), is a critical point. However, its removal might not result in subtrees that all have 15 nodes or fewer, especially considering the connections through node 1.,,## Step 10: Consider the tree's structure and the definition of a centroid,Given the tree's structure and the need for a centroid to have subtrees of 15 nodes or fewer upon its removal, we must carefully consider the impact of removing each highly connected node.,,## Step 11: Analyze the subtree sizes upon removal of key nodes,Upon closer inspection, removing node 1 or node 7 would indeed result in significant changes to the tree structure. However, we need to ensure that no subtree exceeds 15 nodes.,,## Step 12: Conclusion on the centroid with the minimum index,Considering the need for a centroid to divide the tree into subtrees of 15 nodes or fewer and looking for the node with the minimum index that satisfies this condition, we must identify the node that, when removed, best meets this criterion.,The final answer is: $\\boxed{7}$

Response by Llama-3.3-70B 

Figure 5: Llama-3.3-70B incorrectly solves Tree Centroid (Tree) using AL input.

Tree Centroid Tree SL

Input: Please provide the reasoning process and the final answer directly to the question.,,Find the centroid of the given tree with the minimum index. A centroid is a node whose removal results in subtrees of size at most n/2.,,Undirected Graph with 30 vertices and 29 edges,,Edges (Format: Node Node):,1 29,1 7,7 5,1 2,7 3,1 25,3 28,1 13,13 15,7 6,3 21,5 22,22 18,28 4,15 17,1 24,7 30,25 9,1 10,9 16,21 11,1 8,3 12,4 27,22 19,13 14,2 26,8 23,23 20,

Response: [Omit part of the Response Step 3: Calculate the maximum size of subtrees for a centroid To find the centroid, we need to ensure that when the centroid node is removed, each resulting subtree has a size of at most n/2. Since n = 30, n/2 = 15. Therefore, we are looking for a node that, when removed, leaves no subtree with more than 15 nodes. ## Step 4: Analyze the given edges to identify potential centroids Given the edges, we can start from any node and explore the tree to find nodes that could potentially be centroids. However, without drawing the tree or performing a detailed analysis of each node's connections, we can reason that a centroid is likely to be a node that is central and has a significant number of connections or is part of a central path in the tree. ## Step 5: Identify the node with the minimum index that could be a centroid The problem asks for the centroid with the minimum index. This implies we should start examining nodes from the lowest index upwards. Node 1 is connected to several other nodes (29, 7, 2, 25, 13, 24, 10, 8), making it a central node with a high degree. Its removal would likely result in subtrees that are smaller than half the size of the tree, given its central position and the distribution of its connections. ## Step 6: Confirm the choice of node 1 as a potential centroid Given the description and the connections provided, node 1 appears to be a critical central node. Its removal would split the tree into several smaller subtrees, each connected to node 1. Without a visual representation or further detailed analysis, it's reasonable to infer that node 1, due to its high connectivity and central role, could serve as a centroid that meets the condition of having subtrees of size at most n/2 when removed. The final answer is: $\\boxed{1}$

Response by Llama-3.3-70B

Figure 6: Llama-3.3-70B correctly solves the same Tree Centroid (Tree) problem using SL input.

Cycle Count Dense NL

Input: Please provide the reasoning process and the final answer directly to the question. Count the total number of simple cycles in the given graph. This is an undirected graph with 6 vertices and 11 edges. The graph contains the following connections: There is an edge between vertex 2 and vertex 5. There is an edge between vertex 5 and vertex 3. There is an edge between vertex 4 and vertex 2. There is an edge between vertex 6 and vertex 3. There is an edge between vertex 1 and vertex 3. There is an edge between vertex 2 and vertex 1. There is an edge between vertex 1 and vertex 5. There is an edge between vertex 5 and vertex 4. There is an edge between vertex 4 and vertex 3. There is an edge between vertex 5 and vertex 6. There is an edge between vertex 3 and vertex 2.

Response: [Omit part of the Response] 4. 6-cycles: There is also the 6–cycle that uses all the vertices. (Again – after checking the restrictions coming from missing edges such as 1–4, 1–6, 2–6, and 4–6 – one may "list" all possibilities and find that there are 4 different 6–cycles.)

Putting it Together

Summing over all lengths we have:
- Triangles: 8
- 4–cycles: 12
- 5–cycles: 13
- 6–cycles: 4

Total = 8 + 12 + 13 + 4 = 37 simple cycles.

Final Answer
The final answer is: 37 simple cycles.

Response by o3-mini 

Figure 7: o3-mini incorrectly solves Cycle Count (Dense) using NL input.

Cycle Count Dense AM

Input: Please provide the reasoning process and the final answer directly to the question. Count the total number of simple cycles in the given graph.This is an undirected graph with 6 vertices and 11 edges.
- Adjacency Matrix: 1 indicates an edge exists, 0 means no connection.
Adjacency Matrix (6×6):

```
    1  2  3  4  5  6
  ----------------------
1 | 0  1  1  0  1  0
2 | 1  0  1  1  1  0
3 | 1  1  0  1  1  1
4 | 0  1  1  0  1  0
5 | 1  1  1  1  0  1
6 | 0  0  1  0  1  0
```

Response: [Omit part of the Response] Step 5. Count 6-cycles (Hamiltonian cycles on all vertices). The full 6-vertex cycle, if the graph were complete, would be (6–1)!/2 = 60. However, note that vertex 6 in our graph is only adjacent to vertices 3 and 5. This forces that in any Hamiltonian cycle the two neighbors of 6 must be exactly 3 and 5. In other words, a 6-cycle "uses" 6 by "inserting" it between vertices 3 and 5. In fact, one may set up a one-to-one correspondence between a 6-cycle in the graph and a 5-cycle in the induced graph on vertices {1,2,3,4,5} that has 3 and 5 consecutive. A short enumeration shows that exactly 2 of the valid 5-cycles on {1,2,3,4,5} have 3 and 5 next to each other. (Any such 5-cycle "lifts" uniquely to a 6-cycle by replacing the edge 3–5 with the two edges 3–6 and 6–5.)
Thus there are 2 Hamiltonian (length-6) cycles.
Step 6. Add up all simple cycles.
Total cycles = triangles (3-cycles) + 4-cycles + 5-cycles + 6-cycles
= 8 + 12 + 10 + 2
= 32.

Final Answer: 32 simple cycles.

Response by o3-mini

Figure 8: o3-mini correctly solves the same Cycle Count (Dense) problem using AM input.

E Prompt Templates

This section details the prompt templates used for the core components of GTA: the Input Representation Selector, the Algorithm Generator, and the Algorithm Decomposer. Each prompt is designed to elicit a specific type of response from the LLM, guiding it through its designated sub-task.

Prompt for Input Representation Selector

Your Task: Optimal Graph Representation Selection
You will be given a graph theory problem description. Your sole responsibility is to analyze this problem and select the **single best representation format** that would make it easiest for a system to process and solve the problem algorithmically.
Available Representation Formats:

- `NL` (Natural Language: a descriptive text-based format)
- `SL` (Structured Language: a templated, keyword-based text format)
- `AM` (Adjacency Matrix: a matrix showing connections and/or weights)
- `AL` (Adjacency List: lists neighbors and/or weights for each vertex)

Considerations for your selection:

- **Problem Type:** What kind of graph question is being asked (e.g., finding paths, checking properties, calculating flows)?
- **Graph Characteristics (infer from description):**
  - Approximate size (number of nodes/edges).
  - Is it likely sparse (few edges) or dense (many edges)?
  - Are edge weights/capacities involved?
  - Is it directed or undirected?
- **Processing Efficiency:**
  - `AM` can be good for dense graphs or when checking non-connections is important.
  - `AL` is often efficient for sparse graphs and algorithms that traverse edges.
  - `NL` and `SL` might be suitable for smaller graphs or problems where the structure is simpler to describe textually, but could be harder to parse for complex algorithms.

Input Problem: `{question}`
Instruction: Review the input problem. Based on your analysis and the considerations above, output **only one** of the following four representation format codes: `NL`, `SL`, `AM`, or `AL`.
Your Output (must be one of NL, SL, AM, AL):

Figure 9: Prompt template for the Input Representation Selector component.

Prompt for Algorithm Generator

You are an Expert Algorithm Planner for Graph Theory Problems.
Your Task: You will be provided with a graph theory problem description. Your objective is to devise a **high-level strategic plan** or the **main algorithmic approach** that would be used to solve this problem.
Crucial Instructions:

- **DO NOT solve the problem.**
- **DO NOT derive the final answer.**
- **DO NOT write code.**

Your output should be a **conceptual outline** of the algorithm or the logical phases involved. Think of it as describing the *methodology* at a high level.
For example, if the problem were "Find the shortest path between node A and node B":

- A good high-level plan might be: "Utilize a breadth-first search (BFS) starting from node A, keeping track of distances, until node B is reached. Alternatively, if edges have weights, apply Dijkstra's algorithm."
- A bad (too detailed/executing) response would be: "1. Initialize distance to A as 0, all others as infinity. 2. Add A to queue. 3. While queue not empty..."

Considerations for your plan:

- Identify the core objective of the problem (e.g., finding a path, counting components, optimizing a value).
- What general class of graph algorithms is typically used for such a problem?
- What are the main conceptual stages of that algorithm?

Input Problem Description: `{question}`
Instruction: Based on the problem description, provide a concise, high-level algorithmic plan or strategy. Focus on the "what" and "why" of the approach, not the detailed "how."
Your High-Level Algorithmic Plan:

Figure 10: Prompt template for the Algorithm Generator component.

Prompt for Algorithm Decomposer

You are an Expert Algorithm Decomposer.
Your Task: You will be given an original graph theory problem description (`question`) and a high-level algorithmic plan (`plan`) designed to solve it. Your responsibility is to decompose this `plan` into a sequence of **2 or 3 highly detailed, specific, and actionable sub-steps**. These sub-steps should guide another system (an "Executor") to carry out the plan and solve the original `question`.
Crucial Instructions:

- **Extreme Detail Required:** Each sub-step must be very specific. Think about what data structures are needed, what values to initialize, what to iterate over, what conditions to check, what information to maintain or update at each stage.
- **Actionable by an LLM:** Phrase each sub-step as a clear instruction that an LLM could follow to perform a part of the algorithm.
- **Logical Sequence:** The sub-steps must follow a logical order that reflects the progression of the `plan`.
- **Coverage:** Together, the sub-steps should comprehensively cover the entire `plan`.
- **DO NOT solve the original problem or provide the final answer.** You are only creating the detailed execution blueprint.

Input:
1. **Original Problem (`question`):** `{question}`
2. **High-Level Algorithmic Plan (`plan`):** `{plan}`
Output Format (Strictly Adhere to This): You MUST output the decomposed sub-steps in the following exact format. Use `### Sub-step X:` for each step.

```
### Sub-step 1:
[Provide an extremely detailed instruction for the first phase of the algorithm based on the plan.
    Be specific about:
- Initializing any necessary data structures (e.g., distance arrays, visited sets, queues, stacks,
    flow matrices).
- Setting initial values for variables or nodes (e.g., source node distance to 0, all others to
    infinity).
- The very first set of operations to perform.]
### Sub-step 2:
[Provide an extremely detailed instruction for the second phase, logically following Sub-step 1. Be
     specific about:
- Iterative processes (e.g., "While the queue is not empty..." or "For each neighbor of the current
     node...").
- Conditions for updates or state changes (e.g., "If a shorter path is found..." or "If the
    capacity is greater than zero...").
- How to update data structures or values based on operations.
- What to do if a certain condition is met (e.g., target node reached, no more augmenting paths
    found).]
### Sub-step 3:
[Provide an extremely detailed instruction for the third and final phase, if necessary to complete
    the plan. Be specific about:
- Continuation of iterative processes.
- Termination conditions for the algorithm.
- How to derive any intermediate results needed for the final answer (e.g., "Sum the capacities of
    edges in the minimum cut set" or "Backtrack from the target node to reconstruct the path").
- Final checks or operations before the overall algorithm concludes.]
```

Instruction: Based on the provided `question` and `plan`, generate 2 or 3 highly detailed sub-steps strictly following the output format above.

Figure 11: Prompt template for the Algorithm Decomposer component.