GRAPHFINDER: FROM PROBLEM RECOGNITION TO ONTOLOGY-GROUNDED AND EXECUTABLE GRAPH FORMALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Many real-world optimization problems appear as unstructured natural-language descriptions that do not mention graphs explicitly, and it is often unclear whether they can be formalized as classical graph problems and solved by established graph algorithms. This paper asks whether LLMs can reliably transform such descriptions into executable, ontology-compatible graph formalizations. We propose GRAPHFINDER, which first determines whether a useful graph abstraction exists from a natural language optimization problem, then selects and instantiates a classical formulation from a curated graph problem ontology and returns an executable solver. The system uses typed slot filling to constrain formalization, treats verbal LLM verification only as advisory, and uses program execution plus machine-checkable output conditions as its acceptance gate. We explicitly distinguish executable success from semantic faithfulness and add a 100-instance equivalence tier with independent reference solvers. On the 520-problem GraphAbstraction-Bench we build, 55-way identification reaches 0.880–0.925 top-1 accuracy, while free-form executable end-to-end success is only 0.30–0.37. Ontology grounding with execution raises executable end-to-end success to 0.84 on the paired 50-problem subset and 0.715 on all 200 implicit optimization problems. On the 100-instance equivalence tier with concrete numeric data and independent reference solvers, reference-correct end-to-end success is 0.80 with predicted classes and 0.72 with oracle classes.

1 INTRODUCTION

Classical graph formalisms have produced both deep theory and practical algorithms for well-characterized combinatorial problems (Cormen et al., 2022; Diestel, 2017; Ahuja et al., 1993; Bengio et al., 2021; Cappart et al., 2023). Shortest-path computation is supported by algorithms with strong correctness and complexity guarantees (Cormen et al., 2022; Tarjan, 1983). Maximum-flow and minimum-cost-flow formulations similarly have mature combinatorial solvers (Ahuja et al., 1993; Cormen et al., 2022). Matching and assignment problems benefit from polynomial-time algorithms and a well-understood reduction to flow formulations (Cormen et al., 2022; Kuhn, 1955). Spanning-tree methods provide exact polynomial-time algorithms for connectivity problems (Cormen et al., 2022). The traveling salesman problem (TSP) illustrates both the strength and the limits of this classical toolkit: once its graph-theoretic formulation is known, its objective is precise, and a large body of exact algorithms, lower bounds, and approximation methods becomes available (Applegate et al., 2006; Diestel, 2017). These methods are widely deployed in logistics, transportation, telecommunications, scheduling, and network design (Toth and Vigo, 2014; Bengio et al., 2021; Cappart et al., 2023).

The practical difficulty is that a useful graph abstraction must be discovered before any of these algorithms can be applied. A TSP description typically mentions cities, pairwise travel distances, and a tour objective, but does not explicitly state that cities should become vertices, distances should become edge weights, and the desired object should become a Hamiltonian tour (Diestel, 2017; Applegate et al., 2006). Only after these semantic bindings are established can the classical TSP literature be used (Applegate et al., 2006; Diestel, 2017). Similarly, a logistics description involving warehouses, capacities, demands, routes, and transport costs becomes a minimum-cost flow problem

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

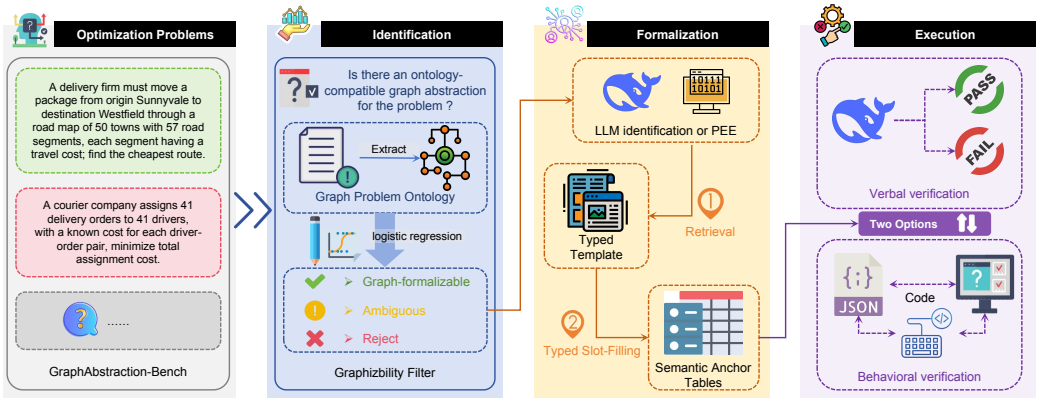

Figure 1: Overview of GraphFinder. A natural-language problem is first assessed for applicability, then matched to a graph problem ontology, instantiated through typed slots, and accepted only after generated code passes execution checks. Ambiguous or unsupported inputs are rejected or forwarded for later processing.

only after those domain elements have been mapped to supply nodes, demand nodes, directed edges, capacities, costs, and an objective (Ahuja et al., 1993; Bengio et al., 2021; Toth and Vigo, 2014).

The central challenge is therefore not merely recognizing graph terminology, but producing a formalization whose semantics and executable behavior remain faithful to the non-graph source problem. An LLM may know that a description resembles a graph problem, yet still select the wrong classical problem, omit constraints, add constraints that were never stated, or generate code that does not run. Existing LLM-based optimization methods typically assume that the target mathematical formalism is already known or allow the model to generate an unrestricted formulation without enforcing graph-specific structure or executable behavior (AhmadiTeshnizi et al., 2024; Ramamonjison et al., 2022; Huang et al., 2025; Li et al., 2023a). We formalize the task as three connected stages. *Identification* asks which classical graph problem, if any, applies; *formalization* asks how domain entities, relations, constraints, and objectives instantiate that problem; and *execution* asks whether a generated solver completes, returns a solution and objective, and can be checked against the formalization. Our central thesis is that, under our class graph problem ontology, *identification is substantially easier than faithful formalization; executable and semantically faithful instantiation remain the main bottleneck*. This distinction changes what a system must control. Instead of teaching an LLM more graph names, the system must constrain how the LLM formalizes a problem and use execution feedback to reject unreliable outputs.

To address these failure modes, we propose GRAPHFINDER, which decomposes graph abstraction into a grounded four-module pipeline in which each stage controls one of the challenges identified above. A *Graphizability Filter* decides whether the problem admits a useful graph abstraction compatible with a curated graph problem ontology. A retriever selects candidate formalisms using a structure-aware embedding based retrieval or direct LLM identification. *Typed slot-filling* then maps the domain problem into a fixed template, and a *Semantic Anchor Table* records the bidirectional mapping from problem concepts to graph elements. Finally, an *execution gate* turns the completed abstraction into a solver program, runs it and checks the result. Beyond the four core stages, GRAPHFINDER uses the Problem Essence Embedding (PEE) as a retrieval mechanism rather than the primary identifier. Direct LLM identification is stronger for exact selection, while PEE provides a deterministic and low-cost structural shortlist for reranking. Verbal LLM verification is likewise advisory and is not used to gate acceptance. Figure 1 illustrates the architecture.

These modules are designed to make each failure mode explicit and checkable. Unsuitable inputs are rejected before formalization, wrong formalism candidates remain constrained by the ontology, domain bindings are traceable, and non-executable results cannot pass acceptance. Our contributions are as follows.

- GRAPHFINDER, an end-to-end framework that decomposes natural-language graph abstraction into admission, retrieval, formalization, and executable acceptance. The key is to replace unconstrained

LLM generation with constrained decisions at each stage. A structure-aware retrieval layer supplies a deterministic candidate shortlist and complements direct LLM identification without replacing it.
- An empirical characterization of the identification-formalization gap together with a controlled grounding-by-verification analysis. The first result shows that recognizing a graph problem is far easier than instantiating it faithfully. The second shows that ontology-grounded formalization and execution-based verification are complementary while verbal self-verification is unreliable as a hard acceptance gate.
- GraphAbstraction-Bench, a benchmark designed to evaluate the full graph-abstraction pipeline rather than classification alone. A small-instance equivalence tier adds exact reference solutions for assessing semantic faithfulness. These resources provide a unified protocol for studying where LLM-based graph formalization fails.

2 Related Work

The related work is organized around the assumptions each line of work makes regarding graph availability and formalization validity.

LLMs for optimization and autoformalization. LLMs have been applied directly as optimizers that propose or refine solutions (Yang et al., 2024; Romera-Paredes et al., 2024) and as assistants that translate natural-language problems into MILP, LP, or solver-readable formulations (AhmadiTeshnizi et al., 2024; Ramamonjison et al., 2022; Yang et al., 2022). More recent systems train open-source OR models or build supply-chain assistants (Huang et al., 2025; Li et al., 2023a; 2024). Graph-specific LLM benchmarks primarily evaluate algorithmic reasoning or code generation on graphs that are already given (Wang et al., 2023; Wu et al., 2024; Xu et al., 2026), whereas GraphAbstraction-Bench evaluates the earlier abstraction decision itself. Autoformalization work translates informal mathematics into formal languages (Wu et al., 2022; Jiang et al., 2023), while self-refinement methods ask an LLM to critique and revise its own output (Madaan et al., 2023). These approaches either assume that the target formalism is already known or leave the LLM free to generate a broad mathematical structure. GraphFinder differs by restricting the target to a curated graph-problem ontology and separating formalism selection from typed domain binding.

Neural algorithmic reasoning and graph structure learning. Neural algorithmic reasoning trains networks to execute or emulate classical algorithms on an explicitly given graph (Veličković et al., 2024; Xu et al., 2019). Graph structure learning methods infer or refine edges when a node set already exists (You et al., 2020). Our problem is the inverse and earlier stage: decide whether a graph should exist, identify which classical graph problem it corresponds to, and infer the graph components from natural language. The forward and inverse problems therefore require different inductive biases and different failure checks.

Retrieval-augmented and analogical reasoning. Retrieval-augmented generation typically grounds factual or textual knowledge (Lewis et al., 2020; Izacard et al., 2023), while sentence encoders provide generic semantic similarity (Reimers and Gurevych, 2019). GraphFinder instead retrieves structured mathematical knowledge: each retrieved item contains a topological skeleton, a typed slot template, and canonical algorithms. This connects the system to analogical reasoning, where Structure-Mapping Theory emphasizes systematic relational mapping rather than attribute overlap (Gentner, 1983; Forbus et al., 2017; Webb et al., 2023). Grounding analogy in typed slots makes the relational mapping explicit and audit-ready.

Structured generation and tool use. Constrained decoding and grammar-guided generation enforce syntactically valid output (Willard and Louf, 2024; Dong et al., 2024), while tool use lets models call external programs (Schick et al., 2023). GraphFinder uses typed ontology templates as a semantic schema rather than only a syntactic grammar: the generated bindings must remain interpretable as domain-to-graph mappings and are passed to an executable solver. **Verification and execution feedback.** Self-refinement and reasoning-and-acting methods use LLM feedback to revise outputs (Madaan et al., 2023; Yao et al., 2023), while program-aided and code-emulator approaches generate or execute programs as part of reasoning (Gao et al., 2023; Li et al., 2023b). Reflexive agents and tool-interactive critics further use execution results to repair failed solutions (Shinn et al., 2023; Gou et al., 2024). They provide a useful contrast to GraphFinder: for a formalization whose ultimate acceptance criterion is executable behavior, programmatic execution is a direct behavioral

signal, whereas verbal critique is an indirect linguistic signal. Our calibration experiment measures this distinction directly and provides a negative result about using an LLM verifier as a hard gate.

These lines of work address related but partial aspects of the problem, as they either operate after a graph has been provided, generate broad mathematical formulations without graph-specific structure, or rely on linguistic feedback that is only indirectly connected to formalization correctness. In this paper, GRAPHFINDER complements them by jointly constraining applicability through a structured ontology, formalism selection by retaining explicit domain-to-graph mappings, and semantic instantiation by using program execution as a behavioral acceptance criterion. This formulation makes graph abstraction verifiable beyond surface semantic similarity and directly measures the remaining failure modes.

3 METHOD

This section formalizes natural-language graph abstraction and presents GRAPHFINDER. The core technical difficulty is that a single generation process must determine whether a graph formulation is useful, select the correct classical problem, instantiate domain-specific entities and constraints, and produce executable solver code.

GRAPHFINDER addresses this difficulty by decomposing the task into ontology-constrained identification, typed grounded formalization, and execution-based verification, so that each stage has a restricted output space and a machine-checkable failure signal. The remainder of this section maps each component to a stage. Problem Formulation and Graph Problem Ontology define the identification space, Failure Modes and Problem Essence Embedding describe retrieval, Graphizability Filter controls admission, Grounded Formalization instantiates the selected template, and Verification evaluates executability. An illustrative end-to-end logistics example is shown in Figure 2.

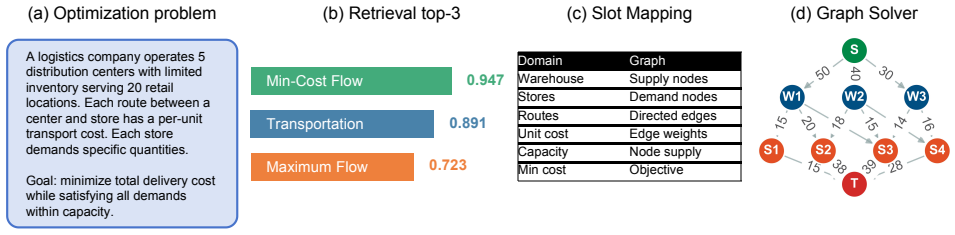

Figure 2: Illustrative end-to-end example: a logistics description is identified as minimum-cost flow, its domain entities are bound to typed graph slots, and a solver produces an executable flow solution.

3.1 PROBLEM FORMULATION

Let P be a natural-language optimization problem, possibly containing no graph terminology. Let $\mathcal{O} = \{o_1, \ldots, o_n\}$ be a curated ontology of n classical graph problems, where each entry $o \in \mathcal{O}$ includes a topological skeleton $\mathcal{S}_o$, a typed slot-filling template $\mathcal{T}_o$, and admissible canonical algorithms. We model the full task as producing a triplet

$$(P) \mapsto (o, \mathcal{G}, C), \tag{1}$$

where $o \in \mathcal{O}$ is the identified classical problem, $\mathcal{G}$ is a graph-theoretic formalization, and C is an executable solver program derived from $(o, \mathcal{G})$.

We separate three acceptance levels rather than collapsing them into a single success condition. A formalization is *schema-valid* when its output obeys the selected ontology template. It is *executable* when the generated program completes and returns a solution and objective. It is *faithful* when it also preserves the problem entities, constraints, and objective, which we write as

$$\text{Faithful}(P, \mathcal{G}) = E(P, \mathcal{G}) \wedge C(P, \mathcal{G}) \wedge O(P, \mathcal{G}) \wedge F(P, \mathcal{G}), \tag{2}$$

where E, C, O, and F denote entity preservation, constraint preservation, objective preservation, and feasibility-semantics preservation, respectively. Program execution can verify executability directly, but faithfulness additionally requires semantic checks or an independent reference solver.

The three stages are: (1) Identification maps P to an ontology entry o or to NONE, with success measured against the annotated class or family. (2) Formalization maps (P, o) to a graph abstraction $\mathcal{G}$ and a Semantic Anchor Table $\mathcal{A}$ recording the correspondence between problem concepts and graph elements. (3) Execution maps $(o, \mathcal{G})$ to a solution representation s and objective value z, or to failure. A useful graph abstraction is not merely an encoding of P as a graph. It must use an ontology-compatible classical problem, preserve the stated constraints and objective, and support execution of a corresponding solver.

3.2 Failure Modes

Failure modes denote the concrete ways in which an LLM-generated abstraction can diverge from a correct and executable graph formalization. We make them explicit so that each design choice has a corresponding check. Four failure modes motivate the architecture. The first is applying a graph formulation to an unsuitable problem, which is addressed by an admission filter. The second is selecting the wrong ontology entry, which is addressed by retrieval over the structured graph-problem ontology. The third is mapping domain concepts to the wrong graph elements or omitting constraints, which is addressed by typed slot-filling and Semantic Anchor Tables. The fourth is accepting a formalization that cannot be executed, which is addressed by code generation followed by behavioral checks. At runtime the problem flows through admission filtering, retrieval, grounded formalization, code generation, and execution in that order.

3.3 Graph Problem Ontology

The ontology contains $|\mathcal{O}|$ classical graph problems organized into five families for evaluation: Path, Flow, Matching and Assignment, Covering and Partitioning, and Touring and Routing. The ontology is flat for retrieval but family-structured for evaluation. Each entry in the ontology records a topological skeleton, a canonical description, a core objective, one to three key algorithms with complexity, a typed slot-filling template, representative application domains, and problem complexity. These entries are curated from standard graph-theory and algorithms texts (Cormen et al., 2022; Diestel, 2017; Ahuja et al., 1993), expanded with LLM assistance, and checked by a graph-theory expert. Appendix A records the ontology construction protocol.

The ontology is the main grounding mechanism of GraphFinder. One can view the ontology as a structured knowledge graph in which each entry is a node with typed slots, and edges connect entries to their families, algorithms, and application domains. It reduces a potentially unbounded LLM formalization space to $|\mathcal{O}|$ named mathematical structures, each with explicit topological assumptions and algorithmic semantics. Identification becomes selection from a finite structured set, and formalization becomes binding domain entities to typed slots rather than inventing a new formalism.

3.4 Graphizability Filter

The graphizability filter asks whether P admits a *useful, ontology-compatible* graph abstraction, not whether P can be encoded as some arbitrary graph. It first extracts entities, relations, constraints, and the objective for the graph optimization problem. It then computes four descriptive features: the density of extracted binary relations, a proxy for transitive relational structure, the proportion of constraints stated over pairs of entities, and the degree to which the objective can be expressed as local node or edge terms. A logistic regression maps these features to a graphizability score $G(P) \in [0, 1]$.

The features are descriptive rather than necessary graph conditions. Sparse graphs are still graphs, and transitivity is not required by shortest path, matching, or maximum flow. The filter emits four outcomes: graph-formalizable when $G(P) \geq 0.5$, ambiguous when $0.3 \leq G(P) < 0.5$ and processing continues with a warning, unsupported when the problem is graph-like but outside the ontology and GraphFinder abstains, and non-graph when $G(P) < 0.3$ and the input is rejected. The *unsupported* outcome is not determined by $G(P)$ alone. When $G(P) \geq 0.3$, an ontology-compatibility check compares the extracted entities, relations, constraints, and objective against the typed slots of candidate entries. GraphFinder emits *unsupported* if at least one mandatory requirement cannot be represented without changing its semantics, such as a global multiplicative coupling or a compositional objective formed from multiple ontology entries. The rule is deliberately conservative:

treating an ontology-incompatible graph-like input as formalizable would create a plausible but unfaithful mapping, whereas abstention makes the limitation observable.

3.5 Problem Essence Embedding

Identification can be performed directly by an LLM, but full-ontology prompts are costly and generic semantic encoders do not represent topological similarity well. PEE is therefore introduced at the retrieval stage to provide a compact structure-aware ranking of ontology entries. It is a fine-tuned dual encoder mapping natural-language problems into $\mathbb{R}^d$, where d is the embedding dimension, obtained by projecting representations from a text encoder (Reimers and Gurevych, 2019).

Training protocol. The training objective makes cosine similarity reflect topological structure rather than surface vocabulary. An anchor is encoded as

$$\texttt{[SKELETON] [SEP]}\ \mathrm{description}(o), \tag{3}$$

where `[SKELETON]` denotes the topological skeleton of entry o, `[SEP]` separates that structural information from the entry description, and $\mathrm{description}(o)$ denotes the canonical natural-language description of o. while positives are skeleton-free domain-language descriptions of the same problem class. Hard negatives belong to the same family but a different class, while easy negatives belong to different families. Queries at inference time are skeleton-free, matching the positive distribution and preventing test-time access to privileged structure. The loss combines supervised contrastive learning (Khosla et al., 2020) with MultiSimilarity loss (Wang et al., 2019) as

$$\mathcal{L} = \mathcal{L}_{\mathrm{SupCon}} + \lambda\, \mathcal{L}_{\mathrm{MultiSimilarity}}. \tag{4}$$

The scalar λ controls the relative weight of the MultiSimilarity term. Its value, the temperature, the embedding dimension, and all remaining training hyperparameters are reported in Appendix A.

Inference and role. Ontology entries are encoded in advance using their skeleton-conditioned descriptions. A query P is encoded without a skeleton prefix, and the top-k entries are returned by cosine similarity. PEE is evaluated as an efficient structure-aware candidate generator rather than as the primary identifier. Its three evaluation properties are structural awareness under frozen-encoder and lexical controls, retrieval utility, and accuracy-cost complementarity with an LLM reranker.

3.6 Grounded Formalization

Grounded formalization instantiates the selected ontology entry in three steps. Retrieval selects a typed template, typed slot filling binds domain concepts to its slots, and the Semantic Anchor Table records the resulting bidirectional mapping. This sequence constrains identification and makes downstream solver output traceable.

Retrieval. A query is embedded with PEE or named by an LLM identifier. The selected ontology entry supplies a typed template $\mathcal{T}_o$. For example, a minimum-cost flow template may contain slots for supply entities, demand entities, directed transport relations, capacities, unit costs, and the objective.

Typed slot-filling. The LLM fills each slot with concrete elements of P, preferably using exact problem text, and returns JSON. The LLM chooses *bindings*, not a new formalism. A completed mapping is stored in the *Semantic Anchor Table*, a bidirectional map between problem concepts and graph elements. For instance, a warehouse maps to a supply node, a delivery point to a demand node, a route to a directed edge, and a transport rate to an edge cost. This table keeps downstream decisions traceable and allows graph solutions to be translated back into domain language.

What grounding buys and costs. Grounding provides identification discipline by forcing the answer into $\mathcal{O}$ and reduces unstated extra freedom, though it does not by itself guarantee constraint coverage. Its current cost is constraint coverage: fixed templates can underrepresent descriptions with many specialized constraints.

3.7 Verification Signals

Verification can be verbal or behavioral. Verbal verification asks an LLM to judge whether the stated constraints are represented, while behavioral verification runs the generated program and checks

machine-readable output. GraphFinder treats these signals separately in order to determine which one should control acceptance.

Verbal verification. An optional LLM verifier checks whether each original constraint is represented, correctly translated, and free of extra freedom. It returns PASS or FAIL with issues. On FAIL, the mapper may receive one repair round using those issues as feedback.

Behavioral verification. The abstraction is converted into a self-contained code program running in a sandboxed subprocess with a fixed timeout. The current execution gate is deliberately limited to machine-checkable signals. It requires a zero return code, a final JSON object containing a solution and an objective, and completion within a fixed timeout. This proves executability, not semantic faithfulness. On failure, GRAPHFINDER may re-abstract within bounded rounds.

Selecting the acceptance signal. The choice of which signal should gate acceptance is treated as an experimental question rather than a design assumption. The factorial and calibration analyses in Section 4 show that verbal PASS/FAIL is largely uninformative about execution success. GraphFinder therefore uses the execution gate as the acceptance signal and verbal feedback only as an optional repair signal. The selected configuration consists of LLM identification, typed grounding, and execution-based acceptance.

4 EXPERIMENTS

The experiments address three questions: (1) whether identification and formalization are distinct difficulties, (2) which combination of grounding and verification yields executable formalizations, and (3) how the system behaves when the correct abstraction is absent or obscured.

We organize these questions into several experiment groups. The main text presents E1–E6 together with their principal result tables, while the supplementary analyses (E7–E9) are presented in Appendix B.

4.1 E1: IDENTIFICATION IS SUBSTANTIALLY EASIER THAN FORMALIZATION

Table 1: The same LLM names formalisms accurately, but free-form end-to-end success collapses; ontology grounding with execution restores executable success.

System	E2E
DeepSeek identification	—
FF + execution gate ($n = 200$)	0.37
GND + execution gate ($n = 200$)	0.715
Oracle-Class + execution gate ($n = 200$)	0.825

This experiment measures 55-way identification accuracy, free-form executable E2E, and the failure distribution of the grounded route. The headline comparison is shown in Table 1. Its most important result is that identification reaches 0.880–0.925 while free-form E2E is only 0.30–0.37, because recognizing a graph problem does not automatically produce a schema-valid and executable formalization. Grounding plus the execution gate raises E2E to 0.84/0.715, mainly because the ontology constrains identification and the gate removes non-runnable outputs.

4.2 E2: GROUNDING AND EXECUTION VERIFICATION ARE COMPLEMENTARY

This experiment tests how grounding and execution verification combine. The unified comparison results in Table 2 show that grounding adds only $+0.02$ under verbal verification but $+0.48$ under execution gating. This happens because the execution gate provides a direct behavioral signal, whereas the verbal verifier rejects many formalizations without providing useful information about whether the generated program can run. Strong prompt baselines do not close the gap, showing that the architectural constraint is more important than additional prompting.

Table 2: Unified comparison of LLM-only, retrieval-augmented, grounded, and oracle routes on GA-Implicit.

Route	Identification	Executable E2E
FF + verbal verification ($n = 50$)	0.48	0.30
FF + execution ($n = 200$)	0.415	0.370
BM25+ + GND + execution ($n = 200$)	0.430	0.365
PEE + GND + execution ($n = 200$)	0.310	0.235
GND (LLM) + execution ($n = 200$)	0.880	0.715
GND (LLM) + filter + execution ($n = 200$)	0.830	0.660
Oracle-Class + execution ($n = 200$)	1.000	0.825

Table 3: Free-form formalizations are richer but less disciplined; grounded formalizations preserve objective fidelity and avoid extra freedom but undercount constraints.

Rubric	Grounded	Free-form
Entity coverage	0.775	0.954
Constraint coverage	0.565	0.913
Objective fidelity	0.980	0.939
No extra freedom	0.980	0.847
Overall mean	0.825	0.913

4.3 E3: Semantic Formalization Quality

This experiment compares entity coverage, constraint coverage, objective fidelity, and unintended extra freedom between grounded and free-form formalizations. The scores are shown in Table 3. The important result is that free-form formalizations are semantically richer overall, but grounded formalizations preserve objective fidelity and avoid extra freedom better. This explains why the E2E advantage of grounding comes from identification discipline and executable reliability rather than from judged semantic completeness.

4.4 E4: PEE as a Structural Shortlist

Table 4: LLM identifiers dominate exact retrieval, while PEE improves over its frozen semantic encoder and provides a structural candidate generator. GA-Implicit, $n = 200$.

Method	P@1	P@3	P@5	MRR	Family P@1
BM25 (v1)	0.185	0.450	0.520	0.348	—
BM25+ (tokenized, skeleton-augmented)	0.430	0.655	0.735	0.548	0.615
Sentence-BERT (frozen)	0.170	0.380	0.505	0.337	—
PEE	0.310	0.520	0.755	0.486	0.715
DeepSeek zero-shot	0.880	0.960	0.970	0.917	1.000
DeepSeek 5-shot	0.925	0.985	0.995	0.954	1.000

PEE improves retrieval over frozen Sentence-BERT and supports a compact hybrid shortlist, while direct LLM identification remains stronger. Full metrics are in Table 4. PEE's contrastive training on skeleton-conditioned positives and same-family hard negatives encodes topological structure rather than surface vocabulary, which explains its retrieval gains over the frozen sentence encoder. However, its top-1 accuracy remains below direct LLM identification because exact 55-way discrimination also depends on the LLM's broader semantic knowledge and reasoning.

4.5 E5: Rejecting Unsuitable and Adversarial Problems

This experiment measures whether the admission filter can reject unsuitable and adversarial inputs without harming downstream graph formalization. On the admission evaluation set, the main gain is precision. The filter admits far fewer unsuitable problems while retaining comparable accuracy and

only a modest recall reduction. This matters because a false admission can later produce a runnable but unfaithful graph abstraction, so rejecting an unsuitable input is less damaging than silently forcing a formalization.

The filter improves admission precision and correctly abstains on adversarial inputs, but a hard threshold lowers downstream E2E from 0.715 to 0.525. The downstream and adversarial metrics are shown in Tables 5 and 6.

Table 5: Downstream filter ablation on GA-Implicit.

Configuration	**Accepted**	**Executable E2E**
No filter	200	0.715
Ambiguous forwarding (≥ 0.3)	183	0.650
Hard filter (≥ 0.5)	156	0.525

Table 6: Complete 60-item adversarial evaluation under an abstention-aware protocol.

Metric	**Value**
Overall exact / correct abstention	0.483 (29/60)
Top-1 on ontology-targeted entries	0.261 (6/23)
Family accuracy on ontology-targeted entries	0.261 (6/23)
Correct abstention on unsupported / non-graphable	1.000 (37/37)
Trap-fall rate	0.000 (0/60)

4.6 E6: Small-Instance Reference Equivalence

Table 7: Expanded 100-instance reference equivalence. A result is reference-correct only when it executes, matches the exact reference objective, and satisfies the problem-specific solution check.

Measure	**Predicted class**	**Oracle class**
Identification top-1	0.900 (90/100)	1.000
Execution success	0.870 (87/100)	0.770 (77/100)
Reference objective match	0.850 (85/100)	0.740 (74/100)
Reference-correct E2E	0.800 (80/100)	0.720 (72/100)
False acceptance above execution	0.080 (7/87)	0.065 (5/77)

This experiment measures machine-checkable reference equivalence on 100 concrete small instances, comparing identification, execution, objective match, and reference-correct E2E. The results are shown in Table 7. The important result is that reference-correct E2E is 0.80 with predicted classes and 0.72 with oracle classes, lower than execution-only success. The residual errors concentrate in flow and touring families because these problems require stronger constraint checking than the current execution gate provides.

5 Conclusion

In this paper, we study natural-language graph abstraction and identify a fundamental gap: LLMs recognize graph problems well but formalize them poorly. To address this, we propose GraphFinder, which turns graph abstraction into a constrained, verifiable pipeline: a graphizability filter decides whether a graph should exist, a curated graph-problem ontology restricts which classical problem is selected, typed slot-filling binds how domain entities map to graph elements, and an execution gate accepts a formalization only when it actually runs—replacing unreliable verbal self-verification with a direct behavioral signal. On GraphAbstraction-Bench, grounding plus execution gating raises executable E2E from 0.30–0.37 to 0.84/0.715, with reference-correct E2E of 0.80/0.72 on the 100-instance tier. We release GraphAbstraction-Bench to support further work on grounded graph formalization.

REFERENCES

Ali AhmadiTeshnizi, Wenzhi Gao, and Madeleine Udell. OptiMUS: Scalable optimization modeling with (MI)LP solvers and large language models. In *International Conference on Machine Learning*, 2024.

Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

David L. Applegate, Robert E. Bixby, Vasek Chvatal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

John E. Beasley. OR-Library: Distributing test problems by electronic mail. *Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24:1–61, 2023.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

Reinhard Diestel. *Graph Theory*. Springer, 5th edition, 2017.

Yixin Dong, Charlie F. Ruan, Yaxing Cai, Ruihang Lai, Ziyi Xu, Yilong Zhao, and Tianqi Chen. XGrammar: Flexible and efficient structured generation engine for large language models. arXiv preprint arXiv:2411.15100, 2024.

Kenneth D. Forbus, Ronald W. Ferguson, Andrew Lovett, and Dedre Gentner. Extending SME to handle large-scale cognitive modeling. *Cognitive Science*, 41(5):1152–1201, 2017.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. In *International Conference on Machine Learning*, 2023.

Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7(2): 155–170, 1983.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. In *Mathematical Programming Computation*, 2021.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, 2024.

Chenyu Huang, Zhengyang Tang, Shixi Hu, Ruoqing Jiang, Xin Zheng, Dongdong Ge, Benyou Wang, and Zizhuo Wang. ORLM: A customizable framework in training large models for automated optimization modeling. *Operations Research*, 2025.

Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research*, 24:1–43, 2023.

Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Timothée Lacroix, Jiacheng Liu, Wenda Li, Mateja Jamnik, Guillaume Lample, and Yuhuai Wu. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations*, 2023.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. In *Advances in Neural Information Processing Systems*, 2020.

Harold W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1–2):83–97, 1955.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, 2020.

Beibin Li, Konstantina Mellou, Bo Zhang, Jeevan Pathuri, and Ishai Menache. Large language models for supply chain optimization. arXiv preprint arXiv:2307.03875, 2023a.

Chengshu Li, Jacky Liang, Andy Zeng, Xinyun Chen, Karol Hausman, Dorsa Sadigh, Sergey Levine, Li Fei-Fei, Fei Xia, and Brian Ichter. Chain of code: Reasoning with a language model-augmented code emulator. In *Advances in Neural Information Processing Systems*, 2023b.

Sirui Li, Janardhan Kulkarni, Ishai Menache, Cathy Wu, and Beibin Li. Towards foundation models for mixed integer linear programming. arXiv preprint arXiv:2410.08288, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Rindranirina Ramamonjison, Haley Li, Timothy T. Yu, Shiqi He, Vishnu Raghunathan, Aditya Vora, et al. Augmenting operations research with auto-formulation of optimization models from problem descriptions. In *Findings of the Association for Computational Linguistics: EMNLP*, 2022.

Nils Reimers and Iryna Gurevych. Sentence-BERT: Sentence embeddings using siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 3982–3992, 2019.

Gerhard Reinelt. TSPLIB: A traveling salesman problem library. *ORSA Journal on Computing*, 3(4): 376–384, 1991.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Robert Endre Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

Paolo Toth and Daniele Vigo, editors. *Vehicle Routing: Problems, Methods, and Applications*. Society for Industrial and Applied Mathematics, 2nd edition, 2014.

Petar Veličković, Charles Blundell, et al. Neural algorithmic reasoning. *Patterns*, 5(7), 2024.

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*, 2023.

Xun Wang, Xintong Han, Weilin Huang, Dengke Dong, and Matthew R. Scott. Multi-similarity loss with general pair weighting for deep metric learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5022–5030, 2019.

Taylor Webb, Keith J. Holyoak, and Hongjing Lu. Emergent analogical reasoning in large language models. *Nature Human Behaviour*, 7:1526–1541, 2023.

Brandon T. Willard and Rémi Louf. Efficient guided generation for large language models. *Transactions on Machine Learning Research*, 2024.

Qiming Wu, Zichen Chen, Will Corcoran, Misha Sra, and Ambuj K. Singh. GraphEval36K: Benchmarking coding and reasoning capabilities of large language models on graph datasets. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems*, 2022.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.

Zixiang Xu, Yanbo Wang, Chenxi Wang, Lang Gao, Zirui Song, Yue Huang, Zhaorun Chen, Xiangliang Zhang, and Xiuying Chen. GTA: Graph theory agent and benchmark for algorithmic graph reasoning with LLMs. arXiv preprint arXiv:2609.12265, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *Advances in Neural Information Processing Systems*, 2024.

Zhiling Yang, Yuxin Wang, Yan Huang, Licong Cheng, et al. NL4Opt competition: Formulating optimization problems based on their natural language descriptions. In *NeurIPS Competition Track*, 2022.

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*, 2023.

Jiaxuan You, Rex Ying, and Jure Leskovec. Graph structure learning for robust graph neural networks. In *Advances in Neural Information Processing Systems*, 2020.

AI USE STATEMENT

We used generative AI tools for drafting and editing portions of the manuscript. All LLM calls used as experimental subjects or system components are part of the evaluated GraphFinder pipeline and are disclosed in the method and reproducibility details. All AI-assisted text, ontology entries, and code were reviewed and verified by the authors for correctness, provenance, and consistency with the reported experiments. The authors take responsibility for the final content of this work, including all text, claims, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human subjects or personally identifiable data. The benchmark is derived from public graph-algorithm libraries and synthesized domain-language problems; no private operational data is released. The main societal risk is inappropriate reliance on automated graph abstraction in high-stakes planning or decision settings. The Graphizability Filter, explicit abstention, family-level evaluation, and execution gate are designed to make failures and unsupported problems observable rather than silently forcing an incorrect graph formulation. The adversarial evaluation further documents that surface-level cues can mislead the system, which argues against unsupervised deployment in safety-critical applications.

REPRODUCIBILITY STATEMENT

This paper is reproducible. The supplementary package contains the GraphAbstraction-Bench data, prompts, evaluation code, trained PEE checkpoint, and raw result files. An anonymized implementation is available at `https://anonymous.4open.science/r/GraphFinder-2306`.

Appendix A records training hyperparameters, LLM-call settings, benchmark construction and isolation, evaluation protocols, and the server replication package. The main text states the sample sizes, evaluation metrics, statistical tests, and execution conditions used for each reported result.

A Reproducibility and Data

A.1 Implementation and Training

PEE training. The base encoder is `sentence-transformers/all-mpnet-base-v2` with a 128-dimensional projection. Training uses SupCon plus $0.3\times$MultiSimilarity loss, temperature 0.07, batch size 16, gradient accumulation 3, learning rate 2×10^{-5}, warmup 100 steps, bfloat16 AMP, up to 30 epochs, early-stopping patience 8, seed 42, and two RTX 4090 GPUs. Anchors are skeleton-prefixed; positives are skeleton-free and contain 825 examples; each sample uses four same-family hard negatives and two cross-family easy negatives.

A.2 LLM and Prompt Settings

LLM calls. All calls use `deepseek-chat`, temperature 0, four retries with exponential backoff, and the following token limits: filter 100, mapping and verification 1200–1500, code generation 2000. Prompts are available verbatim in the released experiment scripts.

A.3 Code Generation and Execution

Code generation and execution. The completed formalization and selected ontology entry are passed to a code-generation prompt that requires a self-contained Python program defining `solve()`; the program may use the standard library, NumPy, and NetworkX only, must not read files or use the network, and must print one final JSON object containing `solution` and `objective`. A fenced Python block is extracted from the response, written to a temporary file, and executed in a subprocess with a 20-second timeout. Execution success requires return code zero, a final JSON line, and both required fields. The main results use one code-generation attempt; repair-based strong prompt baselines are reported separately.

A.4 Graphizability Filter Details

Graphizability filter details. An LLM extraction prompt returns entities, binary relations, constraints, and the objective. Four features are computed: relation density $n_r/(n_e(n_e-1))$, a transitivity proxy $\min(0.3n_r/(n_e(n_e-1)) + 0.1, 0.8)$, the fraction of constraints containing pairwise markers, and an objective decomposability indicator. The score uses the repository-default logistic coefficients $\{2.5, -1.5, 3.0, 2.0, -0.8\}$, followed by a threshold sweep on 180 GA-Explicit and 20 GA-Negative examples; thresholds 0.3 and 0.5 define the ambiguous and hard branches. The unsupported check uses the score branch plus a compatibility check against the typed slots of the candidate ontology entries.

A.5 Ontology Construction Protocol

Ontology construction protocol. In this paper, we set $|\mathcal{O}| = 55$ to cover the most common graph problems in combinatorial optimization, but the ontology can be expanded or specialized for different domains. The ontology contains 55 classical graph problems organized into five families for evaluation: Path (11), Flow (9), Matching and Assignment (10), Covering and Partitioning (14), and Touring and Routing (11). Starting from classical graph-algorithms textbooks, one graph-theory expert listed canonical problem families and their defining constraints. An LLM proposed candidate entries and slot templates; the expert merged duplicates, rejected variants that were not classical graph problems, and corrected family labels. The resulting 55 entries were checked again against the textbook definitions and against the benchmark annotations.

A.6 Benchmark Construction and Isolation

Benchmark construction and isolation. GA-Implicit test templates are disjoint from the domain templates used to construct PEE training positives. Descriptions are assigned class, family, reference abstraction, and graphability labels. The construction and evaluation scripts are released with the benchmark. GA-Explicit draws from OR-Library (Beasley, 1990), TSPLIB (Reinelt, 1991), MIPLIB (Gleixner et al., 2021), and textbooks. GA-Implicit uses pure domain language generated from unseen domain templates, and its 200 test templates are disjoint from PEE training-positive templates. GA-Negative contains NLP and machine-learning tasks without a useful graph abstraction. GA-Adversarial contains 60 paired confusable descriptions. Annotation records class, family, and a reference abstraction, with inter-annotator agreement $\kappa = 0.87$. Evaluation defines exact-match accuracy against the annotated class. Family-tolerant accuracy credits a top-k candidate that belongs to the annotated family. For adversarial entries whose targets are outside the ontology, the correct ontology-constrained output is abstention, and naming the surface-trap class is a trap fall. End-to-end success is the conjunction of abstraction correctness, schema-valid formalization, and execution success. The 100-instance equivalence tier is excluded from the core 520 count because it carries concrete numeric instances and exact reference solutions rather than a class label.

Table 8: GraphAbstraction-Bench covers explicit graph problems, implicit domain-language problems, negative examples, and adversarial surface traps.

Subset	Size	Primary purpose
GA-Explicit	180	Problems whose graph terminology is already explicit
GA-Implicit	200	Domain-language problems requiring graph abstraction
GA-Negative	80	NLP/ML tasks without a useful graph abstraction
GA-Adversarial	60	Surface traps requiring correct class or abstention

A.7 Evaluation Protocols

Evaluation protocols. Paired configurations use the seed-42 50-problem subset; replication uses all 200 GA-Implicit problems. Execution success requires a zero subprocess return code, JSON output containing a solution and objective, and a 20-second timeout. End-to-end success for execution-gated configurations is abstraction correct and schema-valid and execution success. Structure correlation uses the 14-bit signature with Spearman rank correlation, average ranks, and two-sided normal-approximation p-values. Within-family correlation uses same-family pairs only. The verifier calibration joins verbal verdicts with execution outcomes on identical problems. McNemar tests use continuity-corrected chi-square statistics.

A.8 Server Replication Package

Server replication package. The released scripts rerun PEE top-k evaluation, hybrid reranking, frozen-SBERT structural controls, and PEE execution-gated end-to-end evaluation on all 200 problems. Model loading uses the exported checkpoint in an offline HuggingFace environment, with a configurable mirror endpoint if required.

B Supplementary Results by Experiment

B.1 E1 Supporting Details

Prompted 55-way classification reaches 0.880 to 0.925 top-1 with perfect family accuracy. All 24 zero-shot errors are within-family confusions, such as minimum cut versus minimum s-t cut and capacitated assignment versus assignment. The same LLM, when asked to formalize freely, names the correct class in only 41.5% of cases. Even a well-formed free-form formalization executes only 83.5% of the time, yielding only 30 to 37% end-to-end success. Grounding closes much of this gap: the typed template forces an ontology-consistent identification, and execution gating raises executable end-to-end success to 0.84 ($n = 50$) and 0.715 ($n = 200$, 95% Wilson interval $[0.649, 0.773]$).

Supplying the true class raises executable end-to-end success to 0.825 ($n = 200$), showing that identification errors explain part, but not all, of the remaining gap.

Table 9: Failure taxonomy for the grounded execution-gated route on GA-Implicit. The remaining failures are dominated by retrieval and runtime errors rather than schema violations.

Failure type	**Count**
Identification error	24
Execution runtime error	26
Execution timeout	6
Code generation returned no code	1
Schema invalid	0
Total failures	57

B.2 E2 Supporting Details

This comparison separates prompt engineering from architecture: all execution-gated routes share the same code-generation and execution protocol, and the large differences are driven by identification quality and typed grounding. We do not report OptiMUS or NL4Opt scores because their released pipelines target MILP-style outputs and different instance formats; a direct run would require an additional adaptation layer that is outside this paper's graph-schema evaluation.

Table 10: Grounding and execution verification improve end-to-end success mainly when combined, not when either mechanism is added alone. Paired 50-problem subset.

	+ LLM verification	**+ execution gate**
Free-form (FF)	0.30	0.36
Grounded (GND, LLM retrieval)	0.32	**0.84**

Holding verification fixed, grounding adds $+0.02$ under verbal verification but $+0.48$ under the execution gate. Holding grounding fixed, the execution gate adds $+0.06$ for free-form and $+0.52$ for grounded. Neither ingredient is strong alone; together they unlock each other, with an interaction of $+0.44$. Paired McNemar tests are significant: grounded+execution versus grounded+verification is 26:0 ($p < 10^{-4}$), and versus free-form+verification is 35:0 ($p < 10^{-4}$).

On all 200 GA-Implicit problems, retrieval quality orders end-to-end success monotonically: PEE top-1 $0.31 \rightarrow 0.235$, BM25 top-1 $0.43 \rightarrow 0.365$, free-form naming $0.415 \rightarrow 0.370$, and LLM identifier $0.88 \rightarrow 0.715$.

Table 11: Multi-seed replication of the grounded execution-gated route on three independent 50-problem samples.

Seed	**Identification**	**Formalization**	**Execution**	**E2E**
42	0.900	1.000	0.780	0.720
43	0.880	1.000	0.920	0.800
44	0.860	1.000	0.800	0.720
Mean	0.880	1.000	0.833	0.747

To check the stability of the paired E2 result, we reran the grounded execution-gated route on three independent 50-problem samples drawn with seeds 42, 43, and 44, keeping the same prompts, typed grounding, and execution protocol as E2. Identification ranges from 0.860 to 0.900, formalization is always successful, and execution success ranges from 0.780 to 0.920. Across these seeds, the mean executable E2E is 0.747 with standard deviation 0.038, and the observed values span 0.720–0.800. This is consistent with the full 200-problem estimate of 0.715; the seed-level variation is concentrated in execution outcomes rather than formalization, and it shows that a single paired 50-problem subset is not stable enough for a headline claim. The main-text E2 conclusion therefore relies on the full $n = 200$ result.

Table 12: The verbal verifier's PASS and FAIL groups have nearly identical execution success, so the verdict is a poor acceptance signal. Fifty paired problems.

Quantity	Value
Verifier pass rate	0.44 (22/50)
P(execution success | verify-pass)	0.909 (20/22)
P(execution success | verify-fail)	0.893 (25/28)
Repair rate (one round)	0.22
E2E with verifier as hard gate	0.32
E2E with execution gate only	0.84

The verifier accepts 22 and rejects 28 formalizations, but execution succeeds at nearly the same rate in both groups (0.909 versus 0.893). Its verdicts carry almost no information about executable success, while rejecting half of the pipeline. Using it as a hard gate reduces end-to-end success from 0.84 to 0.32 (McNemar 26:0, $p < 10^{-4}$).

Table 13: Strong prompt baselines on the same seed-42 20-problem subset. All routes use free-form formalization except the grounded reference; CoT, Self-Refine, and ReAct-style repair strengthen prompting but do not close the identification gap.

Route	Identification	Valid	Exec.	E2E
FF + execution (stored)	0.30	1.00	0.85	0.25
FF + CoT + execution	0.25	1.00	0.85	0.25
FF + Self-Refine + execution	0.20	0.95	0.55	0.15
FF + ReAct repair + execution	0.20	0.95	0.90	0.20
GND + execution (stored, same subset)	0.90	1.00	0.75	0.75

These small-sample controls separate prompt strength from architecture: reasoning, self-critique, and execution repair leave exact identification at 0.20–0.25 and E2E at 0.15–0.25, while grounded typed retrieval attains 0.90 and E2E 0.75 on the identical 20 problems. Full per-problem outputs and prompts are included in the released result package.

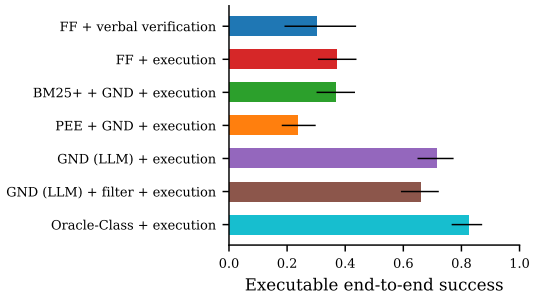

Figure 3: Unified baseline comparison with Wilson intervals.

B.3 E3 Supporting Discussion

Free-form formalizations enumerate more entities and constraints but add more unstated freedom and have undisciplined identification. Grounded formalizations preserve ontology compatibility, objective fidelity, and no-extra-freedom at high rates, but undercount constraints because the templates are still minimal. The judge favors free-form overall, so the end-to-end advantage of grounding comes from identification discipline and executable reliability, not from judged semantic completeness.

Table 14: A 20-candidate structure-aware shortlist nearly matches full-ontology accuracy while using a much shorter prompt. $n = 200$; token cost is estimated.

k	Top-1	Top-5	MRR	Prompt tokens (est.)
5	0.695	0.755	0.725	28.0k
10	0.780	0.890	0.832	47.6k
20	0.865	0.980	0.923	87.2k
55 (full ontology)	0.880	0.970	0.917	227.2k

Table 15: Training ablations for PEE. Contrastive training improves structural alignment, while the MultiSimilarity term improves retrieval and hybrid reranking. Skeleton conditioning does not improve retrieval, so its benefit is limited to the index representation.

Variant	P@1	P@5	MRR	Family P@1	Hybrid-20 Top-1
Frozen SBERT	0.170	0.505	0.337	—	—
SupCon only	0.290	0.690	0.474	0.735	0.80
No skeleton	0.345	0.775	0.515	0.740	0.84
Canonical (SupCon+MS)	0.310	0.755	0.486	0.715	0.86

B.4 E4 PEE Supporting Details

The contrastive objective, rather than the skeleton prefix alone, drives most structural alignment. Removing the skeleton prefix does not reduce retrieval but also does not retain the same ontology-index structure. MultiSimilarity contributes measurable gains in P@5 and hybrid top-1.

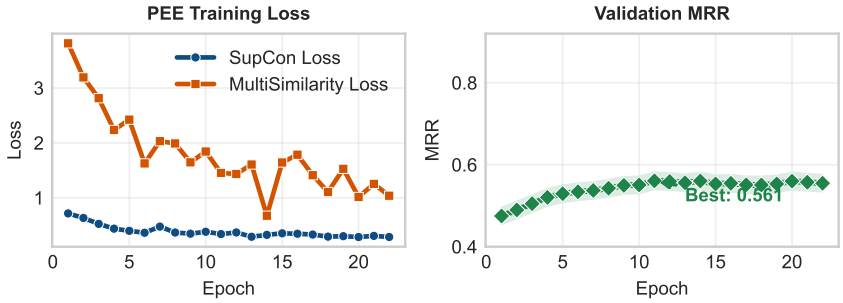

Figure 4: PEE training dynamics for the canonical configuration. Left: SupCon and MultiSimilarity losses over training. Right: validation MRR, which reaches 0.561 at epoch 11 and remains near that value before early stopping.

PEE reduces the retrieval prompt by about 60% and adds negligible amortized preparation cost, but it does not reduce the measured LLM latency. Monetary cost is expected to track prompt-token reductions but should be recomputed under the deployed model version and pricing.

For each class we derive a 14-bit structural signature covering directedness, weights, source/sink structure, capacities, bipartiteness, tours, matchings, flows, covering, paths, NP-hardness, maximization, all-pairs structure, and connectivity. PEE embedding distance correlates with signature Hamming distance substantially better than its frozen backbone (0.355 versus 0.146) and better than lexical overlap (0.164). The within-family correlation of 0.201 ($p = 10^{-10}$) is the key control: after removing family-indicator bits, PEE still tracks fine-grained structural differences. These results establish that skeleton-conditioned contrastive training injects structure into the representation rather than only vocabulary.

Table 16: The PEE shortlist trades a small local preparation cost for a substantial prompt-token reduction; wall-clock LLM latency is similar at the measured setting. Full-ontology and top-20 latency are averaged over 10 real calls; PEE timing is for all 200 queries on one local GPU.

Route	Prompt tok.	Completion tok.	LLM latency	Local prep
Full 55-way identification	1679	23	0.64 s	—
PEE top-20 + LLM rerank	663	25	0.69 s	0.13 s/200
BM25+ + LLM rerank	663	25	0.69 s	0.36 ms/query

Table 17: Structural-awareness evidence under identical pair protocols. Lower cosine distance corresponds to smaller structural Hamming distance.

Encoder / signal	ρ (all)	ρ (within family)	Same class	Cross family
PEE	**0.355**	**0.201**	0.004	0.363
Sentence-BERT (frozen)	0.146	0.038	0.050	0.615
Lexical overlap (BoW)	0.164	—	—	—

B.5 E5 Filter and Adversarial Details

The filter reaches 91.0% accuracy with 0.988 precision. Its missed graphable problems can be forwarded through the ambiguous branch, trading early rejection for later processing rather than hard failure.

The score-based filter rejects 44 of the 200 graphable implicit problems at the default threshold. Forwarding the ambiguous band recovers 27 of those problems and raises downstream success from 0.525 to 0.650, while the no-filter upper bound remains 0.715.

B.6 E6 Reference-Equivalence Details

Expanding the tier exposes residual error that the earlier 24-item pilot missed: seven predicted-class runs pass execution but fail the independent solution check, and five oracle-class runs do so. Flow and touring families account for most false acceptance, while path and covering instances remain well calibrated.

B.7 E7 Local Analyses

This experiment checks whether execution success is a sufficient signal of semantic acceptance. The central judge result is shown in Table 18; the other local analyses are reported in Tables 19, 20, and 21. The important result is that 90% of sampled execution successes are judged semantically incorrect,

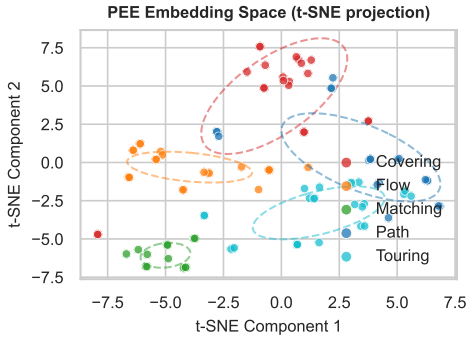

Figure 5: t-SNE projection of PEE representations, colored by annotated graph problem family. Fine-tuned PEE groups same-family descriptions while separating same-family hard negatives; the frozen-encoder and lexical controls in Table 17 bound the contribution of surface vocabulary to this geometry.

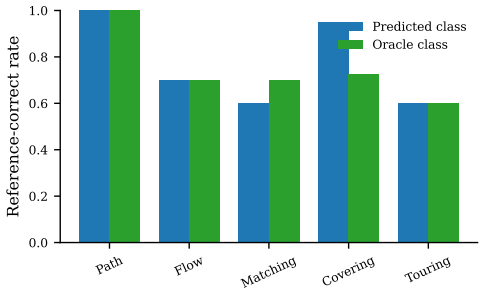

Figure 6: Reference-correct equivalence by ontology family.

because many GA-Implicit descriptions do not contain concrete numeric instance data and generated code often creates demonstrative instances. This finding makes the execution-semantic distinction central to the benchmark and motivates the machine-checkable E6 tier.

Table 18: A stricter full-evidence judge inspected generated code and outputs. Most execution-success samples were judged incorrect because GA-Implicit descriptions often lack concrete instance data.

Outcome	**Count**
Execution success	40/50
Judged incorrect	36/40
Judged correct	4/40
Judged unverifiable	0/40
Incorrect rate among judged successes	0.900

Table 19: Execution success is not equivalent to semantic acceptance. Rows join existing execution outcomes with independent LLM-judge scores on identical problems.

Formalization	**Exec.**	**False accept**	**Mean exec. ok**	**Mean exec. fail**
Grounded	0.36 (18/50)	0.167 (3/18)	0.896	0.785
Free-form	0.531 (26/49)	0.077 (2/26)	0.935	0.889

Some generated programs can execute while receiving only moderate semantic scores, especially in the grounded setting. This justifies treating abstraction correctness and semantic judgment as complements to the execution gate rather than replacements for it.

The clean full-evidence rerun removes code truncation and still confirms the benchmark limitation: 90% of sampled execution successes are judged incorrect. Many GA-Implicit problems do not contain concrete numeric instance data, so generated code often emits demonstrative solvers rather than a solver grounded in the stated instance. The secondary 0.5B local judge fails to emit structured verdicts and is not used as evidence; E6 replaces it with machine-checkable reference data.

The skeleton-index format yields the highest P@5 and family-level retrieval, while the plain-index plain-query format gives a slightly higher exact P@1. This supports keeping the skeleton prefix as a structural index signal rather than a direct query shortcut.

The held-out sample confirms that the LLM identifier transfers better than PEE at exact top-1, while PEE retains substantial family-level coverage. It controls template leakage but not the more general linguistic-distribution bias discussed in the Limitations section. On the local CPU, PEE encodes the 55-entry index and all 200 queries in approximately 1.33 seconds. The estimated prompt sizes for $k = 5, 10, 20$ are 26.8k, 46.4k, and 86.0k words, compared with 225.4k words for the full ontology; the $k = 20$ shortlist therefore uses approximately 38.2% of the full prompt length.

Table 20: Skeleton-conditioned ontology indexing improves P@5 and family-level retrieval compared with plain indexing. All results use the same trained PEE checkpoint on GA-Implicit ($n = 200$).

Index / query	P@1	P@3	P@5	P@20	Fam. P@1	MRR
Skeleton / unknown	0.310	0.520	0.755	0.980	0.715	0.486
Plain / plain	0.340	0.490	0.650	0.950	0.685	0.473
Plain / unknown	0.305	0.575	0.680	0.975	0.720	0.486

Table 21: Identification remains strong on a 100-problem template-disjoint held-out sample from the implicit benchmark; this is not a human-authored OOD tier. LLM predictions are the stored DeepSeek zero-shot outputs.

Method	P@1	P@3	P@5	Fam. P@1	Fam. P@5	MRR
PEE	0.310	0.540	0.760	0.720	0.920	0.494
DeepSeek zero-shot	0.900	0.980	0.990	1.000	1.000	0.935

B.8 E8 Multi-Model Replication

We rerun identification and the grounded execution route across DeepSeek, GLM-4.5-Air, Qwen2.5-7B, and two small local models. GLM-4.5-Air reaches comparable identification and executable E2E ($0.945/0.680$), while Qwen2.5-7B and the smaller models are substantially weaker. The result is therefore not model-independent, because exact 55-way identification and code generation are highly model-capacity dependent.

Table 22: Multi-model replication. Identification uses all 200 GA-Implicit problems. Executable E2E uses the seed-42 50-problem subset except DeepSeek, whose full 200-problem execution-gated result is shown.

Model	Identification P@1	Family P@1	Formalization	Executable E2E
DeepSeek	0.880	1.000	1.000	0.715
GLM-4.5-Air	0.945	0.985	1.000	0.680
Qwen2.5-7B	0.565	0.925	1.000	0.200
Qwen2.5-0.5B	0.000	0.105	0.640	0.000
Llama-3.2-1B	0.030	0.100	0.340	0.000

GLM-4.5-Air and DeepSeek are much stronger than the local checkpoints on exact identification and executable E2E. GLM shares only 5 of DeepSeek's 24 identification errors, so the two strong models have largely complementary failure sets. Qwen2.5-7B retains strong family-level identification but produces much weaker executable code, and the sub-billion and one-billion models fail the protocol.

B.9 E9 Structural Controls

This experiment checks whether PEE's structural signal is explained by permutation effects or surface vocabulary, using 5,000 label shuffles and 50 controlled paraphrases. The observed Spearman correlation of 0.355 remains far above the permutation null, and paraphrase changes only a small number of top-1 predictions; this supports structural awareness because skeleton-conditioned contrastive training captures topology beyond lexical overlap.

A 5,000-sample permutation test shifts structural labels among ontology classes while leaving the embedding space unchanged. The observed Spearman correlation between PEE distance and structural Hamming distance is 0.355, compared with a null mean of 0.045 and standard deviation 0.052, giving $p < 0.0002$. A controlled-paraphrase study rewrites domain vocabulary while preserving numbers and constraints; PEE retains its target at rank one for $28/50$ queries after paraphrasing, versus $32/50$ before paraphrasing.

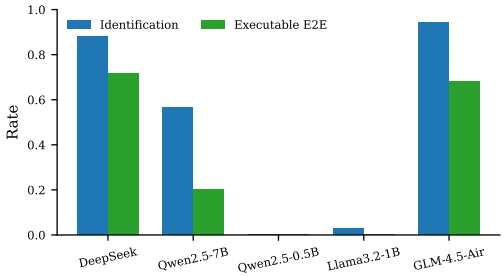

Figure 7: Multi-model identification and executable E2E.

Table 23: PEE structural controls. The permutation null is estimated over 5,000 label shuffles; paraphrase uses 50 domain-vocabulary rewrites.

Measure	**Value**
Observed Spearman ρ	0.355
Permutation null mean	0.045
Permutation p-value	< 0.0002
Lexical-overlap ρ	0.164
Original PEE top-1	0.320
Paraphrased PEE top-1	0.280

These controls show that the measured structure signal is not explained by label permutations or lexical overlap. Paraphrase changes four previously correct top-1 predictions (8% of the queries), so structure-aware retrieval is robust but not invariant to surface rewording.

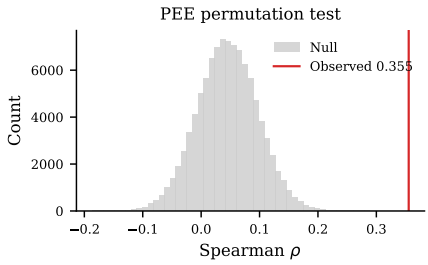

Figure 8: PEE structural-signal permutation null distribution.

C Discussion and Limitations

What the paper establishes. First, identification and formalization are distinct difficulties: naming is strong (0.880 to 0.925), while free-form executable end-to-end formalization is weak (0.30 to 0.37). Second, ontology grounding and execution verification are complementary for *executable* end-to-end success, jointly reaching $0.84/0.715$; on the 100-instance reference tier, reference-correct end-to-end success is 0.80 with predicted classes and 0.72 with oracle classes. Third, verbal self-verification is poorly calibrated as a hard acceptance gate for these programs. Fourth, PEE provides a secondary structure-aware retrieval layer whose shortlist substantially reduces prompt tokens, while its latency and exact top-1 accuracy are not superior to direct LLM identification. Fifth, the expanded 100-instance reference tier shows that the execution gate is substantially calibrated on path and covering instances but still admits solution-level errors in flow and touring problems. Sixth, training ablations show that contrastive learning improves structural alignment and that the MultiSimilarity term improves retrieval, whereas skeleton conditioning does not provide an additional retrieval gain. Seventh, the failure taxonomy shows that the remaining grounded-route failures concentrate in identification and runtime execution rather than schema validity. Eighth, permutation and controlled-

paraphrase tests support the structural claim; multi-seed replication gives a stable mean E2E of 0.747, and GLM-4.5-Air replicates the main identification and execution result.

Applicability and verification scope. GRAPHFINDER is best suited to combinatorial problems with a useful classical graph formulation; it is not a general mathematical formalizer. Execution success in the main benchmark means only that the generated program returns zero, emits JSON with a solution and objective, and finishes within the timeout. It does not by itself prove semantic equivalence between the original problem and the program. The E7 semantic-join analysis makes this boundary quantitative, and E6 provides a machine-checkable but small instance-level validation. The abstraction-correctness term in executable E2E guards against accepting a runnable but wrongly classified formalism, but it does not prove that the generated solver uses all stated constraints and objective terms.

Limitations.

- **Model scope.** The main results use `deepseek-chat`. GLM-4.5-Air reaches comparable identification and executable E2E, while Qwen2.5-7B and the small local checkpoints are substantially weaker.
- **Benchmark coverage.** The benchmark has 520 problems and includes template-generated implicit examples, but GA-Implicit generally contains no concrete numeric instance data, so its execution gate measures program completion rather than correctness on a particular instance. It also lacks a fully human-authored domain-level out-of-distribution tier.
- **Single-label annotation.** Some descriptions admit multiple defensible formalisms; family-level evaluation reduces, but does not remove, this ambiguity.
- **Execution-semantic gap.** The main execution gate checks return status, JSON shape, and timeout only, and it does not prove constraint satisfaction or equivalence. A clean full-evidence judge labels 90% of sampled execution successes as incorrect, mainly because many implicit problems lack concrete instance data. E6 bounds this gap on 100 synthetic instances but does not establish human-authored generalization.
- **Template coverage.** Grounded formalizations have judged constraint coverage of 0.565, showing that current typed templates underrepresent specialized constraints.
- **LLM-judge circularity.** E3 and E7 use an LLM judge only comparatively on identical inputs and do not interpret absolute scores as ground truth. Human expert annotation and human-vs-judge agreement remain open.

Future Experiments. (1) Expand the equivalence tier across more classes, larger instances, and models, and couple it with human-authored OOD problems. (2) Extend multi-model replication to larger open-weight models, additional seeds, and expert-annotated semantic evaluations. (3) Inject the structural signature directly into PEE metric learning and evaluate the paraphrase protocol at larger scale. (4) Support compositional formalizations $G_1 \oplus G_2$ and richer ontology templates to raise constraint coverage while preserving grounding.

Future Work. A human-authored, expert multi-label annotation protocol for a 50-problem semantic-quality subset remains planned future work and is not used as evidence in this submission. Completing that annotation, together with human-versus-LLM-judge agreement analysis, is a priority for the next revision.