Graph-V: A Training-Free Graph-Based Framework for Stepwise Reasoning Verification

Anonymous ACL submission

Abstract

Large Language Models (LLMs) now commonly employ chain-of-thought reasoning to solve complex problems, but evaluation usually stops at checking the final answer—leaving the correctness of intermediate steps underexplored. We introduce **Graph-V**[1], a general-purpose training-free framework for verifying the correctness of each step in a reasoning chain by leveraging a graph structured context. `Graph-V` operates in 3 stages. First, it employs an LLM to construct a logic dependency graph of the solution, explicitly capturing how each reasoning step depends on each other. Next, guided by this graph, each step is paired with a focused context that comprises its essential preceding steps (premises and other relevant dependencies) and a brief window of immediate succeeding steps. Finally, an off-the-shelf LLM judges the validity of each step based solely on this tailored context, isolating the step's inference from irrelevant details. This structured approach reduces distraction and sequential bias, while offering interpretability via explicit dependency links. Without training or fine-tuning, it remains efficient and adaptable by verifying only small subgraphs of the solution. Through rigorous evaluation on ProcessBench, `Graph-V` achieve competitive step verification performance among training-free methods, delivering significant increases in F1 across backbones. These results demonstrate that reasoning correctness can be reliably assessed through a dependency-aware context.

1 Introduction

Prompting LLMs to generate explicit step-by-step solutions – often called chain-of-thought (CoT) reasoning – has led to impressive gains in complex problem-solving across domains (Ahn et al., 2024; Srivatsa et al., 2023). However, most training and evaluation strategies still focus on the final answer as the sole indicator of success (Schulman et al., 2017; Shao et al., 2024). This leaves a critical blind spot: the validity of the intermediate reasoning steps is largely unchecked. A correct answer may result from flawed reasoning, while a single early error can derail an otherwise valid chain. Recent studies suggest that giving feedback or supervision at the step level can significantly improve model performance and reliability (Lightman et al., 2023; Zheng et al., 2025). From an evaluation standpoint, identifying which specific step is wrong in a reasoning chain is essential for diagnosing model failures and ensuring the safety and trustworthiness of LLM outputs (Jiang et al., 2025).

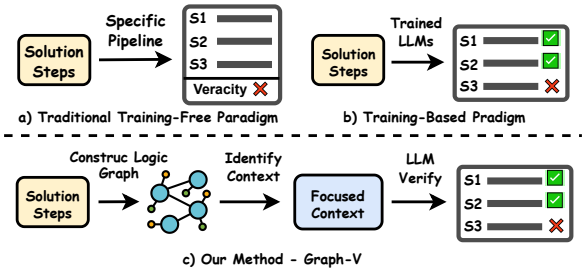

Figure 1: Reasoning verification paradigms: a) Traditional training-free methods only focus on verifying the whole chain; b) Training-based methods require computing resources and carefully crafted training datasets; c) Our approach Graph-V models the logic dependencies among reasoning steps within the solution, then identifies a focused context for a specific step which is finally fed to an LLM for veracity reasoning.

Despite the need, progress on step-wise reasoning verification remains limited. Human-annotated reasoning chains with step-level labels as the references are reliable but labor-intensive and unscalable, spurring interest in automated verification methods. Methods like ROSCOE (Golovneva et al., 2023), ReCEval (Prasad et al., 2023), Socreval (He et al., 2024b) use LLMs for reference-free evaluation without training, but only output a single chain-level score, lacking step-level detail (Figure

[1] https://anonymous.4open.science/r/Graph-V-5B2E

1a). In opposite, training-based methods (Zhang et al., 2025) offer detailed evaluations for each step but are resource-intensive and less adaptable due to their reliance on training data and computation (Figure 1b). In addition, verifying solution steps post-hoc poses challenges in selecting relevant context. Simply providing the full problem and solution to the verifier risks distraction by irrelevant details or bias from prior assessments. Those issues motivate a structured, step-wise verification approach. An initial attempt in this direction, PARC (Mukherjee et al., 2025), makes progress by augmenting each step with its premises, but it is limited: focusing only on immediate premises neglects broader logical structure, while omitting subsequent steps ignores forward-looking context crucial for understanding a step's purpose in the overall solution.

In this work, we propose a novel framework for post-hoc reasoning step verification, termed **`Graph-V`**, which leverages a solution's logical dependency graph to create relevant contexts. **`Graph-V`** (Figure 1c) operates in three phases: *i) Logic Graph Construction* a directed acyclic *logic graph* is constructed to explicitly identify the inter-step logical dependencies; *ii) Relevant Context Identification*, it traverses the logic graph to assemble a focused context for each step, comprising necessary preceding steps required for its derivation, and a short window of subsequent steps for forward-looking context; *iii) Step Correctness Verification*, an LLM judges the correctness of each step based solely on this dependency-informed context. By isolating each step with only its essential context, **`Graph-V`** mitigates distraction, reduces sequential bias, and enhances interpretability. This design also enables efficiency and scalability, as the verifier processes only small, focused contexts. Experiments on the long-form reasoning partition of the ProcessBench benchmark, **`Graph-V`** achieves state-of-the-art step verification accuracy among methods requiring no additional training.

In summary, our contributions are threefold:

1) We propose a training-free framework that decouples step verification into dependency-based context identification and correctness assessment.

2) We identify key factors for accurate verification–*stepwise evaluation*, *context pruning*, and *bidirectional context*–with enhanced LLM reliability.

3) **`Graph-V`** achieves competitive performance on ProcessBench and generalizes across LLM families, highlighting its effectiveness.

2 Related Work

Intermediate reasoning evaluation. Research on evaluating intermediate reasoning steps in LLMs has diverged into two primary approaches: training-based and training-free. The training-based approach, exemplified by process reward models (PRMs) (Lightman et al., 2023), relies on trained models to provide detailed evaluations by scoring each step in a reasoning trace. However, their dependence on training data and computational resources makes them resource-intensive and less adaptable. In contrast, the training-free approach offers greater flexibility and reduced resource demands. Recent methods like ROSCOE (Golovneva et al., 2023), ReCEval (Prasad et al., 2023), and Socreval (He et al., 2024b) leverage LLMs without additional training for reference-free evaluation. While capable of identifying errors, they typically output a single score for the full chain, lacking step-specific granularity. Additonally, (Ling et al., 2023) introduces the concept of "natural program", aiming to enhance the reliability of intermediate reasoning by providing verification along with the current reasoning steps as the generation progresses. The work of (Mukherjee et al., 2025) is a subsequent work aiming to verify steps from full reasoning chains by identifying logical premises for each step. Evaluation-wise, ProcessBench (Zheng et al., 2025) provides a benchmark for multi-step reasoning with human-annotated stepwise correctness labels in mathematical problem-solving. Our work aligns with the training-free paradigm, introducing a framework that generates the validation for each reasoning step.

Graph-Based Knowledge Representation. Graph structures are widely used in modern computing systems (Gregor and Lumsdaine, 2005). They provide a means of structuring knowledge, with nodes capture core concepts and edges denote relationships. Two common paradigms exist: i) *parametric* representation, where nodes and edges consist of vector representations of textual knowledge, then processed by models like GNNs (Scarselli et al., 2009) for downstream tasks (Yang et al., 2021; Ke et al., 2021); and ii) *textual* representation, where original text is retained within the graph's nodes and edges. This approach enables fine-grained reasoning by explicitly modeling dependencies and is often preferred when interpretability and controllability are crucial (Besta et al., 2024). We adopt this textual paradigm for our reasoning verification

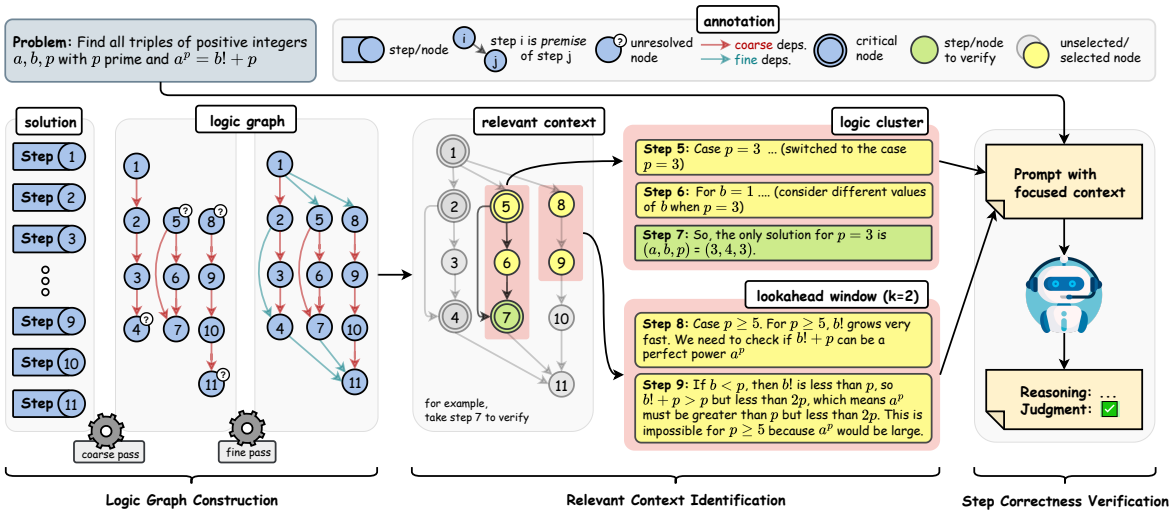

Figure 2: **The `Graph-V` operational flow.** A linear solution is first converted into a **logic graph** through a two-pass procedure to capture dependencies. To verify a target step (e.g., step 7), a focused context is assembled by extracting its **logic cluster** (the necessary preceding steps 5 and 6) and a **lookahead window** (subsequent steps 8 and 9). This minimal, dependency-informed context is then used to prompt an LLM for a final correctness judgment, avoiding the noise of the full solution.

framework by structuring reasoning steps into a graph with each node corresponds to one reasoning step and edges model the logical dependencies, providing a transparent and selective focus for the verification of each step.

3 Preliminaries

We focus on automatic verification of reasoning steps in mathematical solution write-ups. This section introduces the key definitions.

Reasoning Steps. A solution is modeled as a sequence of natural language reasoning steps: $S = \langle s_0, s_1, \ldots, s_{T-1} \rangle$, where each step s_t is generated by a solver for a given math problem Q.

Directed Acyclic Graph (DAG). A directed acyclic graph is defined as $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a set of nodes and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is a set of directed edges. The graph contains no directed cycles. For any edge $(u, v) \in \mathcal{E}$, node u is called a *predecessor* of v, and v is a *successor* of u.

Direct Dependence. A step s_j is a *premise* of a later step s_i ($j < i$) if the result of s_j is the immediate object of transformation in s_i. Conversely, s_i is a *dependent* of s_j. Importantly, this definition captures only direct dependencies: if s_k depends on s_j, and s_i depends on s_k, we do *not* consider s_j as premise of s_i.

Step Correctness. Following the taxonomy in (Zheng et al., 2025), a reasoning step is considered *correct* if it is free from mathematical, logical, conceptual, and completeness errors. We define a binary verification function: $V(s_i) \in \{0, 1\}$, which outputs 1 if step s_i is correct, and 0 otherwise.

4 Methodology

4.1 Overall Design

Given a question Q and a solution composed of sequential steps $S = \langle s_0, \ldots, s_{T-1} \rangle$, our goal is to produce a correctness verdict for each step, $\mathcal{V}_S = \langle V(s_0), \ldots, V(s_{T-1}) \rangle$, by modeling the distribution $p_\theta(\mathcal{V}_S | Q, S)$.

While a large language model (LLM) could be prompted to perform this task directly, its reliability is limited. The full context of Q and S is often noisy, containing logical paths and information irrelevant to verifying any single step. This "noisy context" makes it difficult for a model to focus on the immediate logical inference. We hypothesize that verification can be made more accurate by using a constrained, dependency-informed context.

This hypothesis requires us to first understand the solution's internal logic and then use that understanding to isolate the relevant information for each step. We formalize this by introducing two key intermediate structures: (1) a *logic graph*, $\mathcal{G}_S = (S, E)$, which is a directed acyclic graph capturing the logical flow of the solution. An edge $(s_j, s_i) \in E$ indicates that step s_j is a premise for step s_i; and (2) a set of *relevant contexts*, $\mathcal{C}_S = \langle C_0, \ldots, C_{T-1} \rangle$, where each C_i is a mini-

mal, focused context constructed specifically for verifying step s_i using the logic graph. By introducing these variables, we reframe the verification task as a structured, multi-stage process. The final veracity $\mathcal{V}_S^*$ is calculated by decomposing the target distribution $p_\theta(\mathcal{V}_S|Q, S)$ as follows:

$$\mathcal{V}_S^* \sim \sum_{\mathcal{C}_S, \mathcal{G}_S} \underbrace{p_\theta(\mathcal{V}_S|\mathcal{C}_S)}_{\substack{\text{Step}\\ \text{Correctness}\\ \text{Verification}}} \cdot \underbrace{p_\theta(\mathcal{C}_S|\mathcal{G}_S)}_{\substack{\text{Relevant}\\ \text{Context}\\ \text{Identification}}} \cdot \underbrace{p_\theta(\mathcal{G}_S|Q, S)}_{\substack{\text{Logic Graph}\\ \text{Construction}}}$$

This formulation decomposes the complex verification problem into three more tractable subtasks. We propose a training-free **Graph**-based Step Correctness **V**erification (**`Graph-V`**), a three-stage framework that realizes this decomposition. In fact, while Equation 1 defines the full distribution, our **`Graph-V`** framework approximates it by generating the most probable logic graph and relevant contexts in a deterministic pipeline in three stages:

Stage 1: Logic Graph Construction. First, we model $p_\theta(\mathcal{G}_S|Q, S)$ by constructing the logic graph $\mathcal{G}_S$ from the solution S. This involves employing a dedicated algorithm (section 4.2.1) to identify the premises for every step to map the complete dependency structure of the reasoning chain.

Stage 2: Relevant Context Identification. Next, for each step $s_i \in S$, we use the graph $\mathcal{G}_S$ to assemble a focused context C_i. This context is designed to provide all necessary local information and is defined as $C_i = \{\mathcal{F}_{pre}(s_i), s_i, \mathcal{F}_{post}(s_i)\}$. Here, $\mathcal{F}_{pre}(s_i)$ identifies crucial preceding steps, while $\mathcal{F}_{post}(s_i)$ identifies a small set of succeeding steps to provide forward-looking context. Crucially, the preceding context $\mathcal{F}_{pre}(s_i)$ is not limited to the direct parents of s_i in the graph but may include other relevant prior steps, as will be detailed later. This stage corresponds to modeling $p_\theta(\mathcal{C}_S|\mathcal{G}_S)$.

Stage 3: Step Correctness Verification. Finally, with the relevant context C_i prepared for each step s_i, we prompt an LLM to assess its correctness based *only* on this focused context. Assuming conditional independence between steps given their specific contexts, this final verification stage is formulated as:

$$p_\theta(\mathcal{V}_S|\mathcal{C}_S) = \prod_{i=0}^{T-1} p_\theta(V(s_i)|C_i) \qquad (1)$$

The following section will detail **`Graph-V`**, our proposed method for implementing each stage.

4.2 Graph-V Detail

4.2.1 Logic Graph Construction

To systematically identify relevant context, we construct a logic graph based on the hypothesis that most dependencies in a reasoning chain are *local*—that is, a step s_i is most likely to depend on its immediate predecessors ($s_{i-1}, s_{i-2}, \dots$). This locality assumption allows for an optimized premise search. We then identify these dependencies with an efficient two-pass, coarse-to-fine strategy:

Step 1: Solution Chunking. To create localized contexts for identifying dense dependency structures, we first partition the solution sequence $S = \langle s_0, \dots, s_{T-1} \rangle$ into a set of overlapping blocks $\mathcal{B}$. Each block b_i is a subset of S defined by a sliding window with size w and stride k:

$$\mathcal{B} = \{b_i = \langle s_{i \cdot k}, \dots, s_{i \cdot k + w - 1} \rangle\}_{i=0}^{\lfloor (T-w)/k \rfloor} \qquad (2)$$

Step 2: Two-Pass Dependency Identification. We then apply a two-pass procedure to establish the edges of the logic graph: **i) Coarse Pass** (Local Dependency Identification.) This pass aims to find all direct dependencies *within* each block. For each block $b_i \in \mathcal{B}$, we use a single LLM call to identify all internal *direct dependencies* (e.g., asking which steps in b_i are premises for other steps in b_i). Crucially, for each step, we also ask the LLM if the identified local premises are *sufficient* for verification. Steps are marked as `resolved` if their premise set is deemed complete, and `unresolved` otherwise. **ii) Fine Pass** (Long-Range Dependency Search.) This pass focuses only on the `unresolved` steps. For each `unresolved` step s_u residing in block b_i, we perform a targeted search for additional premises in every *other* block b_j (where $j < i$). This allows us to discover distant or non-local dependencies missed in the coarse pass. For example, we prompt the LLM: "Which steps in block b_j are required to verify step s_u?".

This two-pass procedure efficiently constructs the logic graph by first identifying dense local dependencies and then selectively searching for sparse, long-range ones, minimizing the number of required LLM calls. Figure 2 (left) provides a visual illustration of this process, where the coarse pass discovers local dependencies (red arrows) and the fine pass uncovers distant ones (blue arrows) to complete the graph.

4.2.2 Relevant Context Identification

Designing context functions $\mathcal{F}_{pre}$ and $\mathcal{F}_{post}$ is the central of building effective context for verification. Therefore, we introduce the concept of *logic clusters* and *critical nodes*. A *logic cluster* for a step s_i is the subgraph containing s_i and its ancestors, traced back through all to a set of closest *critical nodes*. We define critical nodes by first identifying what they are *not*: a step s_j is considered *non-critical* if it has exactly one dependent (i.e., it is a premise for only one step) and this dependent is its immediate successor, s_{j+1}. Such nodes represent a simple, linear flow of reasoning. Conversely, a *critical node* is any node that deviates from this simple linear flow, i.e., a node with at least one dependent that is not its immediate successor. This type of dependency marks long-range dependencies where a prior result is referenced later. These critical nodes function as semantic boundaries, ensuring the resulting logic cluster forms a self-contained unit sufficient for verifying step s_i. With this established, we define our context functions as follows:

1) Designing $\mathcal{F}_{pre}$: We define $\mathcal{F}_{pre}(s_i)$ as the function that retrieves the *logic cluster* for step s_i. This provides a context that is both more comprehensive than direct premises and more focused than the entire solution history, capturing necessary preceding steps for verification (section 5.2). For instance, as shown in Figure 2, the logic cluster for step 7 is the set {5, 6, 7}, as step 5 is its closest critical node.

2) Designing $\mathcal{F}_{post}$: While the logic cluster provides the necessary historical context, our analysis in section 5.2 shows that a *lookahead* is also crucial. This allows the LLM to understand a step's purpose within the broader reasoning path. Therefore, we define $\mathcal{F}_{post}(s_i)$ as a simple sliding window that returns the k *immediate steps following* s_i, where k is a tunable hyperparameter. Continuing with the example from Figure 2, a lookahead for step 7 with a window size of 2 would simply be steps {8, 9}.

4.2.3 Step Correctness Verification

Once the relevant context C_i for each step s_i is identified, the second sub-task is to determine the correctness of that step. We leverage a strong general-purpose LLM for this verification (Figure 2). For each step s_i, we provide the LLM with its focused context C_i (comprising its logic cluster and lookahead steps) along with the original question Q. Using a carefully designed prompt 6, we instruct the LLM to first generate a natural language justification j_i that explains its reasoning, and then produce a final correctness verdict $V(s_i)$. While this model-agnostic design allows for the integration of any state-of-the-art LLM for zero-shot verification, we opt for open-sourced models within the Llama and Qwen families for more control over inference. By conditioning the verdict on a focused context C_i and an explicit justification j_i, we ensure a more deliberate and transparent assessment.

4.2.4 Relation to Other Approaches

To contextualize our contribution, we analyze three baselines and their fundamental limitations, which `Graph-V` is designed to overcome.

1) All-at-once Verification. This approach, introduced in (Zheng et al., 2025), provides an LLM with the entire problem Q and solution S in a single prompt, tasking it to autoregressively verify all steps at once. The process is formally modeled by the chain rule:

$$p_\theta(\mathcal{V}_S|Q,S) = \prod_{i=0}^{T-1} p_\theta(V(s_i)|\mathcal{V}_{<i},Q,S) \quad (3)$$

While simple, this method is unreliable due to two failure modes: *i) sequential bias:* the verdict for one step is influenced by the verdicts of previous ones, causing the model to potentially overlook an isolated error later in the solution if most initial steps are correct; and *ii) attentional bias:* the model must discern relevant dependencies from noise if provided an unfiltered solution as context, increasing chances of overlooking critical premises in longer reasoning chains.

2) Stepwise Verification. This approach, a natural improvement from the all-at-once approach, processes each step independently to mitigate sequential bias. This leads to the formulation:

$$p_\theta(\mathcal{V}_S|\mathcal{C}_S) = \prod_{i=0}^{T-1} p_\theta(V(s_i)|Q,S') \quad (4)$$

where S' is either S or $S_{<i}$. While this independent evaluation solves the sequential bias, it still suffers from attentional bias, as the context provided to the LLM remains cluttered with irrelevant steps.

3) PARC Approach. The gist of this approach, proposed by (Mukherjee et al., 2025), is to reduce attentional bias by focusing context for each step s_i, consisting of the question Q and its set of immediate premises P_i:

$$p_\theta(\mathcal{V}_S|\mathcal{C}_S) = \prod_{i=0}^{T-1} p_\theta(V(s_i)|Q,P_i) \quad (5)$$

While this mitigates some bias, the context is often restrictive. By ignoring the broader logical flow

and how a step is used later, it may lack the necessary information for a robust verification.

The `Graph-V` Advantage. Our `Graph-V` framework overcomes the limitations of prior methods. Like Stepwise and Premises-based approaches, it verifies each step independently to eliminate sequential bias. Its key innovation is a superior solution to attentional bias. Instead of using a noisy full history or a restrictive premises-only context, `Graph-V` constructs a richer, more flexible context. By including crucial preceding steps ($\mathcal{F}_{pre}$) and forward-looking succeeding steps ($\mathcal{F}_{post}$), `Graph-V` provides both the backward justification and the forward purpose of each step. This leads to a robust, reliable, and logically-grounded verification.

5 Experiments

5.1 Experimental Setup

Dataset. We conduct experiments on the ProcessBench benchmark (Zheng et al., 2025), which comprises 3,400 step-by-step solutions with human-annotated verification to competition- and Olympiad-level math problems (drawn from GSM8K (Cobbe et al., 2021), MATH (Hendrycks et al., 2021), OlympiadBench (He et al., 2024a), and Omni-MATH (Gao et al., 2025)). To highlight the effectiveness of `Graph-V`'s dependency-aware pruning, we construct a hard subset by identifying examples that exhibit longer reasoning chains before the first error occurs. Specifically, for each of the three higher-difficulty datasets (MATH, OlympiadBench, Omni-MATH), we rank all examples based on the number of steps taken before an incorrect step is introduced and select the top 333 examples per dataset, resulting in 1,000 problems. Our primary evaluations are performed on this hard subset to emphasize the benefits of pruning in the presence of long and complex solution paths. We also construct an "easy" subset by sampling 250 examples from each partitions, yielding 1,000 problems to assess whether Graph-V maintains performance on simpler cases.

Metrics. Following (Zheng et al., 2025), we report: (i) *Error Accuracy*, the proportion of erroneous samples (original label $\neq$ -1) where the model correctly identifies the first faulty step; (ii) *Correct Accuracy*, the proportion of correct samples (original label =-1) where the model correctly predicts no error. We then compute the *F1 score* as the harmonic mean of these two accuracies.

Baselines. We compare `Graph-V` against four baselines. The All-at-once baseline verifies all steps autoregressively at once under the full context. The Stepwise–Prev. and Stepwise–Full baselines verifies steps individually, and use all preceding steps or the full solution respectively. PARC baseline (Mukherjee et al., 2025) tries to locate relevant context by only using steps' premises, which we will show to be inadequate for robust verification. In contrast, `Graph-V` addresses these limitations. It first constructs a logic graph using the algorithm from Section 4.2.1. From this graph, it then extracts a minimal, dependency-informed context for each step, composed of its logic cluster and subsequent steps. The justification for this configuration is analyzed in **(RQ2)** and **(RQ3)** in Section 5.2.

Implementation For a comprehensive evaluation, we implement baselines and our full `Graph-V` pipeline using two model series: Llama 3.3 (Meta, 2024), and Qwen 2.5 (Qwen, 2024). In the logic-graph construction stage, our method is configured in two phases: Phase 1 (coarse pass) uses a sliding window of size 5 with overlap step 1 for context preservation, and Phase 2 (fine pass) uses window size 10 to search for longer-range dependencies. For the ablation study, we evaluate two alternate graph-construction variants: a grouped construction variant that only runs Phase 1 (window size 5, overlap 2), and a targeted construction variant that directly runs Phase 2 (window size 10) without prior chunking. For the succeeding-context portion of step verification, we adopt a fixed lookahead of $k = 5$, i.e., the five immediate steps following the target step are included in the verification context. All methods share the prompts and computational settings to ensure fair comparison.

5.2 Results and Analysis

We present the main experimental results in this section, structured as research questions.

(RQ1): How does `Graph-V` perform against the baselines? Table 1 shows that `Graph-V` consistently outperforms all baselines across the three tested backbones on the long-reasoning partitions of ProcessBench. The All-at-once method performs worst, with a best average F1 of 51.0. Its tendency for high Correct accuracy but poor Error accuracy confirms a significant sequential bias, failing to identify isolated errors. The stepwise baselines mitigate this issue. Stepwise-Full consistently outperforms All-at-once, demonstrating the value

Backbone	Configuration	MATH			OlympiadBench			OmniMATH			Avg
		Error	Correct	F1	Error	Correct	F1	Error	Correct	F1	
Llama3.3 70B	All-at-once	35.5	94.5	51.6	22.5	93.0	36.2	26.5	90.2	40.9	42.9
	Stepwise - Prev.	41.8	76.7	54.1	26.7	47.4	34.1	32.9	52.1	40.4	42.9
	Stepwise - Full	44.5	83.0	58.0	34.2	73.7	46.7	42.4	73.6	53.8	52.8
	PARC	46.4	71.7	56.3	30.0	50.7	37.7	32.9	52.8	40.6	44.9
	Graph-V	49.1	87.4	62.9	43.3	73.2	54.5	44.1	71.8	54.6	**57.3**
Qwen2.5 32B	All-at-once	32.7	96.0	48.8	20.8	93.0	34.0	24.1	91.4	38,2	40.3
	Stepwise - Prev.	39.1	81.2	52.8	32.5	59.6	42.1	30.6	58.3	40.1	45.0
	Stepwise - Full	36.4	91.9	52.1	30.0	84.0	44.2	31.8	77.3	45.0	47.1
	PARC	36.4	78.0	49.6	26.7	63.4	37.5	26.5	92.0	41.1	42.8
	Graph-V	42.7	90.6	58.1	35.8	78.9	49.3	33.5	76.5	46.4	**51.3**
Qwen2.5 72B	All-at-once	41.8	88.8	56.9	31.7	85.4	46.2	37.1	76.1	49.9	51
	Stepwise - Prev.	45.5	86.5	59.6	32.5	88.3	47.5	35.3	77.9	48.6	51.9
	Stepwise - Full	42.7	94.2	58.8	32.5	88.3	47.5	35.9	87.7	50.9	52.4
	PARC	41.8	84.3	55.9	37.5	73.2	49.6	32.4	64.4	43.1	49.5
	Graph-V	47.3	88.3	61.6	45.0	81.2	57.9	34.7	86.5	49.5	**56.3**

Table 1: Performance of approaches, reporting Error, Correct Accuracy, and F1 Score. The baselines are: *All-at-once* (jointly verifying all steps), *Stepwise-Full* (verifying each step independently with full context), *Stepwise-Prev.* (using only preceding steps), and *PARC* (Mukherjee et al., 2025). Evaluations are on long-reasoning partitions from MATH (Hendrycks et al., 2021), OlympiadBench (He et al., 2024a), and OmniMATH (Gao et al., 2025).

of independent verification. The slightly lower performance of Stepwise-Prev. compared to Stepwise-Full hints at the benefit of including succeeding steps as context. Still, both suffer from attentional bias due to overly broad contexts. **Graph-V** overcomes both issues, achieving the highest average F1 scores (57.3, 51.3, 56.3), about 4–5 points above the next best method. Interestingly, contrary to our expectations, the PARC baselines performed poorly, suggesting that its approach underutilizes the logical structure to construct effective contexts. Most notably, **Graph-V** delivers substantial gains in Error accuracy across almost all configurations. This demonstrates its superior ability to overcome attentional bias and precisely identify faulty steps by focusing only on the most relevant logical dependencies. On the other hand, the results for easy subset in Table 2 show a diminished advantage for **Graph-V**, with the top-performing methods yielding comparable F1 scores. We attribute this to the simpler problems, which lack the complexity and long-range dependencies where **Graph-V** excels.

(RQ2): What is the most effective strategy to construct logic graph from a reasoning chain? Operating under the hypothesis that an accurate logic graph is crucial for effective verification, we assess three different graph construction strategies. To isolate the impact of graph quality, we evaluate performance using only graph-dependent

Config	Llama3.3	Qwen2.5
All-at-once	41.0	53.7
Stepwise - Prev.	65.2	62.7
Stepwise - Full	69.1	64.0
Graph-V	69.1	64.5

Table 2: Performance of **Graph-V** against three baselines. Evaluations report F1 scores on the merged ablation subset of easy samples from all partitions in ProcessBench across two backbones.

context methods: direct premises and logic clusters. As shown in Table 3, the *Coarse pass only* strategy, which considers only local dependencies, is the most efficient with just 4.8 LLM calls but yields the lowest accuracy (e.g., 43.2 F1 with logic cluster). In contrast, the *Fine pass only* strategy, which exhaustively searches all preceding blocks, achieves the best accuracy (46.8 F1) but nearly doubles the cost (9.2 calls). Our proposed *Coarse-to-fine* strategy offers the best trade-off, reaching accuracy close to the fine pass (46.4 F1) while requiring only a fraction more LLM calls (6.5) than the coarse pass. This highlights that our approach effectively captures the most critical logical dependencies without the prohibitive cost of a full global search, making it a practical strategy. Additional experiments to assess quality of the generated logic graphs are provided in Appendix B.

Strategy	Premises	Logic cluster	LLM Calls
Coarse pass only	42.0	43.2	4.8
Fine pass only	44.9	46.8	9.2
Coarse-to-fine	44.6	46.4	6.5

Table 3: End-to-end performance (F1 score) and efficiency (average LLM calls) of three graph construction strategies. Performance is evaluated using different context types (direct *Premises* and a *Logic cluster*, both without succeeding context.). All experiments use Llama3.3-70B as the verifier.

Config	Llama3.3-70b	Qwen2.5-32b
Preceding Context Ablation		
All preceding steps	42.87	45.00
Premises (PARC)	44.60	42.73
Logic cluster	46.43	45.73
Succeeding Context Ablation		
None	46.43	45.73
All dependents	53.37	46.93
Succeeding steps (k=3)	55.57	49.90
Succeeding steps (k=5)	57.33	51.27
Succeeding steps (k=7)	52.87	48.70
Succeeding steps (all)	55.20	50.77

Table 4: Performance (F1 score) from the ablation study on preceding and succeeding context, using Llama3.3-70b and Qwen2.5-32b as verifiers.

(RQ3): What are the most effective strategies for selecting preceding and succeeding context? Using the same logic graph construction algorithm, we performed a two-stage ablation study to identify the optimal context (Table 4). First, for the preceding context, the *logic cluster* proved most effective, achieving the highest scores (Llama3-70B: 46.43 F1, Qwen2.5-32b: 45.73 F1). This approach strikes a better balance than using all preceding steps, which introduces noise due to irrelevant steps, or using only direct premises, which is insufficient as it can miss crucial, non-adjacent steps. Next, using the logic cluster as the preceding context, we found that a forward-looking context is critical. After careful ablation, adding the *5 immediate subsequent steps* yielded the most dramatic performance boost, increasing the Llama3-70B score from 46.43 to 57.33 F1 and the Qwen2.5-32b score from 45.73 to 51.27 F1. This fixed-window lookahead was more effective than using all of a step's children in the graph or all subsequent steps, suggesting that a localized view of a step's immediate purpose is more valuable than a long-range and noisy view of all its future uses.

(RQ4): Interpretability and Error analysis of `Graph-V`. Representative outputs of `Graph-V` on case studies (Table 6, 7 in Appendix E) illustrate its interpretability and its ability to capture reasoning dependencies, while also providing detailed justification for correct and incorrect steps. To assess reliability, we further qualitatively analyzed 100 incorrect verifications made by `Graph-V`, categorizing failures into three main groups. The most prevalent category, **Subjective Assessment Errors (44%)**, arises when the verifier incorrectly marks a correct step as wrong after applying its subjective judgement. This includes cases where correct steps were evaluated as "unclear" or "oversimplified", "incomplete", or as having "no final contribution", which then leads to the biased verdict. The second category, **Logical and Conceptual Errors (35%)**, comprises failures in the verifier's core reasoning, including invalid deductions or flawed logic, misunderstanding of mathematical concepts, and incorrect calculations. The third category, **Scope and Procedural Errors (15%)**, covers cases where the model misidentifies which step should be verified because step boundaries are unclear, or it departs from its task by jumping ahead to the final answer. Finally, about 6% of errors fall under **Other Errors**, including issues in logic graph construction (e.g., missing premises), mislabeling, or format mismatches between gold labels and verifier outputs (e.g., 0.2 vs. 1/5). These results suggest that while our approach narrows the context, future work should focus on constraining the LLM's subjective evaluation and procedural deviation.

6 Conclusion

Reliably verifying complex LLM reasoning is a critical bottleneck for trustworthy AI. This paper introduces `Graph-V`, a framework that reframes stepwise verification as a structured, dependency-aware task. By evaluating each step within a minimal context, `Graph-V` mitigates sequential and attentional biases. On the ProcessBench benchmark, `Graph-V` achieves competitive performance with significant F1 improvements across model families. The results validate our hypothesis: a pruned, bidirectional context, combining a step's "logic cluster" with a forward-looking window, is crucial for accurate, efficient assessment. Error analysis shows failures often arise from verifier subjectivity, suggesting future work on constraining LLM behavior. Ultimately, `Graph-V` provides a principled path towards transparent, interpretable, reliable reasoning.

Limitations

Despite significant progress, LLMs remain prone to errors and biases, logical inconsistencies, conceptual mistakes, and as reflected in our findings on subjective assessment. These limitations highlight the need for cautious deployment and open important avenues for future research in reasoning-intensive applications. In this study, we concentrated on math-related reasoning tasks. Since the verification process relied solely on the LLM itself, the model was vulnerable to both basic computational errors (e.g., addition or multiplication mistakes) and failures in more complex calculations. Furthermore, real-world reasoning tasks are more diverse and sophisticated, they often require implicit reasoning, where the model must integrate additional background knowledge beyond what is explicitly provided. Addressing this challenge will demand improvements in pipeline components and reasoning mechanisms to better handle implicit, knowledge-intensive tasks.

References

Janice Ahn, Rishu Verma, Renze Lou, Di Liu, Rui Zhang, and Wenpeng Yin. 2024. Large language models for mathematical reasoning: Progresses and challenges. In *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024: Student Research Workshop, St. Julian's, Malta, March 21-22, 2024*, pages 225–237. Association for Computational Linguistics.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pages 17682–17690. AAAI Press.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *CoRR*, abs/2110.14168.

Bofei Gao, Feifan Song, Zhe Yang, Zefan Cai, Yibo Miao, Qingxiu Dong, Lei Li, Chenghao Ma, Liang Chen, Runxin Xu, Zhengyang Tang, Benyou Wang, Daoguang Zan, Shanghaoran Quan, Ge Zhang, Lei Sha, Yichang Zhang, Xuancheng Ren, Tianyu Liu, and Baobao Chang. 2025. Omni-math: A universal olympiad level mathematic benchmark for large language models. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net.

Olga Golovneva, Moya Chen, Spencer Poff, Martin Corredor, Luke Zettlemoyer, Maryam Fazel-Zarandi, and Asli Celikyilmaz. 2023. ROSCOE: A suite of metrics for scoring step-by-step reasoning. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Douglas P. Gregor and Andrew Lumsdaine. 2005. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 423–437. ACM.

Chaoqun He, Renjie Luo, Yuzhuo Bai, Shengding Hu, Zhen Leng Thai, Junhao Shen, Jinyi Hu, Xu Han, Yujie Huang, Yuxiang Zhang, Jie Liu, Lei Qi, Zhiyuan Liu, and Maosong Sun. 2024a. Olympiadbench: A challenging benchmark for promoting AGI with olympiad-level bilingual multimodal scientific problems. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pages 3828–3850. Association for Computational Linguistics.

Hangfeng He, Hongming Zhang, and Dan Roth. 2024b. Socreval: Large language models with the socratic method for reference-free reasoning evaluation. In *Findings of the Association for Computational Linguistics: NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pages 2736–2764. Association for Computational Linguistics.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the MATH dataset. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Fengqing Jiang, Zhangchen Xu, Yuetai Li, Luyao Niu, Zhen Xiang, Bo Li, Bill Yuchen Lin, and Radha Poovendran. 2025. Safechain: Safety of language models with long chain-of-thought reasoning capabilities. In *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 23303–23320. Association for Computational Linguistics.

Pei Ke, Haozhe Ji, Yu Ran, Xin Cui, Liwei Wang, Linfeng Song, Xiaoyan Zhu, and Minlie Huang. 2021. Jointgt: Graph-text joint representation learning for text generation from knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6,*

2021, volume ACL/IJCNLP 2021 of *Findings of ACL*, pages 2526–2538. Association for Computational Linguistics.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. *ArXiv*, abs/2305.20050.

Zhan Ling, Yunhao Fang, Xuanlin Li, Zhiao Huang, Mingu Lee, Roland Memisevic, and Hao Su. 2023. Deductive verification of chain-of-thought reasoning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Meta. 2024. Build the future of ai with meta llama 3, 2024.

Sagnik Mukherjee, Abhinav Chinta, Takyoung Kim, Tarun Anoop Sharma, and Dilek Hakkani-Tür. 2025. Premise-augmented reasoning chains improve error identification in math reasoning with llms. *CoRR*, abs/2502.02362.

OpenAI. 2025. Introducing gpt-4.1 in the api. `https://openai.com/index/gpt-4-1/`. Accessed: 2025-09-12.

Archiki Prasad, Swarnadeep Saha, Xiang Zhou, and Mohit Bansal. 2023. Receval: Evaluating reasoning chains via correctness and informativeness. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 10066–10086. Association for Computational Linguistics.

Qwen. 2024. Qwen2.5: A party of foundation models.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The graph neural network model. *IEEE Transactions on Neural Networks*, 20:61–80.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *CoRR*, abs/2402.03300.

Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. 2023. A survey of using large language models for generating infrastructure as code. In *Proceedings of the 20th International Conference on Natural Language Processing (ICON)*, pages 523–533, Goa University, Goa, India. NLP Association of India (NLPAI).

Junhan Yang, Zheng Liu, Shitao Xiao, Chaozhuo Li, Defu Lian, Sanjay Agrawal, Amit Singh, Guangzhong Sun, and Xing Xie. 2021. Graphformers: Gnn-nested transformers for representation learning on textual graph. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 28798–28810.

Renrui Zhang, Chengzhuo Tong, Zhizheng Zhao, Ziyu Guo, Haoquan Zhang, Manyuan Zhang, Jiaming Liu, Peng Gao, and Hongsheng Li. 2025. Let's verify and reinforce image generation step by step. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pages 28662–28672. Computer Vision Foundation / IEEE.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2025. Processbench: Identifying process errors in mathematical reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, pages 1009–1024. Association for Computational Linguistics.

A Benchmark Datasets

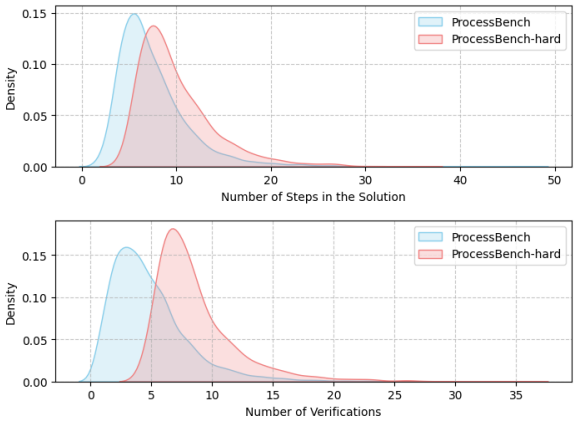

Figure 3: Kernel Density Estimate plot for the number of steps and the number of required verification in each sample

To evaluate performance across varying levels of verification complexity, we construct two evaluation subsets from the ProcessBench dataset: a *hard* subset and an *easy* subset, each consisting of 1,000 examples. We define verification complexity based on the number of verification steps required to identify the first incorrect step or confirm full correctness. Specifically, for each sample, the number of verification steps is computed as `label` $+ 1$ if `label` ≥ 0, and `len(steps)` otherwise; this reflects the actual number of steps a

verifier must inspect to either find an error or verify the solution as entirely correct. This metric mirrors human problem-solving, where longer reasoning chains typically correspond to harder problems, and excessive verification increases contextual demands that can strain LLMs. To construct the hard subset, we select the top 333 samples from each of the three more challenging datasets (MATH, OlympiadBench, and Omni-MATH), ranked by verification steps, resulting in 1,000 high-complexity problems. In contrast, the easy subset is built by randomly sampling 250 examples from each of the four original datasets (GSM8K, MATH, OlympiadBench, and Omni-MATH), yielding 1,000 lower-complexity examples.

Figure 3 shows kernel density estimates comparing the full dataset to the hard subset along two dimensions: number of steps and number of verifications. Both plots indicate a clear rightward shift in the hard subset, with longer solution chains and deeper error positions. In particular, the verification distribution in the hard subset exhibits a heavier tail—extending beyond 35 steps—suggesting that these examples require significantly more reasoning effort before reaching a conclusion.

B Additional Experimental Results

In this section, we present supplementary experimental results evaluating the performance of different approaches in constructing the logic graph during the *Logic Graph Construction Phase*. To establish a reference standard, we employ GPT-4.1 (OpenAI, 2025) as the oracle model for generating gold-standard logic graphs. The graphs produced by various methods are then compared against this oracle output. Specifically, we examine three approaches in the construction process, leveraging Qwen2.5 72B (Qwen, 2024) as the baseline: **1) Coarse pass only**: Apply only the coarse pass for graph construction. **2) Fine pass only**: Apply only the fine pass for graph construction. **3) Coarse-to-fine**: Sequentially combine the coarse and fine passes in an integrated manner. We evaluate the quality of the generated graphs using two metrics: *(i) Recall*: Measure the proportion of oracle edges correctly recovered, and *(ii) Token Count*: Measure the total number of tokens consumed during the construction process.

Table 5 reports the experimental results. The findings suggest that the **Coarse-to-fine** approach provides a balanced trade-off: it effectively captures both local and long-range logical dependencies, while simultaneously reducing the token consumption associated with LLM calls.

Strategy	Recall	Token Counts
Coarse pass only	78.2	1.3K
Fine pass only	82.5	4.4K
Coarse-to-fine	86.7	2.5K

Table 5: The graph quality (Recall and Token Counts) of different graph construction strategies (Qwen 2.5 72B as the baseline) compared to the gold-standard graph labels provided by GPT-4.1

C The Graph-Based Nature of Logical Reasoning Chains

While direct premises are necessary, they are not always sufficient for robust verification. The insufficiency arises because the semantic context required to interpret a premise may itself be established in steps that are ancestors of that premise. Consider a linear reasoning chain, $s_j \rightarrow \cdots \rightarrow s_k \rightarrow s_i$. The direct premise for s_i is s_k. However, if s_j introduces a critical definition or assumption (e.g., assigning a value to a variable) that is propagated through to s_k, then a verifier assessing s_i cannot succeed without access to the context of s_j. A premises-only context for s_i would only contain $\{s_k\}$, thus omitting the crucial information from s_j. The logic cluster, by definition, traces the full ancestral path back to the source node (s_j), ensuring this entire self-contained logical thread is included in the context. This guarantees that the verifier has access to the complete sub-argument, from its foundational assumptions to its conclusion, which is essential for accurate assessment. To better access the logical connection internally present in the reasoning chain, we further provide statistics derived from the gold-standard labels generated by GPT-4.1 (same setup in Appendix B). For samples with reasoning chains of length ≥ 10 steps, each node has on average 2.24 directly connected nodes and 5.53 nodes reachable within two hops (through its immediate children). Additionally, for long-distance connection (an edge between two nodes whose step indices differ by at least 5), each sample solution contains an average of 1.49 long-distance connections. These highlight the necessity of modeling reasoning chains as graphs, as such a representation effectively captures both short-range and long-range dependencies within the logical flow.

D Algorithm Details

The detailed implementation of the algorithm for Logic Graph Construction and Logic Cluster Algorithm are provided as the pseudo-code in Algorithm 1 and 2.

E Case Study

We also present the case study results of DISV on the ProcessBench dataset in Tables 6 and 7. The table includes a problem $\mathcal{P}$, solution steps $\mathcal{S}$, constructed logic graph $\mathcal{G}$, predicted lable p, and correct label g. Table 6 showcases correctly predicted examples, demonstrating Graph-V's ability to accurately capture underlying logic structures and generate well-grounded step. Conversely, Table 7 highlights an incorrectly predicted case of two error types as detailed in Section 5.2. The example illustrate Reasoning Errors. These insights serve as a foundation for future improvements.

F Prompt Templates

For better reproducibility, we include all prompt templates in the appendix. A quick reference list summarizing the templates and their respective usages is provided below:

- Figure 4: Prompt the LLM to find all local premises of every steps within a block of steps.
- Figure 5: Prompt the LLM to find premises for the targeted step given a block of steps.
- Figure 6: Prompt the LLM to verify a targeted step given a list of steps.
- Figure 7: Prompt the LLM to verify all the steps in the solution.

Algorithm 1: Logic Graph Construction Algorithm

```
Input  : Sequence of steps S = [s_1, s_2, ..., s_n];
         Coarse pass window size w_local;
         Coarse pass stride δ;
         Fine pass window size w_long
Output : Graph G representing dependency between reasoning steps
Initialize:
  E ← ∅ ;                                   // Initialize global set of dependency edges
  n ← |S| ;                                 // Total number of solution steps
Function CoarsePass(S, i, w_local):
    W ← S[i : i + w_local] ;                // Local context window
    (E_local, N) ← LLM(COARSE_PASS_CONSTRUCTION, W) ;   // Local premises and node status
    return (E_local, N)
Function FinePass(S, j, w_long, found_premises):
    E ← found_premises
    for i ← 0; i < j; i ← i + w_long do
        W ← S[i : i + w_long] ;             // Long-range context window
        (E', stop) ← LLM(FINE_PASS_CONSTRUCTION, W, s_j, E) ;   // Long-range premises
        E ← E ∪ E';
        if stop then
            break ;                         // Check if all premises found
    return E
Function Main(S, w_local, δ, w_long):
    E ← ∅;
    for i ← 0; i < n; i ← i + δ do
        (E_local, N) ← CoarsePass(S, i, w_local);
        E ← E ∪ E_local;
        foreach step_j ∈ N do
            if step_j.lack_premise then
                E_long ← FinePass(S, j, w_long, step_j.local_premises);
                E ← E ∪ E_long;
    G ← ConstructGraph(E) ;                 // Construct graph G from all collected premises
    return G
```

Algorithm 2: Logic Cluster Construction Algorithm

```
Input  : Graph G = (S, E), target step i
Output : Set of steps in logic cluster L_i
Function IdentifyRootNodes(G):
    R ← ∅ ;                                 // Initialize an empty set for root nodes
    foreach node i in G do
        Succ(i) ← immediate successors of i ;   // Find all immediate successors of node i
        if Succ(i) = ∅ or any j ∈ Succ(i) such that j ≠ i + 1 then
            R ← R ∪ {i} ;                   // Mark node i as a root node
    return sorted list of R;
Function ConstructLogicCluster(G, i, R):
    R ← IdentifyRootNodes(G);
    L_i ← {i};
    Stack ← [i] ;                           // Start cluster with the target node i
    while Stack is not empty do
        c ← Stack.pop() ;                   // Pop a node to explore its ancestors
        foreach predecessor p of c in G do
            if p ∉ L_i then
                L_i ← L_i ∪ {p} ;           // Include unvisited predecessor in the cluster
                if p ∉ R then
                    Stack.push(p) ;         // Explore further back if p is not a root
    return L_i;
```

Question Q	Solution S	Logic graph $\mathcal{G}$	Predicted p; Gold g
Let T be a rational number. Let N be the smallest positive T-digit number that is divisible by 33 . Compute the product of the last two digits of N.	**S0:** We start by noting that a number divisible by 33 must be divisible by both 3 and 11. Let's denote this number by N. The smallest T-digit number is 10^{T-1}. We need to find the smallest number greater than or equal to 10^{T-1} that is divisible by 33. First, we compute the remainder when 10^{T-1} is divided by 33. Let $r = 10^{T-1} \mod 33$. Then, the smallest number greater than or equal to 10^{T-1} that is divisible by 33 is $10^{T-1} + (33 - r)$. This number is N. **S1:** To find the last two digits of N, we need to compute $N \mod 100$. Since $N = 10^{T-1} + (33 - r)$, we have: $N \equiv 10^{T-1} + (33 - r) \pmod{100}$. **S2:** We need to consider two cases: $T = 2$ and $T \geq 3$. **S3:** Case 1: $T = 2$ **S4:** The smallest 2-digit number divisible by 33 is $N = 33$. The last two digits are 33, so their product is $3 \times 3 = 9$. **S5:** Case 2: $T \geq 3$ **S6:** For $T \geq 3$, 10^{T-1} is a number that ends in at least two zeros. Therefore, $10^{T-1} \mod 100 = 0$. This simplifies our expression for $N \mod 100$ to: $N \equiv 33 - r \pmod{100}$. **S7:** Since $r = 10^{T-1} \mod 33$ and 10^{T-1} is a multiple of 11 for $T \geq 3$, r is a multiple of 11. The possible values for r are 0, 11, 22. We need to check each case: **S8:** If $r = 0$, then $N \equiv 33 - 0 \equiv 33 \pmod{100}$. If $r = 11$, then $N \equiv 33 - 11 \equiv 22 \pmod{100}$. If $r = 22$, then $N \equiv 33 - 22 \equiv 11 \pmod{100}$. **S9:** In all cases, the last two digits of N are 33, 22, or 11. However, since N is the smallest T-digit number divisible by 33, it cannot end in 11 or 22 (because 11 and 22 are not 3-digit numbers or greater). Therefore, the last two digits of N must be 33. The product of the last two digits of N is $3 \times 3 = 9$. **S10:** Thus, the answer is: $\boxed{9}$.	$\langle 0, 1\rangle$ $\langle 0, 4\rangle$ $\langle 0, 7\rangle$ $\langle 1, 6\rangle$ $\langle 2, 3\rangle$ $\langle 2, 5\rangle$ $\langle 2, 6\rangle$ $\langle 3, 4\rangle$ $\langle 4, 9\rangle$ $\langle 5, 6\rangle$ $\langle 5, 7\rangle$ $\langle 6, 7\rangle$ $\langle 7, 8\rangle$ $\langle 8, 9\rangle$ $\langle 9, 10\rangle$	**Considering the step 7.** "Since $r = 10^{T-1} \mod 33$ and 10^{T-1} is a multiple of 11 for $T \geq 3$, r is a multiple of 11. The possible values for r are 0, 11, 22.". First, we need to verify the claim that 10^{T-1} is a multiple of 11 for $T \geq 3$. This claim is demonstrably false. For example, if $T = 3$, $10^{T-1} = 10^2 = 100$, and $100 \pmod{11} = 1$, which is not a multiple of 11. Likewise, for $T = 4$, $10^{T-1} = 10^3 = 1000$, and $1000 \pmod{11} = 10$. Since 10^{T-1} is not necessarily a multiple of 11, the deduction that $r = 10^{T-1} \pmod{33}$ must be a multiple of 11 (i.e., 0, 11, or 22) is incorrect. The remainder r can, in fact, take any value from 0 to 32. Thus, step 7 contains a significant error in its reasoning. **Verdict:** **False** **Label:** **False**

Table 6: Case study of Graph-V on the ProcessBench dataset with a correctly predicted sample

Question Q	Solution S	Logic graph $\mathcal{G}$	Predicted p; Gold g
Let $f(x) = 2^{kx} + 9$, where k is a real number. If $f(3) : f(6) = 1 : 3$, determine the value of $f(9) - f(3)$.	**S0:** Given that $f(3) : f(6) = 1 : 3$, we have: $f(3) = \frac{1}{3} f(6)$ **S1:** Let's substitute the given function definition into these expressions: $f(3) = 2^{k \cdot 3} + 9$, $f(6) = 2^{k \cdot 6} + 9$ **S2:** So, $2^{3k} + 9 = \frac{1}{3}(2^{6k} + 9)$ **S3:** Expanding this equation gives: $3(2^{3k} + 9) = 2^{6k} + 9$, $3 \cdot 2^{3k} + 27 = 2^{6k} + 9$, $3 \cdot 2^{3k} = 2^{6k} - 18$ **S4:** This equation, relating 2^{3k} and 2^{6k}, does not directly provide a clear path to $f(9) - f(3)$ without first determining k or a specific structure for the exponents. **S5:** Given the direct goal of finding $f(9) - f(3)$, let's simplify the function directly with respect to x: $f(x) = 2^{kx} + 9$ **S6:** For $f(9)$: $f(9) = 2^{9k} + 9$ **S7:** And for $f(3)$: $f(3) = 2^{3k} + 9$ **S8:** Therefore, $f(9) - f(3) = (2^{9k} + 9) - (2^{3k} + 9) = 2^{9k} - 2^{3k}$ **S9:** Without specifically solving for k from the ratio condition, the correct approach to answering the question as initially framed involves recognizing the structure of the function $f(x) = 2^{kx} + 9$ directly for the calculation of $f(9) - f(3)$, which was: $f(9) - f(3) = 2^{9k} - 2^{3k}$ **S10:** Without solving for k specifically from the given ratio condition, this is the most straightforward response based on the function's definition and the goal of calculating $f(9) - f(3)$. The specific numerical value of $f(9) - f(3)$ cannot be determined without the value of k, given the current information.	$\langle 0, 2\rangle$ $\langle 1, 2\rangle$ $\langle 2, 3\rangle$ $\langle 3, 4\rangle$ $\langle 4, 5\rangle$ $\langle 5, 6\rangle$ $\langle 5, 7\rangle$ $\langle 6, 8\rangle$ $\langle 7, 8\rangle$ $\langle 8, 9\rangle$ $\langle 9, 10\rangle$	**Considering the step 9.** Step 9 correctly derived $f(9) - f(3) = 2^{9k} - 2^{3k}$, but its incompleteness lies in not determining the numerical value. Using the given ratio $f(3) : f(6) = 1 : 3$, we establish $\frac{2^{3k}+9}{2^{6k}+9} = \frac{1}{3}$. Let $y = 2^{3k}$. This leads to the quadratic equation $y^2 - 3y - 18 = 0$, whose positive solution is $y = 6$. Thus, $2^{3k} = 6$. Substituting this back, $f(3) = 2^{3k} + 9 = 6 + 9 = 15$, and $f(9) = 2^{9k} + 9 = (2^{3k})^3 + 9 = 6^3 + 9 = 216 + 9 = 225$. Therefore, $f(9) - f(3) = 225 - 15 = 210$, demonstrating that the problem required solving for 2^{3k} to achieve the final numerical answer, which step 9 did not do. **Verdict:** **False** **Label:** **True**

Table 7: Case study of Graph-V on the ProcessBench dataset with an incorrectly predicted sample

COARSE_PASS_GRAPH_CONSTRUCTION

Overview:
You are an expert in mathematical logic and reasoning. You will be given a math problem and a partial solution consisting of a set of indexed steps. Your task is to analyze each step in the provided set and: (1) Identify only the direct premises it depends on; and (2) Determine whether the step is resolved.

Definitions:
- Direct Premise: A step's direct premise is an earlier step whose concluding expression or result serves as the immediate object of transformation in the target step. The target step acts directly upon the output of its direct premise.
- Resolved: A step is resolved if all its direct premises are included in the provided solution set. If any direct premise is missing, the step is unresolved.

Instruction: How to Analyze Each Step
Your analysis must strictly follow this process. Do not re-derive or re-prove the target step.
1. Identify the Core Action: Determine the target step's primary mathematical operation (e.g., substitution, simplification, applying a rule, algebraic manipulation).
2. Isolate the Active Input: Pinpoint the specific expression, equation, or value that the action is performed on. This is the material being directly transformed.
3. Trace the Input to its Source: Locate the preceding step where this active input was the final conclusion. This source step is the direct premise.

Guiding Principle:
A direct premise provides the triggering result for the target step's action. Distinguish this from "background context." For example, if a step substitutes $y = 3$, its direct premise is the step that concluded $y = 3$, not an earlier step that introduced the variable y.

Task:
1. For each step, briefly explain your reasoning by stating the core action and tracing its active input(s) to their source step(s).
2. Based on your analysis, generate a JSON object that strictly adheres to the following format:

{"<index>": {"premises": [<list of premise indices>], "resolved": <boolean>}, ...}

[Math Problem]
{problem}

[Partial Solution]
{tagged_steps}

Figure 4: Coarse pass graph construction prompt

FINE_PASS_GRAPH_CONSTRUCTION

Overview:
You are an expert in mathematical logic and reasoning. You are given a math problem, a partial solution consisting of a set of indexed steps, a target step, and its previously known direct premises (optional). Your goal is to (1) identify only the direct premises from the partial solution that the target step depends on; and (2) determine if the target step is resolved.

Definitions:
- Direct Premise: A step's 'Direct Premise' is an earlier step whose concluding expression or result is the immediate object of transformation in the target step. The target step acts directly upon the output of its direct premise.
- Resolved: The target step is "resolved" if all of its direct premises are included in the union of the provided solution set and its known premises. If any direct premise is missing from this combined set, the step is "unresolved".

Instruction: How to Analyze the Target Step
Your analysis must strictly follow this process. Do NOT re-derive or re-prove the target step.

1. Identify the Core Action: Determine the target step's primary mathematical action (e.g., substitution, simplification, applying a rule, algebraic manipulation).
2. Isolate the Active Input: Pinpoint the specific expression, equation, or value that the action is performed on. This is the material being directly transformed.
3. Trace the Input to its Source: Find the preceding step where this active input was the final conclusion. This source step is the direct premise.

Guiding Principle: A direct premise provides the triggering result for the target step's action. You must distinguish this from "background context." For example, if a step substitutes y=3, its direct premise is the step that concluded y=3, not the much earlier step that originally defined the variable y.

Task:
1. Briefly explain your reasoning by stating the core action and tracing its active input(s) to their source step(s).
2. Based on your analysis, generate a JSON object that strictly adheres to the following format:
{{ "premises": [<list of integer indices for direct premises>], "resolved": <boolean>}}

[Math Problem]
{problem}

[Partial Solution]
{tagged_steps}

[Known Premises]
{tracked_premises}

[Target Step]
{target_step}

Figure 5: Fine pass graph construction prompt

STEPWISE_VERIFICATION

The following is a math problem and a (partial) solution (split into steps, enclosed with tags, and indexed from 0). The solution may be partial or complete:

[Math Problem]
{problem}

[Solution]
{tagged_steps}

Use the problem statement and given steps as context, your task is to review and critique ONLY step {idx}. You must reason and explain clearly BEFORE go to your final answer. Do NOT directly skip to the conclusion or provide your final answer before reasoning. If an error is found in step {idx}, return 0. Otherwise, return 1.

Please put your final answer (i.e., 1 or 0) in \boxed{{}}.

Figure 6: Stepwise verification prompt

ALL_AT_ONCE_VERIFICATION

The following is a math problem and a solution (split into steps, enclosed with tags and indexed from 0):

[Math Problem]
{problem}

[Solution]
{tagged_steps}

Your task is to review and critique the solution step by step. For each step, you must reason and explain clearly BEFORE go to your final answer. Do NOT directly skip to the conclusion or provide your answer before reasoning. Once you identify an error in a step, return the index of the step where the earliest error occurs. Otherwise, return the index of -1 (which typically denotes "not found").

Please put your final answer (i.e., the index) in \boxed{{}}.

Figure 7: All-at-once verification prompt