GraphStateEval: A Step-by-Step Evaluation Framework for Graph Algorithm Execution in Large Language Models via Intermediate State Tracing

Anonymous Authors[1]

Abstract

We introduce **GraphStateEval**, a step-by-step evaluation framework and benchmark for measuring how accurately large language models (LLMs) execute graph algorithms, requiring models to reproduce complete internal memory states—queue, stack, priority queue, visited set, and neighbour lists—at every loop iteration under zero-shot execution with no teacher forcing or intermediate state corrections. Ground-truth traces are generated deterministically by a programmatic engine with canonical tie-breaking rules, enabling arbitrarily complex, automatically verified benchmarks that resist memorisation and saturation. We evaluate Gemma 3 12B on BFS, DFS, and Dijkstra across five graph topologies scaled from $N{=}3$ to $N{=}12$ nodes using two zero-shot prompting strategies (natural language and source code). Our Step-Exact-Match (SEM) metric reveals a pronounced topology hierarchy: path graphs sustain near-perfect tracing throughout, while sparse-random graphs degrade substantially beyond $N{\approx}6$; conversely, dense-random and complete graphs often maintain surprising resilience up to $N{\approx}10$ due to shallower traversals. All results reflect single-seed evaluation and should be interpreted as exploratory. A persistent gap between SEM and Final-Answer-Correctness (FAC) across BFS, DFS, and Dijkstra illustrates algorithmic reward hacking—where correct final answers are produced despite internally incorrect intermediate state traces—showing that answer-level evaluation can drastically overestimate true step-level algorithmic execution fidelity.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

1. Introduction

Graph algorithms are canonical iterative procedures whose correctness depends on the faithful maintenance of explicit data structures across every loop iteration. BFS maintains a FIFO queue, DFS a LIFO stack, and Dijkstra a min-heap priority queue together with a distance dictionary. Executing any of these correctly requires a model to initialise those structures precisely, update them at every step, and terminate at the right moment.

Existing work on LLM graph reasoning (Wang et al., 2023; Fatemi et al., 2024; Guo et al., 2023; Perozzi et al., 2024) asks models to produce final answers to graph-reachability or shortest-path questions and assesses correctness by comparing output to ground truth. This single-score evaluation has a notable blind spot: a model can produce the correct traversal order by pattern-matching on graph structure without ever maintaining a valid queue or stack. We refer to this phenomenon as *algorithmic reward hacking*, and demonstrate it empirically across BFS, DFS, and Dijkstra. Note that the SEM–FAC gap documents an observational discrepancy between surface and step-level correctness; it does not prove the internal generative mechanism, which may include step reordering or hallucinated steps that happen to yield a correct final answer.

Prior work on step-level evaluation has relied on *teacher forcing* (Taylor et al., 2024): the correct state S_{k-1} is supplied before predicting S_k, preventing error accumulation and making it impossible to observe cascading failures. Research on transformer capabilities has established that compositional reasoning degrades with dependent chain length (Dziri et al., 2023), that each additional step compounds error probability (Merrill & Sabharwal, 2024), and that self-attention lacks a dedicated register mechanism for holding many co-existing intermediate values (Gong & Zhang, 2024). These results motivate our framework's focus on fully autonomous multi-step execution.

The CLRS algorithmic reasoning benchmark (Veličković et al., 2022) evaluates algorithmic reasoning in graph neural networks; CLRS-Text (Markeeva et al., 2024) extends this to language models using natural-language state descrip-

tions under teacher forcing. GraphStateEval goes further in three respects: (i) it requires *complete* internal memory snapshots in structured JSON at every step; (ii) it evaluates *fully autonomous* zero-shot execution without teacher forcing; and (iii) its ground-truth generator is fully parametric, producing verified benchmarks of arbitrary difficulty without human annotation. This approach is motivated by the systematic complexity-dimension framework of Patil et al. (2026), which demonstrated that varying one difficulty axis at a time while holding others fixed reveals mechanistically distinct failure modes in LLM reasoning.

Contributions.

- **GraphStateEval Framework.** A step-level evaluation framework covering BFS, DFS, and Dijkstra across five graph topologies for $N \in \{3, \ldots, 12\}$.
- **Dynamic Auto-Verified Dataset Generator.** A fully parametric, deterministic ground-truth engine with canonical tie-breaking that produces arbitrarily complex verified traces without human annotation, supporting dynamic benchmarks that resist saturation by memorisation.
- **Dual Prompting Strategy.** A systematic comparison of natural-language and source-code zero-shot prompting, revealing algorithm-dependent trade-offs.
- **Reward-Hacking Diagnosis.** Empirical evidence of a persistent SEM–FAC gap across BFS, DFS, and even Dijkstra, illustrating that answer-level benchmarks can drastically overestimate step-level execution fidelity.

2. Related Work

LLM graph reasoning. Wang et al. (2023) introduced NLGraph showing that large models struggle with a range of graph reasoning tasks including reachability, shortest path, and connectivity. Fatemi et al. (2024) found performance is highly sensitive to graph serialisation format. Guo et al. (2023) conducted a broad empirical study of GPT-4 on graph-structured data, and Perozzi et al. (2024) studied graph structure encoding for LLMs. All evaluate only final-answer correctness, making it impossible to diagnose *how* a model reached its answer.

Algorithmic reasoning benchmarks. The CLRS benchmark (Veličković et al., 2022) evaluates graph neural networks on classical algorithms by exposing intermediate execution traces as supervision. CLRS-Text (Markeeva et al., 2024) adapts this to language models, representing algorithm states in natural language under teacher forcing; finding that LLMs struggle with algorithms requiring complex iterative state maintenance, broadly consistent with our findings. GraphStateEval complements CLRS-Text by requiring complete structured JSON snapshots under fully autonomous execution, introducing topology as an explicit complexity axis, and generating benchmarks programmatically.

Teacher forcing vs. autonomous execution. MAGMA (Taylor et al., 2024) evaluates LLMs on graph algorithm steps by providing the correct previous state as input. Teacher forcing prevents error accumulation: a model that consistently makes per-step errors can still score well because each prediction is reset to ground truth. GraphStateEval evaluates the model's ability to maintain consistent state across the entire execution without external correction, directly exposing cascading failures. Direct comparison to MAGMA is not performed as its teacher-forced evaluation protocol is fundamentally incompatible with our autonomous execution setup; MAGMA would need to be re-evaluated in zero-shot mode.

Transformer limitations on iterative computation. Dziri et al. (2023) showed transformer performance on compositional tasks degrades as the number of dependent steps grows. Merrill & Sabharwal (2024) proved that additional steps compound error probability in chain-of-thought generation. Gong & Zhang (2024) demonstrated formally that self-attention lacks a register mechanism for holding many independent intermediate values.

Dynamic benchmarks. Static benchmarks risk being folded into training data. GraphStateEval addresses this by making complexity a runtime parameter: generating harder problems requires changing a single integer, with new graph instances produced on demand without human annotation. Patil et al. (2026) applied a similar parametric philosophy to algebraic reasoning, systematically varying nine orthogonal complexity dimensions and finding that working memory is a scale-invariant architectural bottleneck; GraphStateEval instantiates a similar principle for graph algorithm execution.

3. The GraphStateEval Framework

3.1. Problem Setup and Topology Taxonomy

We evaluate on five graph topologies spanning a range of structural complexity. Graph scale N denotes the number of nodes, varied from 3 to 12.

Path. A strictly linear chain with branching factor 1 at every interior node. The queue or stack never exceeds $\mathcal{O}(1)$ live entries, making state management straightforward.

Complete. Every node connected to every other; branching factor $N-1$. At the first dequeue/pop, the entire neighbourhood is inserted, stressing the frontier data structure.

Tree. A uniformly random labeled tree generated via a

Prüfer sequence. It features irregular, unpredictable hierarchical branching.

Sparse Random (s.rnd). An Erdős–Rényi graph at low edge density ($p = 0.25$) with guaranteed connectivity. Structural irregularity prevents topological pattern-matching and requires the model to track exact insertion sequences.

Dense Random (d.rnd). Same family at high edge density ($p = 0.65$), combining high branching with structural irregularity.

The branching factor at each node is the primary driver of queue and stack memory demand. Path graphs impose a constant branching factor of 1; random graphs impose variable, unpredictable branching.

3.2. Deterministic Ground-Truth Generation

A programmatic reference engine executes BFS, DFS, and Dijkstra on every (graph, N, topology) triple with the following **canonical tie-breaking rules** that guarantee a single unambiguous reference trace:

- **BFS.** Unvisited neighbours are sorted in *ascending* node-ID order before enqueueing. Nodes are marked visited *at enqueue time*.
- **DFS.** Unvisited neighbours are sorted in *descending* order before pushing, so the smallest-ID unvisited neighbour is at the top. Nodes are marked visited *at pop time*; duplicate entries are discarded silently when popped (only successful, unvisited pops are recorded as steps).
- **Dijkstra.** The priority queue is ordered by ascending (distance, node-ID). The nodes are marked as visited at extraction time. Already-visited nodes encountered at extraction are discarded silently (only successful, unvisited extractions are recorded as steps). Relaxation pushes a new entry without removing stale ones.

Edge weights for Dijkstra are assigned by the deterministic function $w(u, v) = ((u + v) \bmod 9) + 1$, which maps any pair of connected node IDs to a reproducible integer in $[1, 9]$ with no additional randomness. The source node is fixed to node 1 across all algorithms, topologies, and scales. This generator is fully parametric: to produce graphs of a new size or topology, one changes a single argument, making the benchmark resistant to saturation by memorisation.

3.3. Internal State Snapshot Variables

All three algorithms share **universal fields** recorded at every step i: $\{$ `step`, `node_extracted`, `visited` $\}$ where `visited` is the sorted list of finalised nodes. Each algorithm additionally requires algorithm-specific fields:

BFS: `queue` (exact FIFO array, left = front); `neighbors_enqueued` (ascending, only new enqueues at step i).

DFS: `stack` (exact LIFO array, right = top); `neighbors_pushed` (descending, only pushes at step i).

Dijkstra: `priority_queue` (full 2-D $[d, v]$ array sorted by (d, v), with duplicate entries retained); `distances` (complete map $v \mapsto d_v$ or `"inf"` covering every $v \in V$); `neighbors_relaxed` (ascending, only those with strict improvement `alt` $< d_v$).

3.4. Two-Pronged Zero-Shot Prompting

We deploy two independent zero-shot prompting strategies for each algorithm. Both are zero-shot—no worked example is provided—and the model generates the entire trace in a single forward pass.

Natural Language (NL). The algorithm is described in structured plain English, specifying initialisation, the per-step loop body including tie-breaking rules, and the termination condition.

Source Code (Code). A complete executable Python implementation is provided using standard library idioms (`collections.deque` for BFS, a plain list with `.pop()` for DFS, `heapq` for Dijkstra). The model is instructed to simulate execution of this exact code step by step.

The Code prompts additionally provide a pre-sorted adjacency list alongside the edge list, since the Python implementation operates directly over adjacency list representations. This difference reflects the natural input format each modality demands: providing the adjacency list directly to the Code prompt isolates core algorithmic execution fidelity by preventing it from being conflated with standard data-parsing overhead, while the NL prompt handles the equivalent sorting logic naturally via explicit tie-breaking rules.

Both strategies use a two-phase generation structure: the model first emits a `<thinking>` chain-of-thought scratchpad (Wei et al., 2022) (ungraded), then encloses its final answer in `<json>...</json>` tags as an array of step dictionaries. Complete prompt templates are provided in Appendix B.

3.5. Evaluation Metrics

Step-Exact-Match (SEM). The `<thinking>` scratchpad is not graded; only the content within `<json>...</json>` tags is evaluated. The evaluable step count is:

$$N_{\text{eval}} = \max\big(|\text{LLM}_{\text{steps}}|, |\text{GT}_{\text{steps}}|\big) \quad (1)$$

Early termination assigns $\mathrm{SEM}_i = 0$ for all missing steps; extra hallucinated steps are compared against $\{\}$. Step i earns $\mathrm{SEM}_i = 1$ only when *every* sub-field matches simultaneously:

$$\mathrm{SEM}_i = \mathbb{1}\left[\mathtt{fr}_i^{\mathrm{LLM}} = \mathtt{fr}_i^{\mathrm{GT}}\right] \wedge \mathbb{1}\left[\mathtt{vis}_i^{\mathrm{LLM}} = \mathtt{vis}_i^{\mathrm{GT}}\right] \wedge \mathbb{1}\left[\mathtt{nbrs}_i^{\mathrm{LLM}} = \mathtt{nbrs}_i^{\mathrm{GT}}\right] \quad (2)$$

where `fr` denotes the frontier (`queue` for BFS, `stack` for DFS, `priority_queueUdistances` for Dijkstra), and `vis` denotes `visited`. The overall SEM score is:

$$\mathrm{SEM} = \frac{1}{N_{\mathrm{eval}}} \sum_{i=1}^{N_{\mathrm{eval}}} \mathrm{SEM}_i \quad (3)$$

Matching is positional: LLM step i is compared only against GT step i, so a step-order error at position k cascades forward.

Final Answer Correctness (FAC). FAC measures whether the model's output encodes the correct algorithmic result, independent of intermediate correctness. For BFS and DFS, FAC $=1$ iff the model's `traversal_order` exactly matches ground truth. For Dijkstra, FAC $=1$ iff the final `distances` dictionary assigns the correct shortest-path distance to every node.

The **SEM–FAC gap** (FAC $-$ SEM) is our primary diagnostic for *algorithmic reward hacking*: a large positive gap indicates a model produces a correct surface answer through incorrect internal mechanics.

Output generation and parsing. All runs used a maximum output length of 4,096 tokens. For any configurations that hit this limit, we re-ran with 8,192 tokens. The grader extracts content between `<json>` and `</json>` tags, falling back to markdown ```` ```json ```` fences if absent. Parse failures are scored as zero.

4. Experiments

Model and setup. To demonstrate the utility of the GraphStateEval framework, we conduct a pilot study using **Gemma 3 12B** (Gemma Team, 2025), a recent open-weights instruction-tuned model, evaluated at temperature 0 for deterministic outputs. This serves as a proof of concept to illustrate the types of insights, such as algorithmic reward hacking, that step-level tracing can reveal. Each cell corresponds to one graph instance (topology, N, algorithm, strategy) configuration. All scoring is fully automated; no human annotation is used at evaluation time. These are single-seed results; variability across seeds is a noted limitation.

Important Caveat: Single-Seed Evaluation. The experiments presented below evaluate exactly one graph instance per configuration (single-seed). Because algorithmic trace complexity is highly sensitive to the exact edge placements in random and irregular topologies, performance can exhibit significant instance-level variance. Consequently, these results should be interpreted as an exploratory pilot study illustrating qualitative failure modes and the utility of the GraphStateEval framework, rather than definitive quantitative benchmarks.

5. Results and Analysis

5.1. BFS: Reward Hacking and the SEM–FAC Gap

Path graphs. Path graphs achieve $\mathrm{SEM} = 1.00$ at every scale under both NL and Code strategies (Table 1). A path graph's BFS queue never holds more than one live entry at a time, keeping state management at constant complexity.

Complete graphs and the Code advantage. Complete graphs (Code) maintain $\mathrm{SEM} = 1.00$ through $N = 10$, collapsing only at $N = 11$–12. Despite the branching factor of $N-1$ immediately flooding the queue, the Code strategy reproduces the queue state precisely at larger scales. We attribute this to the complete graph's regularity: once the visited set is known, the next queue state is fully determined by the canonical ascending-sort rule, which the Python code prompt makes explicit. The NL strategy degrades earlier (falling to $\mathrm{SEM} = 0.27$ by $N = 10$), suggesting that the explicit code idiom provides a useful anchor for state tracking.

Random graphs. Random topologies show diverging behaviour based on density. Sparse-random graphs exhibit steep SEM degradation early, dropping to 0.33 at $N = 5$ under both strategies. In contrast, dense-random graphs remarkably maintain near-perfect SEM through $N = 9$–10 before collapsing. While structural irregularity forces the model to track exact insertion sequences, the higher edge density in dense-random graphs drastically reduces the effective traversal depth, delaying the compounding of errors. Single-seed evaluation introduces instance-level variance; the anomalously high sparse-random (Code) value of 0.83 at $N = 11$ before dropping to 0.08 at $N = 12$ illustrates this variability.

Tree graphs. Trees achieve $\mathrm{SEM} = 1.00$ through $N = 5$ before degrading. While their strict acyclic hierarchy may provide more structural cues than general sparse-random graphs, the highly irregular and unpredictable branching of Prüfer trees causes severe compounding errors as the traversal depth increases. Consequently, the somewhat non-monotonic performance observed across scales is an expected result of the high instance-level structural variance inherent to uniformly random tree generation.

Table 1. BFS Step-Exact-Match (SEM). Colour key (all tables): **green** ≥ 0.80, yellow 0.40–0.79, *orange* 0.10–0.39, *red* < 0.10.

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.08*	*0.08*
complete (NL)	0.75	0.60	**1.00**	0.43	*0.38*	*0.33*	*0.30*	*0.27*	*0.25*	*0.23*
d.rnd (Code)	**1.00**	**1.00**	**1.00**	**0.86**	**0.88**	**1.00**	**1.00**	*0.09*	0.58	*0.31*
d.rnd (NL)	0.75	**0.80**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.15*	*0.13*
path (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	**1.00**	*0.33*	*0.29*	0.50	*0.22*	*0.20*	*0.00*	**0.83**	*0.08*
s.rnd (NL)	0.75	**1.00**	*0.33*	*0.29*	0.50	0.56	*0.18*	*0.18*	*0.15*	*0.13*
tree (Code)	**1.00**	**1.00**	**1.00**	0.57	0.62	*0.33*	*0.30*	*0.27*	*0.17*	*0.07*
tree (NL)	**1.00**	**1.00**	**1.00**	0.57	**1.00**	*0.33*	**1.00**	*0.18*	*0.17*	*0.23*

Table 2. BFS Final Answer Correctness (FAC).

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*
complete (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
d.rnd (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	**1.00**
d.rnd (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*
path (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**
s.rnd (NL)	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*
tree (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
tree (NL)	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**

The SEM–FAC gap: algorithmic reward hacking. Comparing Tables 1 and 2 reveals the central finding for BFS: FAC substantially exceeds SEM across most non-path topologies. For example, complete (NL) maintains FAC = 1.00 through N = 12 while SEM has degraded to 0.23; similarly, dense-random (NL) maintains FAC = 1.00 through N = 11 while SEM has already collapsed to 0.15. This gap arises because BFS level-order traversal is partially predictable from graph structure: a model that has internalised level-order patterns can reproduce the correct traversal sequence without correctly tracking the queue. A benchmark measuring only FAC would rate such a model highly—an overestimate of true step-level execution fidelity.

5.2. DFS: Backtracking and Recency Bias

DFS degrades substantially faster than BFS at equivalent topology and scale (Table 3). Path graphs remain at SEM $=1.00$ throughout. Complete graphs fall to 0.11 by $N=8$. Sparse and dense random topologies converge to low values from $N=6$ onward.

The DFS challenge is qualitatively distinct from BFS. Three interacting requirements combine: (i) applying the canonical descending-sort push rule at each step; (ii) tracking a stack that grows and shrinks non-monotonically; and (iii) simulating backtracking—returning to a node pushed several steps earlier. Requirement (iii) conflicts with the position-based attention biases observed in transformers (Dziri et al., 2023): the model must recall nodes that are *old* in the context.

The DFS SEM–FAC gap persists strongly in complete graphs, where FAC remains at 1.00 even at N=12 despite SEM falling to 0.15 (Table 4). This occurs because a DFS traversal on a complete graph with our canonical tie-breaking simply visits nodes in ascending numerical order, providing a trivial pattern for the model to guess without tracking the stack. However, for less regular topologies like trees and random graphs, FAC tends to degrade alongside SEM at larger scales. In these graphs, DFS traversal orders depend heavily on backtracking choices specific to the instance and cannot easily be recovered by structural pattern-matching once the stack discipline has been lost.

5.3. Dijkstra: Two-Structure Maintenance

Dijkstra presents a qualitatively distinct state-maintenance challenge in GraphStateEval, combining two sources of complexity absent in the other unweighted algorithms (Tables 5–6): (i) the priority queue must be maintained as a sorted list of $[d, v]$ pairs, so a single wrong insertion corrupts extraction order; (ii) the distance dictionary—covering all $|V|$ nodes—must be updated in-place at every relaxation.

Path (NL) is a notable exception: it sustains SEM $=1.00$ through $N=12$, whereas Path (Code) degrades sharply

Table 3. DFS Step-Exact-Match (SEM).

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**0.80**	0.60	0.50	*0.19*	*0.38*	*0.11*	*0.18*	*0.27*	*0.25*	*0.15*
complete (NL)	**0.80**	0.60	0.50	*0.29*	*0.11*	*0.11*	*0.20*	*0.18*	*0.17*	*0.08*
d.rnd (Code)	**0.80**	0.67	*0.33*	0.40	*0.22*	*0.11*	*0.30*	*0.18*	*0.17*	*0.08*
d.rnd (NL)	0.50	0.67	0.56	*0.29*	*0.25*	*0.22*	*0.10*	*0.18*	*0.17*	*0.15*
path (Code)	0.75	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	**0.80**	0.50	*0.14*	*0.25*	*0.33*	*0.20*	*0.13*	*0.17*	*0.33*
s.rnd (NL)	**1.00**	0.60	**0.86**	*0.14*	*0.14*	*0.22*	0.40	*0.27*	*0.07*	*0.15*
tree (Code)	**1.00**	**1.00**	*0.29*	*0.20*	0.62	*0.22*	*0.20*	*0.27*	*0.08*	*0.29*
tree (NL)	**1.00**	**1.00**	*0.17*	**1.00**	*0.12*	0.56	*0.30*	*0.27*	*0.25*	*0.31*

Table 4. DFS Final Answer Correctness (FAC).

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*	**1.00**	**1.00**	**1.00**
complete (NL)	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	**1.00**
d.rnd (Code)	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
d.rnd (NL)	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
path (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	**1.00**	*0.00*
s.rnd (NL)	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*
tree (Code)	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*
tree (NL)	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*

from N=6. We tentatively hypothesize that simulating the `heapq` tuple-key API might introduce spurious overhead absent from the NL description, which focuses solely on the logical update rule. However, further targeted experimentation is needed to definitively isolate the cause of this performance drop.

FAC also collapses for Dijkstra across complete and random topologies by N=8–10 (Table 6), though path and tree graphs maintain accuracy slightly longer. Interestingly, the SEM–FAC gap persists even in Dijkstra. Despite the requirement for exact numerical computation, the model frequently recovers the correct final distance vector even when priority-queue tracking collapses entirely (e.g., tree (NL) achieves FAC =1.00 at N=11 despite SEM =0.12). This suggests the model may abandon rigorous algorithmic simulation in favour of holistic distance estimation, reaffirming that FAC alone is an insufficient measure of faithful execution. Note the high variance in FAC at larger scales (binary 0/1 values at individual seeds), reinforcing the need for multi-seed evaluation in future work.

5.4. Prompting Strategy Comparison

Code prompting outperforms NL for BFS on complete graphs (SEM =1.00 vs. 0.27 at N=10). For Dijkstra on path graphs, NL outperforms Code (1.00 vs. $\leq$0.09 at N=8–10). For DFS, neither strategy consistently dominates. This strategy–algorithm interaction shows that the optimal framing depends on the algorithm: Code helps when the idiom is simple and unambiguous; NL helps when the Code idiom introduces spurious complexity (e.g., `heapq` tuple invariant simulation).

5.5. Cross-Algorithm Difficulty

Across topologies and scales, the observed algorithm difficulty is highly topology-dependent. BFS is consistently the easiest, as it maintains a single data structure and its traversal order is partially predictable from graph structure. Dijkstra and DFS present distinct, non-comparable failure modes. Dijkstra collapses on complete graphs (due to dual-structure maintenance and arithmetic complexity) and path graphs under Code prompting (due to idiom overhead). Conversely, DFS proves exceptionally fragile on trees and sparse-random graphs because it requires potentially deep, non-monotonic stack backtracking, compounding error accumulation when the model must recall nodes that are old in the context.

5.6. Topology as a Complexity Dimension

Across all three algorithms, topology is a primary complexity axis orthogonal to graph scale. However, there is no universal ordering of difficulty: while path graphs are consistently the most tractable (often maintaining SEM =1.00 through N=12), the relative difficulty of tree, complete,

Table 5. Dijkstra Step-Exact-Match (SEM).

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**1.00**	0.40	**1.00**	**1.00**	*0.25*	*0.22*	*0.00*	*0.09*	*0.08*	*0.00*
complete (NL)	**1.00**	*0.00*	**1.00**	0.75	*0.25*	*0.09*	*0.10*	*0.09*	*0.08*	*0.08*
d.rnd (Code)	**1.00**	**1.00**	**1.00**	**0.88**	*0.25*	*0.00*	*0.30*	*0.18*	*0.00*	*0.08*
d.rnd (NL)	*0.25*	*0.00*	**0.83**	**0.88**	*0.22*	*0.22*	*0.10*	*0.09*	*0.08*	*0.08*
path (Code)	**1.00**	**1.00**	**1.00**	*0.14*	*0.12*	*0.00*	*0.00*	*0.09*	*0.08*	*0.15*
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	0.60	0.57	0.57	0.44	0.56	*0.30*	*0.31*	*0.25*	*0.00*
s.rnd (NL)	*0.00*	0.60	*0.14*	*0.10*	*0.25*	*0.11*	*0.30*	*0.27*	*0.17*	*0.15*
tree (Code)	*0.25*	**1.00**	**1.00**	0.71	0.62	0.44	*0.30*	*0.18*	*0.17*	*0.31*
tree (NL)	0.75	*0.00*	**1.00**	**1.00**	*0.25*	0.44	*0.20*	*0.18*	*0.12*	*0.23*

Table 6. Dijkstra Final Answer Correctness (FAC).

Topology (Strat.)	3	4	5	6	7	8	9	10	11	12
complete (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
complete (NL)	**1.00**	*0.00*	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*
d.rnd (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*	*0.00*
d.rnd (NL)	**1.00**	*0.00*	**1.00**	**1.00**	*0.00*	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*
path (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	**1.00**	**1.00**	**1.00**
path (NL)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
s.rnd (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**	*0.00*	*0.00*
s.rnd (NL)	*0.00*	**1.00**	**1.00**	**1.00**	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*	*0.00*
tree (Code)	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*	**1.00**
tree (NL)	**1.00**	*0.00*	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	*0.00*

and random topologies varies wildly by algorithm and edge density.

For instance, while sparse-random graphs degrade rapidly across all algorithms due to structural irregularity, dense-random graphs prove incredibly resilient in BFS due to their shallow traversal depth. Complete graphs exhibit an interesting profile: maximal branching floods the frontier immediately, yet their perfect regularity allows the Code strategy to maintain state at much larger scales than sparse-random graphs in BFS.

6. Discussion

Implications for evaluation. Our results illustrate that answer-level evaluation alone can overestimate LLM competence on algorithmic tasks. A model scoring well on FAC across most non-path topologies (such as complete or dense-random graphs) may be achieving this through structural pattern-matching or holistic estimation rather than correct algorithmic maintenance. Step-level metrics like SEM provide a more diagnostic view of actual execution fidelity.

Context maintenance versus algorithmic logic. The collapse in Dijkstra traces at larger scales highlights a critical intersection in LLM evaluation: the boundary between algorithmic logic failure and context-maintenance failure. Because our framework requires maintaining large state representations, performance degradation captures both structural reasoning limits and in-context state maintenance degradation. Disentangling these two factors is a primary direction for future work.

Dynamic benchmarking. The parametric nature of GraphStateEval's generator directly addresses benchmark saturation. As models improve, the benchmark can be extended to larger N, denser graphs, or new algorithms without human annotation, making it resistant to being folded into training data.

Single-seed and single-model limitations. The high instance-level variance observed in individual cells (e.g., sparse-random Code SEM $= 0.83$ at $N = 11$ for BFS, binary FAC values in Dijkstra at large scales) underscores that multi-seed evaluation is necessary for precise threshold estimation. Furthermore, future iterations of the benchmark will require evaluations across distinct model families, such as autoregressive versus reasoning-focused architectures, to establish universal baselines. The current results should be interpreted as exploratory findings demonstrating the framework's utility rather than definitive performance benchmarks.

Token budget and extended runs. Re-running previously truncated configurations at 8,192 tokens confirmed that the

SEM collapse is a genuine algorithmic failure rather than a budget artifact.

7. Conclusion

We presented GraphStateEval, a step-by-step evaluation framework for graph algorithm execution in LLMs. Three principal findings emerge from our evaluation of Gemma 3 12B. (i) **Topology is an independent complexity axis**: path graphs sustain near-perfect tracing throughout; sparse-random topologies degrade sharply early across algorithms, whereas dense-random and complete graphs often remain highly resilient up to $N{\approx}10$ in BFS due to shallower traversals and canonical regularities. (ii) **Answer-level evaluation masks actual execution fidelity**: the persistent SEM-FAC gap across BFS, DFS, and even Dijkstra illustrates algorithmic reward hacking that is entirely invisible to FAC-only benchmarks, showing that models can reliably guess correct final answers without faithfully maintaining valid intermediate data-structure states. (iii) **Algorithm difficulty is mechanism- and topology-dependent**: BFS is consistently the most tractable. Dijkstra fails under maximal branching or complex code idioms due to its dual-structure arithmetic demands, while DFS suffers from severe context-recall collapse during the potentially deep, non-monotonic stack backtracking required by trees and sparse-random graphs.

Future work. Extending GraphStateEval to multiple models and multiple seeds per configuration is the most pressing next step. Additional algorithms (Bellman-Ford, Prim, topological sort), larger graphs, and investigation of fine-tuning on GraphStateEval traces are further directions.

Impact Statement

This paper presents a diagnostic evaluation framework for algorithmic reasoning. While advancing LLM capabilities broadly carries significant societal implications, this specific benchmarking tool is designed for evaluation and does not directly introduce new negative societal impacts.

AI Disclosure

The authors acknowledge the use of AI tools such as ChatGPT, Claude, and Gemini for improving the presentation and grammar of this paper. All results, analyses, and proposed methods are solely the authors' contributions. The authors take full responsibility for the content of this paper.

References

Dziri, N., Lu, X., Sclar, M., Li, X. L., Jiang, L., Lin, B. Y., Welleck, S., West, P., Bhagavatula, C., Bras, R. L., Hwang, J. D., Sanyal, S., Ren, X., Ettinger, A., Harchaoui, Z., and Choi, Y. Faith and fate: Limits of transformers on compositionality. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Fatemi, B., Halcrow, J., and Perozzi, B. Talk like a graph: Encoding graphs for large language models. In *International Conference on Learning Representations*, 2024.

Gemma Team. Gemma 3 technical report. Technical report, Google DeepMind, 2025. arXiv:2503.19786.

Gong, D. and Zhang, H. Self-attention limits working memory capacity of transformer-based models, 2024. arXiv preprint arXiv:2409.10715.

Guo, J., Du, L., Liu, H., Zhou, M., He, X., and Han, S. GPT4Graph: Can large language models understand graph structured data? An empirical evaluation and benchmarking, 2023. arXiv preprint arXiv:2305.15066.

Markeeva, L., McLeish, S., Ibarz, B., Bounsi, W., Kozlova, O., Vitvitskyi, A., Blundell, C., Goldstein, T., Schwarzschild, A., and Veličković, P. The CLRS-text algorithmic reasoning language benchmark. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*. PMLR, 2024.

Merrill, W. and Sabharwal, A. The expressive power of transformers with chain of thought. In *International Conference on Learning Representations*, 2024.

Patil, P., Kumar, D., Sinha, Y., and Mandal, M. Beyond accuracy: Diagnosing algebraic reasoning failures in LLMs across nine complexity dimensions, 2026. arXiv preprint arXiv:2604.06799.

Perozzi, B., Fatemi, B., Zelle, D., Tsitsulin, A., Kazemi, M., Al-Rfou, R., and Halcrow, J. Let your graph do the talking: Encoding structured data for LLMs, 2024. arXiv preprint arXiv:2402.05862.

Taylor, A. K., Cuturrufo, A., Yathish, V., Ma, M. D., and Wang, W. Are large-language models graph algorithmic reasoners?, 2024. arXiv preprint arXiv:2410.22597.

Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. The CLRS algorithmic reasoning benchmark. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 22084–22102. PMLR, 2022.

Wang, H., Feng, S., He, T., Tan, Z., Han, X., and Tsvetkov, Y. Can language models solve graph problems in natural language? In *Advances in Neural Information Processing Systems*, volume 36, pp. 2965–2979, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022.

A. Canonical Tie-Breaking Rules: Worked Examples

BFS: 5-Node Path Graph (source = 1)

```
Step 0: action=initialize, queue=[1], visited=[1]
Step 1: node_dequeued=1, neighbors_enqueued=[2],
        queue=[2], visited=[1,2]
Step 2: node_dequeued=2, neighbors_enqueued=[3],
        queue=[3], visited=[1,2,3]
Step 3: node_dequeued=3, neighbors_enqueued=[4],
        queue=[4], visited=[1,2,3,4]
Step 4: node_dequeued=4, neighbors_enqueued=[5],
        queue=[5], visited=[1,2,3,4,5]
Step 5: node_dequeued=5, neighbors_enqueued=[],
        queue=[], visited=[1,2,3,4,5]
traversal_order: [1,2,3,4,5]
```

DFS: 4-Node Graph (edges 1-2, 1-3, 2-4; source = 1)

Neighbours of node 1 are {2,3}; sorted descending → push order [3,2]; 2 ends up at top.

```
Step 0: node_popped=null, stack=[1], visited=[]
Step 1: node_popped=1, neighbors_pushed=[3,2],
        stack=[3,2], visited=[1]
Step 2: node_popped=2, neighbors_pushed=[4],
        stack=[3,4], visited=[1,2]
Step 3: node_popped=4, neighbors_pushed=[],
        stack=[3], visited=[1,2,4]
Step 4: node_popped=3, neighbors_pushed=[],
        stack=[], visited=[1,2,3,4]
traversal_order: [1,2,4,3]
```

Dijkstra: 3-Node Graph (1-2 w=4, 1-3 w=1, 2-3 w=2; source = 1)

Note: edge weights in this example are chosen for illustrative clarity and do not follow the w(u,v) deterministic formula used in benchmark generation.

```
Step 0: node_extracted=null, priority_queue=[[0,1]],
        distances={"1":0,"2":"inf","3":"inf"}, visited=[]
Step 1: node_extracted=1, neighbors_relaxed=[2,3],
        priority_queue=[[1,3],[4,2]],
        distances={"1":0,"2":4,"3":1}, visited=[1]
Step 2: node_extracted=3, neighbors_relaxed=[2],
        priority_queue=[[3,2],[4,2]],
        distances={"1":0,"2":3,"3":1}, visited=[1,3]
Step 3: node_extracted=2, neighbors_relaxed=[],
        priority_queue=[[4,2]],
        distances={"1":0,"2":3,"3":1}, visited=[1,2,3]
traversal_order: [1,3,2]
```

B. Prompt Templates

This appendix provides the complete zero-shot prompt templates used for all evaluations across BFS, DFS, and Dijkstra. Both strategies share a common output format specification and JSON schema, which is provided first.

Shared Output Format and Schema

Both strategies instruct the model to output a <thinking> chain-of-thought scratchpad followed by a final <json> block. The JSON schemas and field rules vary slightly by algorithm to accommodate the required state trackers.

BFS JSON SCHEMA AND RULES

```
{
  "source": <int>,
  "steps": [
    {
      "step": 0,
      "action": "initialize",
      "queue": [<int>],
      "visited": [<int>]
    },
    {
      "step": <int>,
      "node_dequeued": <int>,
      "neighbors_enqueued": [<int>],
      "queue": [<int>],
      "visited": [<int>]
    }
  ],
  "traversal_order": [<int>]
}

Field rules:
- The "steps" array contains one object per step: step 0 followed by exactly one \
object for every dequeue operation, in order. Do not skip any steps.
- Step 0 has the fields: "step" (0), "action" ("initialize"), "queue", "visited". \
The "action" field appears ONLY in step 0 and nowhere else.
- Steps 1 onward have the fields: "step", "node_dequeued", "neighbors_enqueued", \
"queue", "visited". Every dequeue operation produces exactly one step object, even \
if no neighbors were enqueued.
- "queue" is the full queue state AFTER the step completes, preserving FIFO order \
(left = front). Use [] if the queue is empty.
- "visited" is the full visited set AFTER the step completes, sorted ascending. \
Use [] if the visited set is empty.
- "neighbors_enqueued" lists only the neighbors enqueued THIS step, in ascending \
order. Use [] if none were enqueued.
- "traversal_order" is the list of node_dequeued values across all steps after \
step 0, in order.
```

DFS JSON SCHEMA AND RULES

```
{
  "source": <int>,
  "steps": [
    {
      "step": 0,
      "node_popped": null,
      "neighbors_pushed": [],
      "stack": [<int>],
      "visited": []
    },
    {
      "step": <int>,
      "node_popped": <int>,
      "neighbors_pushed": [<int>],
      "stack": [<int>],
      "visited": [<int>]
    }
  ],
```

```
  "traversal_order": [<int>]
}

Field rules:
- The "steps" array contains one object per step: step 0 followed by exactly one \
object for every successful (unvisited) pop operation, in order. Do not skip any steps.
- Step 0 has fixed values: "node_popped" is always null, "neighbors_pushed" is always \
[], "visited" is always []. Only "stack" and "step" vary. The "node_popped": null and \
"neighbors_pushed": [] fields appear ONLY in step 0.
- Steps 1 onward have the fields: "step", "node_popped", "neighbors_pushed", "stack", \
"visited". Every successful pop produces exactly one step object, even if no neighbors \
were pushed.
- "stack" is the full stack state AFTER the step completes, preserving LIFO order \
(right = top of stack). Use [] if the stack is empty.
- "visited" is the full visited set AFTER the step completes, sorted ascending. \
Use [] if the visited set is empty.
- "neighbors_pushed" lists only the neighbors pushed onto the stack THIS step, in the \
exact order they were pushed (descending numerical order, largest first). Use [] if \
none were pushed.
- "traversal_order" is the list of node_popped values across all steps after step 0, \
in order.
```

DIJKSTRA JSON SCHEMA AND RULES

```
{
  "source": <int>,
  "steps": [
    {
      "step": 0,
      "node_extracted": null,
      "neighbors_relaxed": [],
      "priority_queue": [[<int>]],
      "distances": {"<node_id>": <int or "inf">},
      "visited": []
    },
    {
      "step": <int>,
      "node_extracted": <int>,
      "neighbors_relaxed": [<int>],
      "priority_queue": [[<int>]],
      "distances": {"<node_id>": <int or "inf">},
      "visited": [<int>]
    }
  ],
  "traversal_order": [<int>]
}

Field rules:
- The "steps" array contains one object per step: step 0 followed by exactly one \
object for every successful (unvisited) extraction from the priority queue, in order. \
Do not skip any steps.
- Step 0 has fixed values: "node_extracted" is always null, "neighbors_relaxed" is \
always [], "visited" is always []. "priority_queue" contains [[0, source]] and \
"distances" has source=0 and all others="inf". These fixed null/[] values appear \
ONLY in step 0 and nowhere else.
- Steps 1 onward have the fields: "step", "node_extracted", "neighbors_relaxed", \
"priority_queue", "distances", "visited". Every successful extraction produces exactly \
one step object, even if no neighbors were relaxed.
- "priority_queue" is the full priority queue state AFTER the step completes, \
as a list of two-element lists [distance, node]  NOT tuples. Sorted ascending by \
distance first, then ascending by node ID to break ties. A node may appear multiple \
times with different distances  include ALL entries, do not deduplicate. \
Use [] if the priority queue is empty.
- "distances" maps every node ID as a string key (e.g. "1", "2") to its current \
```

```
shortest distance as an integer, or the string "inf" if not yet reached. Every node \
must appear as a key in every step.
- "visited" is the full set of finalized nodes AFTER the step completes, sorted \
ascending. Use [] if empty.
- "neighbors_relaxed" lists only the neighbors whose distance was strictly improved \
this step (alt < current distance) and were pushed into the priority queue. Neighbors \
already visited, or whose distance was not improved, are NOT included. List in \
ascending node order. Use [] if none.
- "traversal_order" is the list of node_extracted values across all steps after \
step 0, in order.
```

Strategy A — Natural Language Prompts

BFS NATURAL LANGUAGE PROMPT

```
You are given an undirected graph and a source node. \
Simulate Breadth-First Search (BFS) from the source node and record the state after every
    step.

---
GRAPH:
Nodes: [NODES]
Edges: [EDGES]
Source node: [SOURCE]

---
BFS RULES:

1. INITIALIZATION (Step 0): enqueue the source node and mark it visited. \
Record this as step 0.

2. EACH STEP: dequeue the front node u. Filter the neighbors of u to only those \
not yet in the visited set. Sort this filtered list in ascending numerical order. \
Enqueue them in that order, marking each visited at the moment of enqueueing \
(not on dequeue). Record this dequeue operation as a step regardless of whether \
any neighbors were enqueued.

3. Repeat step 2 until the queue is empty.

---
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```

DFS NATURAL LANGUAGE PROMPT

```
You are given an undirected graph and a source node. \
Simulate iterative Depth-First Search (DFS) from the source node and record the state \
after every successful step.

---
GRAPH:
Nodes: [NODES]
Edges: [EDGES]
Source node: [SOURCE]

---
DFS RULES:

1. INITIALIZATION (Step 0): push the source node onto an empty stack. The visited set \
is empty  do NOT mark the source as visited yet. Record this as step 0.

2. EACH STEP:
  a. Pop the node at the top (right end) of the stack.
  b. If the popped node is already in the visited set, discard it silently  do NOT \
record this as a step  and pop the next node. Repeat until you pop a node that is \
not in the visited set. Call this node u.
  c. Mark u as visited.
  d. Filter the neighbors of u to only those not yet in the visited set. Sort this \
filtered list in DESCENDING numerical order (largest first). Push them onto the stack \
one by one in that order, so the smallest unvisited neighbor ends up at the top.
  e. Record this pop operation as a step regardless of whether any neighbors were pushed.

3. Repeat step 2 until the stack is completely empty.

---
```

```
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```

DIJKSTRA NATURAL LANGUAGE PROMPT

```
You are given an undirected weighted graph and a source node. \
Simulate Dijkstra's Algorithm from the source node and record the state after every \
successful step.

---
GRAPH:
Nodes: [NODES]
Edges (node1, node2, weight): [EDGES]
Source node: [SOURCE]

---
DIJKSTRA RULES:

1. INITIALIZATION (Step 0): set distance to source = 0 and all other nodes to "inf". \
Push [0, source] onto an empty priority queue. The visited set is empty. \
Record this as step 0.

2. EACH STEP:
  a. Pop the [distance, node] pair with the smallest distance from the priority queue. \
Ties in distance are broken by smallest node ID. If the popped node is already in the \
visited set, discard this pair silently  do NOT record it as a step  and pop again. \
Repeat until you pop a node not in the visited set. Call this node u and its popped \
distance dist(u).
  b. Mark u as visited (its shortest path is now locked in).
  c. Filter the neighbors of u to only those not yet in the visited set. Process this \
filtered list in ascending numerical order.
  d. For each such neighbor v: calculate alt = dist(u) + weight(u, v). If alt is \
strictly less than the current recorded distance for v, update the distance for v to \
alt, push [alt, v] onto the priority queue, and add v to neighbors_relaxed for this \
step. Do NOT remove or replace any existing entries for v already in the priority \
queue   simply push the new pair alongside them.
  e. Record the state of the priority queue, distances, and visited set right now. \
Do this regardless of whether any neighbors were relaxed.

3. Repeat step 2 until the priority queue is completely empty.

---
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```

Strategy B — Source Code Prompts

BFS SOURCE CODE PROMPT

```
You are given an undirected graph, a source node, and a Python implementation \
of BFS. Simulate the execution of this exact code and record the state at each \
iteration of the while loop.

---
GRAPH:
Nodes: [NODES]
Edges: [EDGES]
Source node: [SOURCE]

Adjacency list (exactly as the code receives it  all lists pre-sorted ascending):
[ADJACENCY_LIST]

---
CODE (simulate this exactly):

from collections import deque

def bfs(graph, source):
   visited = set()
   queue = deque()

   # Step 0: initialization
   queue.append(source)
   visited.add(source)

   step = 1
   while queue:
      u = queue.popleft()          # dequeue front
      neighbors_enqueued = []

      for v in graph[u]:           # neighbors already sorted ascending
         if v not in visited:
            visited.add(v)         # mark visited ON ENQUEUE
            queue.append(v)
            neighbors_enqueued.append(v)

      # --- record state after this iteration ---
      # step          = step
      # node_dequeued = u
      # neighbors_enqueued = neighbors_enqueued
      # queue         = list(queue) (left = front)
      # visited       = sorted(visited)

      step += 1

---
TASK:
Simulate bfs(graph=[ADJACENCY_LIST], source=[SOURCE]) step by step.
Do not deviate from the code. Trace every iteration of the while loop.

---
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```

DFS SOURCE CODE PROMPT

```
You are given an undirected graph, a source node, and a Python implementation \
of iterative DFS. Simulate the execution of this exact code and record the state \
at each successful loop iteration.
```

```
---
GRAPH:
Nodes: [NODES]
Edges: [EDGES]
Source node: [SOURCE]

Adjacency list (exactly as the code receives it  all lists pre-sorted ascending):
[ADJACENCY_LIST]

---
CODE (simulate this exactly):

def dfs(graph, source):
   visited = set()
   stack = [source]

   # Step 0: initialization
   # (record step 0 here: stack = [source], visited = [])

   step = 1
   while stack:
      curr = stack.pop()           # pop from right end (top)

      # Mark-on-Pop: skip nodes already visited
      if curr in visited:
         continue

      visited.add(curr)

      # Filter unvisited neighbors, sort descending, push largest first
      # so the smallest unvisited neighbor ends up at the top of the stack
      unvisited_neighbors = [v for v in graph[curr] if v not in visited]
      neighbors_to_push = sorted(unvisited_neighbors, reverse=True)

      for v in neighbors_to_push:
         stack.append(v)

      # --- record state exactly here, after this iteration ---
      # step         = step
      # node_popped  = curr
      # neighbors_pushed = neighbors_to_push (descending order, largest first)
      # stack        = list(stack)     (right = top)
      # visited      = sorted(visited)

      step += 1

---
TASK:
Simulate dfs(graph=[ADJACENCY_LIST], source=[SOURCE]) step by step.
Do not deviate from the code. Trace every iteration of the while loop, skipping \
iterations where curr is already in visited without recording them as steps.

---
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```

DIJKSTRA SOURCE CODE PROMPT

```
You are given an undirected weighted graph, a source node, and a Python implementation \
of Dijkstra's algorithm. Simulate the execution of this exact code and record the state \
at each successful loop iteration.

---
```

```
GRAPH:
Nodes: [NODES]
Edges: [EDGES]
Source node: [SOURCE]

Adjacency list with weights (nodes are keys, values are lists of (neighbor, weight) \
tuples, neighbors already sorted ascending):
[ADJACENCY_LIST]

---
CODE (simulate this exactly):

import heapq
import math

def dijkstra(graph, source, all_nodes):
   distances = {{u: math.inf for u in all_nodes}}
   distances[source] = 0

   pq = [(0, source)]
   visited = set()

   # Step 0: initialization
   # (record step 0 here: pq = [(0, source)], visited = [],
   # distances = inf for all nodes except source = 0)

   step = 1
   while pq:
      dist, u = heapq.heappop(pq) # pops smallest (dist, u); ties broken by smallest u

      # Mark-on-Pop: skip nodes already finalized
      if u in visited:
         continue

      visited.add(u)
      relaxed_neighbors = []

      # graph[u] is a list of (v, weight) tuples, already sorted by v ascending
      for v, w in graph[u]:
         if v in visited:
            continue

         alt = dist + w
         if alt < distances[v]:
            distances[v] = alt
            heapq.heappush(pq, (alt, v))
            relaxed_neighbors.append(v) # appended in ascending order since
                                        # graph[u] is pre-sorted ascending

      # --- record state exactly here, after this iteration ---
      # step         = step
      # node_extracted = u
      # neighbors_relaxed = relaxed_neighbors (ascending order)
      # priority_queue = sorted(list(pq)) (as [[dist, node], ...] sorted ascending)
      # distances    = dict(distances)    (math.inf rendered as "inf")
      # visited      = sorted(visited)

      step += 1

---
TASK:
Simulate dijkstra(graph=[ADJACENCY_LIST], source=[SOURCE]) step by step.
Do not deviate from the code. Trace every iteration of the while loop, skipping \
iterations where u is already in visited without recording them as steps.
```

```
---
OUTPUT FORMAT:
[SHARED OUTPUT FORMAT BLOCK INSERTED HERE]\
```