WHEN TO COMPRESS: EFFICIENT LLM REASONING ON GRAPHS WITH ONE BILLION EDGES

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Models (LLMs) can handle natural-language requests involving graph reasoning tasks by invoking appropriate tools. On large-scale graphs, however, tool execution can involve substantial computation unrelated to the requested answer. Graph compression can reduce this computation, but how it is performed affects both answer correctness and execution efficiency. Even for the same task, the information needed to determine the answer can differ substantially in scope and size across graph instances. Moreover, constructing a smaller answer-preserving graph may cost more than the subsequent execution saves. This raises the question of when and how to compress a graph. We thus introduce a Certificate graph, an executable graph representation that preserves the original task answer, and propose CAGR, a Certificate-Aware Graph Reasoning framework that learns how to obtain such representations and determines when further compression is worthwhile. CAGR learns compression strategies through Certificate Compression Optimization (CCO), which combines toolchain search with LLM policy improvement, while a lightweight Compression Gate weighs further compression costs against subsequent execution savings to determine when to execute. We also introduce GraphLarge, which extends existing LLM graph reasoning benchmarks to the billion-edge scale and covers 28 domain-task settings across four real-world domains. Results show that CAGR achieves a 5.78$\times$ speedup and improves average success rate by 15.53 percentage points over existing graph compression methods.

1 INTRODUCTION

Graph problems such as flow computation, path planning, and connectivity analysis are fundamental to graph analysis, supporting the study of structural relationships and the optimization of network operations (Ahuja et al., 1993). Across diverse domains (e.g., the Web, transportation, and social networks), these real-world problems are often posed as natural-language requests that describe practical needs without prescribing how they should be computed. This flexibility can obscure the underlying task: similar requests may describe different graph problems, while the same problem may be expressed in different ways. Large Language Models (LLMs) provide a new way to handle such requests: they identify the intended graph problem, map it to the corresponding graph operations, and invoke external tools to perform the corresponding graph computations (Zhang et al., 2025).

Existing LLM-based graph reasoning methods focus on interpreting requests and organizing tool execution. GraphTool-Instruction improves request understanding through task decomposition (Wang et al., 2025), while GCoder and GraphCogent enhance generalization through program generation and multi-agent collaboration, respectively (Zhang et al., 2025; Wang et al., 2026). As graphs grow larger, even the right tool may process substantial graph information unrelated to the current request. To narrow this scope, GraphChain learns sequences of graph-reduction tools using task feedback and relevance estimates (Wei et al., 2025). However, a compression strategy that succeeds during training may discard essential structures in a new instance. For example, retaining only the region around query nodes may preserve the shortest path in one graph but exclude it in another.

The graph information required to preserve the task answer therefore depends on the task and the graph instance. We introduce a **Certificate graph**, an executable graph representation that preserves the original task answer. Our observations in Section 3 show that, even for the same task and input graph scale, a compression plan can produce Certificate graphs of substantially different sizes.

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

Moreover, Figure 1 illustrates that constructing a smaller Certificate graph is worthwhile only when subsequent execution savings exceed its construction cost. This raises a central question: when and how should a graph be compressed for a given request?

We thus propose **CAGR**, a **C**ertificate-**A**ware **G**raph **R**easoning framework that learns to progressively construct smaller Certificate graphs and determines when further compression is worthwhile. An LLM-based Toolchain Constructor organizes compression and execution tools into a complete plan, preserving the task answer throughout graph reduction. Certificate Compression Optimization (CCO) alternates toolchain search and LLM policy improvement to learn how to compose and order these tools for effective compression. A lightweight Compression Gate weighs further construction costs against subsequent execution savings, determining when to stop compression and execute on the current graph.

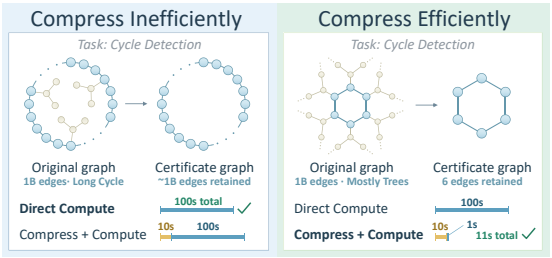


Figure 1: Two cases illustrate the tradeoff between Certificate graph construction and execution. Compression increases total time in one case but reduces it in the other, which raises the question of when to compress a graph.

To examine how broadly this challenge arises in large-scale graph reasoning, we introduce GraphLarge, which extends existing benchmarks to the billion-edge scale and covers 28 task settings across four real-world domains. Experiments show that CAGR achieves a 5.78× speedup and improves average success rate by 15.53 percentage points over existing graph compression methods. Further evaluations on public benchmarks also demonstrate its efficiency advantages and competitive success rates.

Our contributions are threefold:

- We introduce Certificate graphs as executable representations that preserve task answers, and show that both their achievable sizes and construction costs vary substantially across graph instances.
- We propose CAGR, an LLM-based graph reasoning framework that combines learned compression strategies with adaptive execution to determine how to construct Certificate graphs and when to stop compression.
- We introduce GraphLarge, extending LLM-based graph reasoning evaluation to the billion-edge scale across 28 task settings and four real-world domains, and validate CAGR's efficiency and effectiveness on GraphLarge and public benchmarks.

2 Related Work

LLM-based Graph Reasoning. Early studies investigate LLM-based graph reasoning through textual graph representations and instruction training (Wang et al., 2023; Fatemi et al., 2024; Chen et al., 2024; Luo et al., 2024). Tool-based methods extend these capabilities by delegating computation to graph algorithms. GraphTool-Instruction learns tool selection and argument extraction (Wang et al., 2025), GCoder generates executable programs (Zhang et al., 2025), and GraphTeam and GraphCogent organize multi-agent workflows (Li et al., 2024; Wang et al., 2026). GraphChain further learns multi-step tool sequences that progressively reduce the graph information involved in reasoning (Wei et al., 2025). CAGR focuses on constructing answer-preserving Certificate graphs and determining when their construction yields computational savings.

Graph Compression. Query-preserving graph compression reduces graph representations while preserving answers to a specified query class (Fan et al., 2012). Core decomposition and contraction-based methods exploit graph structure to support efficient computation (Batagelj & Zaversnik, 2003; Geisberger et al., 2008). These methods provide foundations for CAGR's compression tools. CAGR learns how to compose such tools through Certificate Compression Optimization and uses the Compression Gate to determine how much compression to execute. Certificate Complexity (Buhrman & de Wolf, 2002) provides the instance-level perspective underlying our Certificate graph formulation. Appendix A further discusses graph reasoning benchmarks and certificate-based perspectives.

3 When Is a Smaller Certificate Graph Worth Constructing?

3.1 Certificate Graph

The size of a graph does not directly indicate how much information is required to determine a task answer. Even for the same function, the output on one input may be determined by a few variables, whereas on another it may depend on most of the input. Certificate Complexity captures this variation by measuring the smallest part of an input that fixes its output (Buhrman & de Wolf, 2002). For a Boolean function $f : \{0,1\}^n \to \{0,1\}$ and an input $x = (x_1, \ldots, x_n)$, it is defined as

$$C_x(f) = \min_{S \subseteq [n]} |S| \quad \text{s.t.} \quad x'_S = x_S \Rightarrow f(x') = f(x),\ \forall x' \in \{0,1\}^n. \tag{1}$$

The constraint requires every input consistent with x on S to produce the same output. Thus, $C_x(f)$ measures the minimum number of input variables whose values determine $f(x)$.

Inspired by Certificate Complexity, we introduce a **Certificate graph** as an executable graph representation that preserves the answer to the original graph task. Let $\mathcal{H}_q$ denote the graph representations executable for request q. For an original graph G, we define the family of Certificate graphs as:

$$\mathcal{G}_q(G) = \{H \in \mathcal{H}_q \mid A_q(H) \equiv_q A_q(G)\}. \tag{2}$$

Here, $A_q(H)$ denotes the mathematical answer to request q on H, and $\equiv_q$ denotes answer equivalence after recovering the result in the original graph's representation when needed (Appendix C.1).

3.2 Observations

A Certificate graph can be progressively reduced through answer-preserving compression. We examine how Certificate graph size and total execution cost vary across instances and compression depths. Following GraphCogent's sampling protocol (Wang et al., 2026), we use multi-walk sampling to extract billion-edge subgraphs from the WDC Hyperlink Graph (Meusel et al., 2015), keeping the graph source and scale consistent. We construct 100 cases each for Shortest Path and Max Flow. Within each task, we apply a fixed sequence of five answer-preserving compression tools and measure Certificate graph size and total compression and execution time at each depth. Appendix B.3 details the experimental settings. Figure 2 reveals two observations.

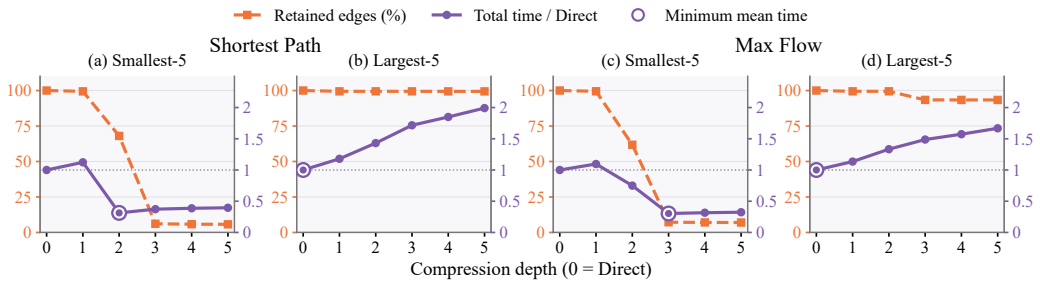


Figure 2: Certificate graph size and total time at different compression depths. For each task, Smallest-5 and Largest-5 average the five instances yielding the smallest and largest final Certificate graphs, respectively. Orange curves show the percentage of original edges retained (left axis). Purple curves show compression plus execution time divided by Direct execution time (right axis). Depth 0 represents Direct execution. Hollow circles mark the depth with the lowest mean time.

Observation 1: Certificate Graph Size Varies Across Instances. Figure 2 compares the five instances yielding the smallest and largest final Certificate graphs for each task. Despite using the same compression sequence, these groups differ substantially in the graph reduction achieved. For Shortest Path and Max Flow, compression removes most edges in the Smallest-5 groups but retains nearly the entire graph in the Largest-5 groups. The size of the constructed Certificate graph therefore depends on the graph instance, even for the same task and input graph scale.

Observation 2: A Smaller Certificate Graph Is Not Always Worth Constructing. Certificate graph size and total execution cost do not necessarily decrease together. In the Smallest-5 Shortest

Path group, total time is lowest at depth 2, while further compression produces smaller Certificate graphs at higher cost. In the Largest-5 Shortest Path and Max Flow groups, Direct execution remains fastest. However, the Smallest-5 groups also show that an initial cost increase can precede substantial execution savings. These results show that the computational benefit of Certificate graph construction depends on both the instance and the compression depth: further compression is worthwhile only when subsequent execution savings outweigh its additional cost.

4 GRAPHLARGE CONSTRUCTION

To better understand when and how to compress a graph before task execution, we introduce GraphLarge, which covers 28 task settings spanning 14 graph problem families across four real-world domains on graphs at the billion-edge scale. We describe its graph sources, construction process, and task design below.

Real-World Graph Construction. GraphLarge is constructed from four real-world graph sources: the WDC Hyperlink Graph for Web (Meusel et al., 2015), Overture Maps for Transportation, Friendster for Social (Yang & Leskovec, 2015), and MAG240M for Citation (Hu et al., 2021). We partition each source graph into three node-disjoint regions for training, validation, and test. Building on GraphCogent's random-walk sampling (Wang et al., 2026), we use multiple walks within each region to select nodes and retain all original edges between them. The resulting instances reach the billion-edge scale and retain available task-relevant attributes, including directions, weights, capacities, and entity and relation types.

Task Design. GraphLarge comprises 28 domain-task settings across four domains, with five in-distribution and two out-of-distribution tasks per domain. Expressed through domain-grounded requests, these tasks cover path finding, reachability, cycle detection, spanning forests, network flow, matching, and triangle counting. These tasks impose different requirements on graph structure, task-specific attributes, and the information needed to determine their answers.

Instance Construction. All instances use a common property-graph representation following the Neo4j GDS projection model, retaining task-relevant labels, relationship types, and properties. Each task contains 300 cases expressed through 10 natural-language request templates that vary phrasing and parameter placement without specifying algorithms or tools. In-distribution cases are split into 180 training, 60 validation, and 60 test cases per task; each held-out task contributes 300 test cases. Reference answers are computed directly on each sampled graph using the corresponding task-specific solver. Appendix D details the task inventory, graph sampling, request generation, and reference-answer construction.

5 METHOD

Motivated by the observations in Section 3, we propose CAGR, a **C**ertificate-**A**ware **G**raph **R**easoning framework for adaptive graph-tool composition and execution. As illustrated in Figure 3, CAGR comprises a Certificate-Aware Tool Library, an LLM-based Toolchain Constructor, and a Compression Gate. The library provides execution tools and Certificate compression tools. Given a graph task, the Constructor generates a complete toolchain consisting of an ordered sequence of compression tools and a selected execution tool. Each applicable compression step preserves the task answer, producing a Certificate graph at every prefix. The Compression Gate weighs further compression costs against subsequent execution savings to determine when to stop compression and invoke the execution tool.

5.1 CERTIFICATE-AWARE TOOL LIBRARY

A compression chain is useful only if it removes graph information without destroying the evidence required by the task. CAGR constructs a compression chain from compression tools that retain the graph evidence required to determine the task answer. Given a Graph problem q on graph G, the resulting toolchain takes the following form:

$$\pi : \quad G = H_0 \xrightarrow{C_1} H_1 \xrightarrow{C_2} \cdots \xrightarrow{C_K} H_K \xrightarrow{E} \widehat{a}. \tag{3}$$

Here, C_i is the i-th compression tool, H_i is the resulting Certificate graph, and E is the selected execution tool that returns a task result $\widehat{a}$. The planned compression chain contains K steps. We write

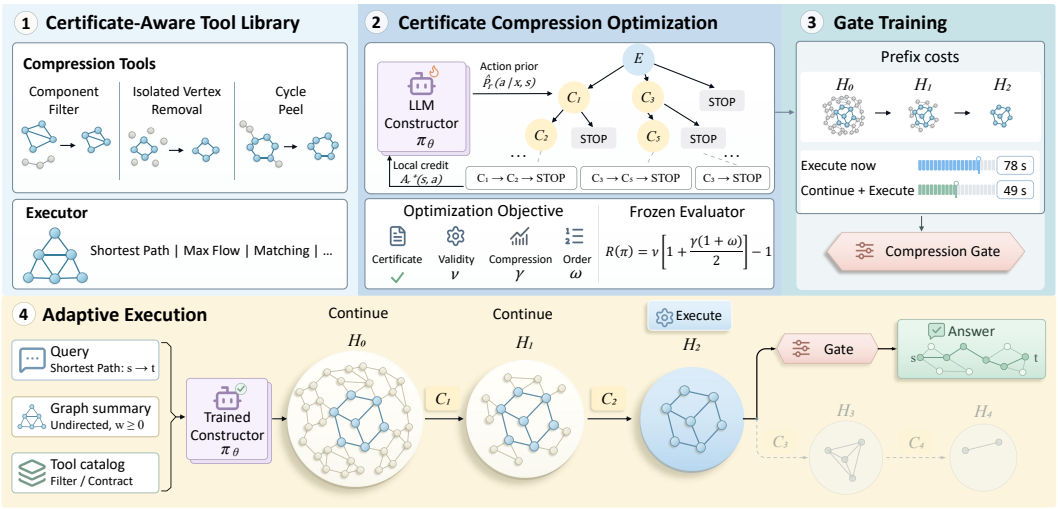


Figure 3: Overview of CAGR. Certificate Compression Optimization trains the LLM-based Toolchain Constructor to compose Certificate compression tools and an execution tool into a complete toolchain. The Compression Gate learns from prefix costs to determine when to stop compression and execute, balancing further compression costs against subsequent execution savings.

$\mathrm{Cert}_q(H; G)$ when H is a Certificate graph for task q on G: an executable graph representation that preserves the original task answer. In particular, $\mathrm{Cert}_q(G; G)$ holds.

Proposition 1 (Safe Prefix Execution). Suppose each compression tool C_i satisfies its applicability conditions and preserves both the Certificate condition and compatibility with the selected execution tool E. Then every prefix preserves the task answer and remains executable by E:

$$\mathrm{Cert}_q(H_i; G), \qquad A_q(H_i) \equiv_q A_q(G), \qquad \forall i \in \{0, \ldots, K\}. \tag{4}$$

The proposition follows by induction. The initial graph $H_0 = G$ contains sufficient evidence, and each compression tool preserves this property from H_{i-1} to H_i. Consequently, CAGR can stop after any compression tool, including before the first compression tool, and invoke the execution tool without changing the task answer.

CAGR implements this condition through Certificate compression tools, which preserve the task answer under their applicability conditions. The Constructor composes these tools with an execution tool into a toolchain that progressively constructs Certificate graphs for task execution. Appendix E details the tool library and preservation proofs.

5.2 Certificate Compression Optimization

We train the Toolchain Constructor through Certificate Compression Optimization (CCO), which learns tool selection and composition to construct smaller Certificate graphs. CCO iteratively alternates between Monte Carlo Tree Search (MCTS) over toolchains and LLM policy improvement (Anthony et al., 2017). The current policy guides the search for complete toolchains, and the resulting action values inform policy updates that guide the next round of search.

Compression Chain Search. The Constructor generates a complete toolchain π of the form shown in Eq. 3, selecting an execution tool and an ordered sequence of compression tools. Generating the complete chain in one call avoids repeated LLM inference during interactive construction, where selecting each tool based on the preceding execution result adds LLM latency to the cost of graph computation (Wei et al., 2025). Learning effective toolchains, however, requires exploring a combinatorial space of tool choices, orderings, and chain lengths. Direct sampling from the current policy may provide few high-quality toolchains, limiting the feedback available for policy improvement. CCO therefore uses policy-guided MCTS to explore alternative continuations from shared toolchain prefixes.

Let x denote the task, public graph summary, and tool library, and let s represent the partial toolchain. Search begins by selecting an execution tool, then extends the compression prefix by appending a compression tool or choosing STOP. At each state, the Constructor proposes actions conditioned on x and s, and their sampling frequencies define the action prior. Search explores these choices (Silver et al., 2017) using:

$$a^* = \arg\max_{a \in \mathcal{A}(s)} \left[Q_{\text{search}}(s,a) + c_{\text{puct}} \widehat{P}_r(a \mid x, s) \frac{\sqrt{N(s)}}{1 + N(s,a)} \right], \tag{5}$$

Here, $\mathcal{A}(s)$ contains the currently proposed actions, and c_{puct} controls exploration. $Q_{\text{search}}(s,a)$ is the mean terminal reward backed up through action a at state s, $\widehat{P}_r(a \mid x, s)$ is the empirical action prior computed from LLM proposal frequencies in iteration r, and $N(s)$ and $N(s,a)$ denote the corresponding visit counts. The first term favors tool decisions with higher observed returns, while the second explores promising but less visited decisions.

Certificate Compression Reward. CCO evaluates a toolchain through execution validity, final compression, and compression order. A valid toolchain produces a sequence of Certificate graphs, with $H_0 = G$ and $H_i = C_i(H_{i-1})$. Let $U_i = U(H_i)$ denote the graph size after the i-th compression tool. We consider chains satisfying $U_i \le U_{i-1}$ and define cumulative compression as $c_i = 1 - U_i/U_0$, with $c_0 = 0$. Final compression is $\gamma = c_K$, the fraction of the original graph size removed, satisfying $0 \le c_1 \le \cdots \le c_K = \gamma \le 1$.

Execution validity $\nu \in \{0, 1\}$ indicates whether the toolchain satisfies the task, Certificate, and representation contracts and produces a correct answer. A supported compression tool that safely leaves the graph unchanged remains valid. To favor earlier compression, we weight each step's additional compression by its position in the chain and define the normalized compression order ω and Certificate compression reward as:

$$\omega = \begin{cases} \dfrac{1}{\gamma} \sum_{i=1}^{K} \dfrac{c_i - c_{i-1}}{i}, & \gamma > 0, \\ 0, & \text{otherwise}, \end{cases} \qquad R(\pi) = \nu \left[1 + \frac{\gamma(1+\omega)}{2} \right] - 1. \tag{6}$$

Here, ν enforces toolchain validity, γ rewards final compression, and ω rewards earlier compression gains. Appending compression tools that leave the graph size unchanged does not increase the reward. Invalid toolchains receive -1, valid non-compressing chains receive 0, and valid chains with positive compression receive a positive reward.

Policy Improvement. A useful tool choice may receive a low average return when followed by poorly composed tools. CCO therefore uses the best observed return to identify decisions that support high-reward toolchains. Let $\mathcal{D}(s,a)$ contain the evaluated toolchains passing through action a at prefix s. We define the decision value and positive search advantage as:

$$B(s,a) = \max_{\pi \in \mathcal{D}(s,a)} R(\pi), \qquad A^+(s,a) = \max\{B(s,a) - V_B(s), 0\}, \tag{7}$$

where $V_B(s)$ is the prior-weighted average of the observed decision values at prefix s. While Q_{search} guides search, B identifies decisions to reinforce through policy improvement. We update the Constructor through Advantage Weighted Policy Improvement (AWPI), weighting each selected action's tokens by its normalized positive search advantage:

$$\mathcal{L}_{\text{AWPI}}(\theta) = - \sum_{(x,s,a) \in \mathcal{D}^+} \overline{A}^+(s,a) \sum_{t \in \text{span}(a)} \log p_\theta(y_t \mid y_{<t}, x). \tag{8}$$

Here, $\mathcal{D}^+$ contains the selected positive-advantage decisions, and $\text{span}(a)$ denotes the tokens assigned to action a in the generated toolchain. Appendix C details decision selection, token weighting, and loss normalization. The updated Constructor then guides the next round of toolchain search.

5.3 Adaptive Execution with a Compression Gate

CCO trains the Constructor to generate complete toolchains, but further compression does not always reduce execution time. Proposition 1 allows the selected tool to operate on the Certificate graph at any prefix without changing the task answer. The Compression Gate therefore determines when to

stop compression by weighing further compression costs against subsequent execution savings. At compression depth d, the Gate estimates the benefit of continuing from the current prefix using graph and tool information:

$$\mathbf{z}_d = \mathbf{g}_d \oplus \mathbf{p}_d, \qquad \widehat{y}_d = \mathrm{Gate}_\phi(\mathbf{z}_d), \tag{9}$$

where $\oplus$ denotes concatenation. Graph information $\mathbf{g}_d$ comprises the original graph summary, query-node statistics, and observed compression state. Tool information $\mathbf{p}_d$ describes the computational complexity of the remaining compression tools and the selected execution tool. A positive prediction executes the next compression tool, after which the Gate reassesses the updated prefix; a non-positive prediction invokes the execution tool on H_d. We implement the Compression Gate as a compact MLP for efficient prediction.

The Gate is trained separately from CCO using measured prefix costs. Its supervision compares execution at the current prefix with compression followed by execution at later prefixes, accounting for the intervening compression cost. The Constructor thus determines the proposed toolchain, while the Gate adapts its execution depth to the instance. Appendix C.4 provides the feature definitions, supervision target, and model architecture.

6 Experiments

6.1 Experimental Settings

Datasets. We evaluate CAGR on GraphLarge, GTools (Wang et al., 2025), and Graph4real (Wang et al., 2026). GraphLarge covers four real-world domains, with mean graph sizes ranging from 56.0M to 1.15B edges across domains. Following their graph generation protocols, we construct 1M-node instances for the 20 GTools tasks and 21 Graph4real tasks, with 100 cases per task. We further evaluate on smaller-scale benchmarks: NLGraph (Wang et al., 2023), Talk like a Graph (Fatemi et al., 2024), GraphWiz (Chen et al., 2024), GraphArena (Tang et al., 2025), GraCoRe (Yuan et al., 2025), and GraphInstruct (Luo et al., 2024). Appendix D details GraphLarge construction; Appendix B provides additional benchmark results.

Baseline Methods. We compare CAGR with GraphTeam (Li et al., 2024), GCoder (Zhang et al., 2025), GraphTool-Instruction (Graph-TI) (Wang et al., 2025), GraphChain (Wei et al., 2025), and GraphCogent (Wang et al., 2026). For the main comparisons, all methods use Qwen3-8B (Qwen Team, 2025), with temperature 0.7, top-p 1.0, and an 8192-token output limit per call. Trainable baselines are retrained with the shared execution-tool catalog following their original training procedures; prompt-based baselines retain their original prompting and interaction protocols. For CAGR, MCTS uses an exploration coefficient of 1.4 and 128 completed evaluations per case. CCO runs for three iterations of 300 policy-update steps each. Local training uses eight NVIDIA A800 80GB GPUs; local inference uses eight NVIDIA RTX 5090 32GB GPUs on a server with 1TB RAM.

Evaluation Protocol. We evaluate task success rate and end-to-end execution time. Success rate measures the percentage of cases satisfying the task-specific criteria in Appendix D.4, while execution time includes LLM inference, graph compression, and task execution. We further report input and output token counts, LLM calls, and the breakdown of time spent on LLM inference and tool execution. For the main experiments, we report means and standard deviations over five runs. Appendix B provides method-specific configurations and timing procedures.

6.2 Main Results

Main Results. Tables 1 and 2 compare task success and execution costs using the same Qwen3-8B backbone over five runs. CAGR improves average success rate by 15.53 percentage points and achieves a 5.78× speedup over GraphChain. CCO jointly learns execution-tool selection and compression planning, while Certificate conditions ensure answer preservation for applicable compression steps. GraphChain remains slower than GCoder in three of the four domains. This comparison highlights that graph reduction alone does not ensure lower total execution cost. CAGR addresses both costs: the Toolchain Constructor generates a complete toolchain in one call, reducing LLM interaction, while the Compression Gate weighs further compression costs against subsequent execution savings.

Table 1: Main results on GraphLarge over five runs, reported as mean ± standard deviation. Success denotes task success rate (%); Time denotes execution time (s/case).

	Web		Transportation		Social		Citation	
Method	Success	Time	Success	Time	Success	Time	Success	Time
GraphTeam	76.33±11.01	92.43±5.35	82.67±8.63	54.00±2.47	78.33±4.97	65.50±2.55	77.73±6.18	66.77±2.88
GCoder	91.33±2.00	61.66±0.42	72.33±2.99	15.89±0.36	89.73±3.55	30.75±1.44	78.33±3.60	27.58±0.87
Graph-TI	82.73±4.71	65.08±1.01	79.27±3.55	25.52±0.54	83.67±4.84	37.02±2.25	73.40±2.10	36.95±0.49
GraphChain	74.73±3.12	56.75±4.54	75.40±9.09	26.48±0.77	79.53±9.13	43.42±4.73	76.87±3.47	45.60±0.35
GraphCogent	82.33±4.76	67.91±5.96	70.33±2.44	29.07±1.92	77.73±6.11	45.16±2.55	78.27±3.61	48.92±7.13
CAGR	**94.67**±1.13	**9.95**±0.87	**87.33**±4.08	**3.95**±0.07	**94.33**±2.12	**9.87**±1.76	**92.33**±2.07	**6.02**±0.40

Table 2: Efficiency decomposition on GraphLarge. Success, token counts, execution times, and LLM calls are averaged per case. Results are means and standard deviations over five runs.

Method	Success	Input Token	Output Token	Total Time	Tool Time	LLM Time	LLM Calls
GraphTeam	78.77 ± 5.92	6974.38 ± 71.42	992.63 ± 28.74	69.68 ± 2.88	30.42 ± 1.73	39.26 ± 2.35	3.47 ± 0.27
GCoder	82.93 ± 1.57	**2392.46 ± 0.00**	78.37 ± 11.26	33.97 ± 0.82	30.28 ± 0.78	3.69 ± 0.17	**1.00 ± 0.00**
Graph-TI	79.77 ± 1.39	5062.71 ± 17.24	189.47 ± 16.18	41.14 ± 0.71	33.46 ± 0.68	7.68 ± 0.05	3.00 ± 0.00
GraphChain	76.63 ± 4.01	6441.93 ± 52.37	437.68 ± 44.31	43.06 ± 2.47	27.43 ± 1.61	15.63 ± 1.50	5.79 ± 0.33
GraphCogent	77.17 ± 1.33	4486.57 ± 31.28	406.73 ± 42.19	47.77 ± 1.83	30.19 ± 0.96	17.58 ± 1.33	2.24 ± 0.11
CAGR	**92.17 ± 1.04**	3898.71 ± 0.00	**59.07 ± 4.66**	**7.45 ± 0.39**	**4.53 ± 0.29**	**2.92 ± 0.25**	**1.00 ± 0.00**

Compared with GCoder, the strongest baseline in task success, CAGR improves success by 9.24 percentage points and achieves a 4.56× speedup. GCoder also uses a single LLM call and fewer input tokens, yet its runtime remains dominated by tool execution. Graph-TI and GraphCogent show similar computational costs, while GraphTeam additionally incurs substantial LLM inference time. These results highlight that efficient tool invocation alone does not resolve the cost of graph computation. CAGR reduces this cost by constructing Certificate graphs for the selected execution tool, extending its efficiency gains beyond reduced LLM interaction.

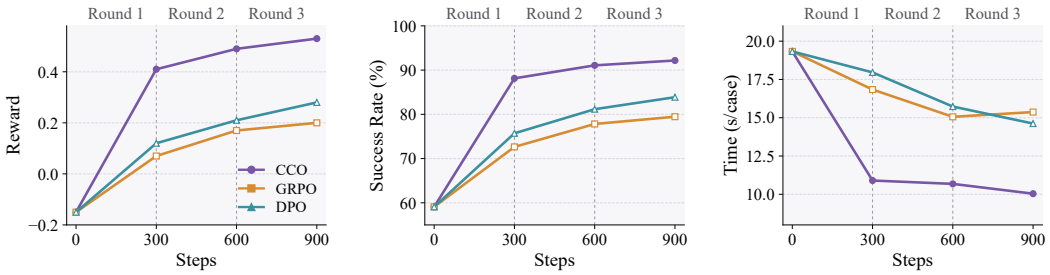


Figure 4: Training dynamics of CCO, GRPO, and DPO. Panels show mean reward, task success rate (%), and execution time (s) across policy updates.

Ablation Study. We compare CCO with GRPO (Shao et al., 2024) and DPO (Rafailov et al., 2023) for Constructor training under the same initialization, tool library, and toolchain evaluation budget. All three use the same toolchain reward: GRPO computes group-relative advantages, while DPO constructs preference pairs by ranking toolchains for each training instance. Figure 4 shows that all three improve, but CCO achieves larger gains in reward and task success together with lower execution time. GRPO's learning signal depends on toolchains sampled from the current policy, while DPO learns from the alternatives covered by its preference pairs. CCO uses policy-guided MCTS to explore alternative continuations and AWPI to reinforce decisions whose best observed completions exceed the prefix baseline. The Constructor thus learns both from high-reward toolchains discovered during search and from decision-level credit for the tool choices that support them.

Table 3 further examines CCO and the Compression Gate. Removing MCTS causes the largest drop in task success and final compression, while removing iterative policy improvement or local credit

Table 3: Component ablation on GraphLarge. γ and ω measure toolchain compression and order.

	GraphLarge			
Variant	γ	ω	Success	Time
w/o MCTS	0.64	0.46	81.93	14.85
w/o Iteration	0.70	0.72	88.13	10.90
w/o Local Credit	0.69	0.74	87.22	12.45
w/o Order ω	0.73	0.43	91.17	14.15
w/o Gate	0.73	0.81	92.17	10.04
CAGR	0.73	0.81	92.17	7.45

Table 4: Generalization experiments on unseen GraphLarge tasks, Graph4real, and GTools.

	GraphLarge		Graph4real		GTools	
Method	Success	Time	Success	Time	Success	Time
GraphTeam	70.23	54.16	83.05	33.80	82.00	29.27
GCoder	76.07	24.71	85.14	5.84	87.50	4.72
Graph-TI	72.73	28.09	86.10	26.56	**95.25**	21.40
GraphChain	68.10	26.38	76.00	28.93	79.25	23.47
GraphCogent	71.63	34.67	88.95	10.84	90.75	8.92
CAGR	**82.27**	**5.92**	**91.62**	**3.96**	93.25	**3.68**

also degrades both. Search explores tool compositions, and iterative updates allow the Constructor to incorporate search feedback; local credit reinforces individual decisions that support high-reward toolchains. Removing the order term ω leaves final compression nearly unchanged but almost doubles execution time. Final Certificate graph size alone therefore does not capture the computational benefit of a toolchain: earlier compression can reduce the graph processed by subsequent tools. Removing the Compression Gate leaves compression metrics and task success unchanged but increases execution time, separating the benefit of learning an effective toolchain from that of selecting when to execute.

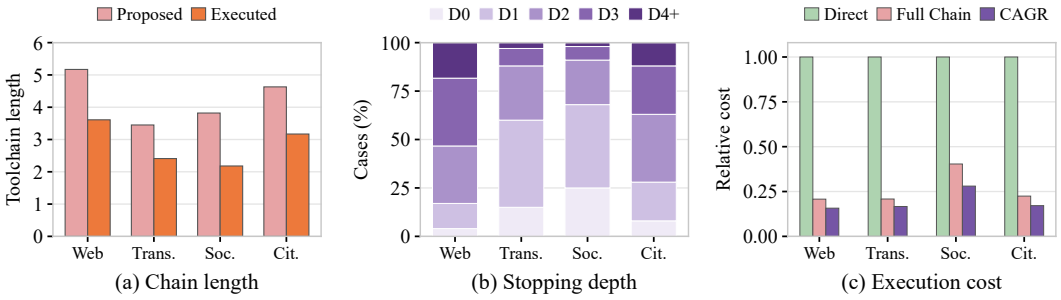


Figure 5: Toolchain construction and adaptive execution GraphLarge. (a) Proposed and executed toolchain lengths. (b) Gate exit-depth distributions, where D_d denotes stopping after d compression steps and D4+ groups all depths of four or more. (c) Execution time normalized by Direct execution.

Adaptive Execution and Generalization. Figure 5 shows how the Compression Gate adjusts the extent of Certificate graph construction within the proposed toolchain. Execution depth varies across domains. CAGR substantially reduces execution cost relative to Direct execution across all four domains and achieves lower average execution time than Full Chain. The Gate selects execution depth according to the expected benefit of further compression rather than always completing the proposed chain. Table 4 further shows that CAGR achieves the highest success rate and lowest execution time on unseen GraphLarge tasks and Graph4real, with the lowest execution time and competitive success on GTools. Results on six additional public benchmarks appear in Appendix B.2.

7 Conclusion

We study when and how to compress graphs for efficient LLM-based graph reasoning. Even for the same task, the graph information needed to preserve the answer and the cost of obtaining it vary across instances. We thus introduce Certificate graphs as executable representations that preserve task answers and develop CAGR to construct and execute them adaptively. Certificate Compression Optimization learns execution-tool selection and the composition of Certificate compression tools, while the Compression Gate weighs further compression costs against subsequent execution savings. We also introduce GraphLarge, covering 28 task settings across four domains at scales reaching one billion edges. Experiments on GraphLarge and public benchmarks demonstrate improved execution efficiency and competitive task success.

AI USE STATEMENT

Generative AI tools assisted with manuscript drafting and revision, including language clarity, paragraph structure, and terminology consistency. They also supported LaTeX formatting, figure and table layout, code editing and debugging, and review of preservation proofs for Certificate compression tools and implementation code, including model training code.

REFERENCES

Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The Transitive Reduction of a Directed Graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

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

Thomas Anthony, Zheng Tian, and David Barber. Thinking Fast and Slow with Deep Learning and Tree Search. In *NeurIPS*, pp. 5360–5370, 2017.

Vladimir Batagelj and Matjaz Zaversnik. An $O(m)$ Algorithm for Cores Decomposition of Networks. *CoRR*, cs.DS/0310049, 2003.

Guy Blanc, Caleb Koch, Jane Lange, and Li-Yang Tan. The Query Complexity of Certification. In *STOC*, pp. 623–636, 2022.

Harry Buhrman and Ronald de Wolf. Complexity Measures and Decision Tree Complexity: A Survey. *Theoretical Computer Science*, 288(1):21–43, 2002.

Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. GraphWiz: An Instruction-Following Language Model for Graph Computational Problems. In *KDD*, pp. 353–364, 2024.

Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. Query Preserving Graph Compression. In *SIGMOD*, pp. 157–168, 2012.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. Talk like a Graph: Encoding Graphs for Large Language Models. In *ICLR*, 2024.

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *WEA*, pp. 319–333, 2008.

Amy E. Hodler and Mark Needham. Graph Data Science Using Neo4j. In *Massive Graph Analytics*, pp. 433–457. Chapman and Hall/CRC, 2022.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*, 2022.

Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. In *NeurIPS Datasets and Benchmarks*, 2021.

Xin Li, Qizhi Chu, Yubin Chen, Yang Liu, Yaoqi Liu, Zekai Yu, Weize Chen, Chen Qian, Chuan Shi, and Cheng Yang. GraphTeam: Facilitating Large Language Model-based Graph Analysis via Multi-Agent Collaboration. *CoRR*, abs/2410.18032, 2024.

Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, and Xing Xie. GraphInstruct: Empowering Large Language Models with Graph Understanding and Reasoning Capability. *CoRR*, abs/2403.04483, 2024.

Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. The Graph Structure in the Web - Analyzed on Different Aggregation Levels. *The Journal of Web Science*, 1(1):33–47, 2015.

Hiroshi Nagamochi and Toshihide Ibaraki. A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph. *Algorithmica*, 7:583–596, 1992.

Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. Advantage-Weighted Regression: Simple and Scalable Off-Policy Reinforcement Learning. *CoRR*, abs/1910.00177, 2019.

Qwen Team. Qwen3 Technical Report. *CoRR*, abs/2505.09388, 2025.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D. Manning, Stefano Ermon, and Chelsea Finn. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. In *NeurIPS*, pp. 53728–53741, 2023.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*, 2023.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. *CoRR*, abs/2402.03300, 2024.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the Game of Go without Human Knowledge. *Nature*, 550(7676):354–359, 2017.

Jianheng Tang, Qifan Zhang, Yuhan Li, Nuo Chen, and Jia Li. GraphArena: Evaluating and Exploring Large Language Models on Graph Computation. In *ICLR*, 2025.

Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. Can Language Models Solve Graph Problems in Natural Language? In *NeurIPS*, 2023.

Rongzheng Wang, Shuang Liang, Qizhi Chen, Jiasheng Zhang, and Ke Qin. GraphTool-Instruction: Revolutionizing Graph Reasoning in LLMs through Decomposed Subtask Instruction. In *KDD*, pp. 1492–1503, 2025.

Rongzheng Wang, Shuang Liang, Qizhi Chen, Yihong Huang, Muquan Li, Yizhuo Ma, Dongyang Zhang, Ke Qin, and Man-Fai Leung. GraphCogent: Mitigating LLMs' Working Memory Constraints via Multi-Agent Collaboration in Complex Graph Understanding. In *WWW*, pp. 3811–3822, 2026.

Chunyu Wei, Wenji Hu, Xingjia Hao, Xin Wang, Yifan Yang, Yunhai Wang, Yang Tian, and Yueguo Chen. GraphChain: Large Language Models for Large-scale Graph Analysis via Tool Chaining. In *NeurIPS*, 2025.

Jaewon Yang and Jure Leskovec. Defining and Evaluating Network Communities Based on Ground-Truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*, 2023.

Zike Yuan, Ming Liu, Hui Wang, and Bing Qin. GraCoRe: Benchmarking Graph Comprehension and Complex Reasoning in Large Language Models. In *COLING*, pp. 7925–7948, 2025.

Qifan Zhang, Xiaobin Hong, Jianheng Tang, Nuo Chen, Yuhan Li, Wenzhong Li, Jing Tang, and Jia Li. GCoder: Improving Large Language Model for Generalized Graph Reasoning. In *CIKM*, pp. 4149–4159, 2025.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language Agent Tree Search Unifies Reasoning, Acting, and Planning in Language Models. In *ICML*, pp. 62138–62160, 2024.

A Additional Related Work

Tool Learning and Search. Tool-based reasoning requires deciding when to invoke tools and how to compose their outputs. ReAct addresses these decisions through interleaved reasoning and action (Yao et al., 2023), while Toolformer learns API invocation from self-supervised examples (Schick et al., 2023). Beyond sequential interaction, Language Agent Tree Search explores alternative reasoning and action trajectories using Monte Carlo Tree Search and environmental feedback (Zhou et al., 2024). Search outcomes can also guide policy improvement, as in Expert Iteration (Anthony et al., 2017), while Advantage-Weighted Regression provides an advantage-weighted learning objective (Peng et al., 2019). CCO combines toolchain search with advantage-weighted policy improvement to learn execution-tool selection and Certificate compression-tool composition. Because an effective compression chain need not be worth executing in full, the Compression Gate further determines when to stop compression and invoke the execution tool.

Graph Reasoning Benchmarks. Graph reasoning benchmarks have expanded from small-graph understanding to tool-assisted problem solving in real-world settings. NLGraph and GraphWiz evaluate structural understanding and algorithmic reasoning on small graphs (Wang et al., 2023; Chen et al., 2024), while GraphArena extends evaluation to computationally demanding combinatorial optimization problems (Tang et al., 2025). As tool use broadens the range of tractable tasks, GTools evaluates task identification, tool selection, and argument extraction across 20 tasks (Wang et al., 2025). Graph4real further grounds requests in four real-world domains, covering 21 tasks across structural querying, algorithmic reasoning, and predictive modeling, with accompanying scalability experiments reaching 10K nodes (Wang et al., 2026).

At larger graph scales, successful tool invocation should also be accompanied by efficient graph computation. GraphLarge evaluates both requirements through 28 task settings across four real-world domains, with graph instances reaching the billion-edge scale. Its tasks span graph traversal, path finding, network flow, matching, and structural counting, allowing evaluation of task success and execution cost under different computational requirements. This setting places greater emphasis on how systems organize graph computation, including whether answer-preserving compression yields sufficient execution savings to justify its construction cost.

Certificates and Construction Cost. Classical graph reductions identify information that can be removed while preserving specified properties. Transitive reduction preserves reachability (Aho et al., 1972), while sparse connectivity certificates preserve bounded connectivity using fewer edges (Nagamochi & Ibaraki, 1992). Certificate graphs provide a common executable representation for composing task-specific, answer-preserving reductions, with correctness established through each tool's applicability and preservation conditions.

Beyond answer preservation, constructing a reduced graph also incurs computational cost. Certificate Complexity characterizes the minimum number of input variables whose values suffice to determine the function value for a given instance (Buhrman & de Wolf, 2002), while work on certificate queries examines the cost of finding such information (Blanc et al., 2022). This distinction between sufficient information and the cost of obtaining it motivates CAGR to learn how Certificate compression tools should be composed and when further compression is justified by subsequent execution savings.

B Experimental Details

B.1 Implementation and Measurement Protocol

We use Qwen3-8B as the common backbone for CAGR and all trainable baselines (Qwen Team, 2025). CCO and its learning ablations start from the same Tool Warm Up checkpoint.

Constructor Training. Tool Warm Up and CCO policy updates use LoRA (Hu et al., 2022) on the attention and feed-forward projection layers, with rank 16, scaling parameter $\alpha = 32$, dropout 0.05, and bfloat16 precision. Both stages use two gradient-accumulation steps. Tool Warm Up runs for two epochs at a learning rate of 10^{-5} and a per-device batch size of 2. CCO then performs three iterations, each comprising toolchain search and 300 policy-update steps. Search uses 128 completed evaluations per case, with priors, search results, and decision weights recomputed in each iteration.

Table 5: Results on six smaller public graph-reasoning benchmarks, averaged over five runs. Success denotes exact-match success rate (%); Time denotes execution time (s/case).

	NLGraph		TLaG		GraphWiz		GraphArena		GraCoRe		GraphInstruct	
Method	Success	Time	Success	Time	Success	Time	Success	Time	Success	Time	Success	Time
GraphTeam	95.70	31.47	96.72	25.87	97.78	36.28	97.10	27.67	92.78	34.18	97.29	28.78
GCoder	98.10	5.21	97.72	4.38	**98.86**	6.44	97.60	4.65	97.22	5.93	**98.22**	4.87
Graph-TI	94.60	21.86	95.47	17.36	95.69	27.16	94.20	18.96	93.33	25.40	94.16	20.36
GraphChain	91.20	25.60	92.18	21.14	90.81	32.87	88.90	22.74	88.89	30.56	90.02	24.85
GraphCogent	96.70	9.74	**99.12**	8.41	98.36	12.20	96.40	8.76	95.56	11.64	97.14	9.21
CAGR	**99.20**	**3.82**	98.88	**3.54**	98.19	**4.21**	**99.30**	**3.61**	**99.44**	**4.09**	98.10	**3.73**

Within each MCTS search, repeated visits to the same toolchain prefix reuse its graph state and evaluation records; compression results are not shared across cases. Policy updates use a per-device batch size of 1 and learning rate of 5×10^{-6} across successive iterations. Search and generation allow at most 10 compression tools per chain, with STOP determining the actual length.

Gate Training. After selecting and freezing the Constructor checkpoint, we train the Gate, a two-hidden-layer MLP with widths of 64 and 32 and ReLU activations. Training uses AdamW with a learning rate of 10^{-3}, a batch size of 128, and Huber loss with transition parameter 1.

Baseline Adaptation. We retain each baseline's original reasoning and interaction workflow while adapting its task-execution interfaces with our shared execution-tool library. Trainable baselines are retrained with the shared execution-tool catalog following their original training procedures. All methods access the same execution-tool implementations and descriptions, which specify supported tasks, required arguments, and applicability conditions. Multiple tools may support the graph-problem family under different graph conditions, leaving tool selection and argument specification to each method. Baselines retain their original information-reduction mechanisms, including GraphChain's dynamically generated reduction code; CAGR's Certificate compression tools are not added to their tool interfaces.

Execution and Measurement. Graph computation runs on two Intel Xeon Gold 6338 CPUs, with 64 physical cores and 128 hardware threads in total. Both training-time MCTS and evaluation use four-way concurrency. All graphs are loaded into memory in a unified representation before evaluation. Initial graph loading is excluded from timing. Each case is executed independently without reusing query-dependent intermediate graphs, evaluation records, or answers from training, other cases, or previous runs. All methods share the same backend and execution-tool implementations. Execution time is measured from task submission to answer recovery, including LLM inference, graph compression, Gate evaluation, and execution-tool computation. Success rate and execution time are macro-averaged across tasks, and we report means and standard deviations over five runs with different random seeds.

B.2 Additional Results

Table 5 extends the evaluation to six smaller public benchmarks, where CAGR retains the lowest execution time and competitive success rates. Here, the Constructor selects execution tools directly without compression, showing that effective tool selection is retained when Certificate graph construction requires no reduction. We further compare CAGR with API-based LLMs, locally deployed LLMs, and task-trained baselines using the same compression and execution tools. All methods receive the same requests, graph summaries, and tool descriptions, and share the graph backend and evaluation criteria. Each method generates a complete toolchain in a single LLM call. GCoder and Graph-TI use Qwen3-8B and are adapted through supervised fine-tuning over the full tool library. CAGR uses a Constructor trained through CCO with either Qwen3-8B or Qwen3.5-9B and its corresponding Compression Gate.

Table 6 shows that CAGR improves success by 33.08 percentage points and achieves a 2.60× speedup over the Qwen3-8B function-calling baseline using the same backbone. Access to the full tool library reduces execution time for GCoder and Graph-TI, but their success rates fall below those of their execution-only counterparts in Table 1. The additional compression tools therefore

Table 6: Comparison with API-based LLMs, local LLMs, and task-trained baselines on GraphLarge using the full tool library. CAGR (Qwen3-8B) results are the five-run means from Table 1.

	Web		Transportation		Social		Citation		Avg.	
Method	Success	Time	Success	Time	Success	Time	Success	Time	Success	Time
API-based LLMs										
DeepSeek-V4-Flash	93.67	10.82	95.67	9.63	92.67	14.58	90.33	16.31	93.08	12.84
GPT-5.4-mini	98.33	26.45	97.67	12.74	96.33	18.96	94.67	20.43	96.75	19.65
Gemini-3.8-Flash	92.67	13.67	95.33	11.86	91.67	17.34	89.33	12.95	92.25	13.96
Local-based LLMs										
Qwen3-8B	59.67	24.83	62.00	10.37	56.00	23.69	58.67	18.42	59.08	19.33
Llama3.1-8B	50.67	17.46	52.00	15.28	46.00	26.74	49.67	16.93	49.58	19.10
Qwen3.5-9B	61.00	20.18	64.33	9.84	56.67	18.92	60.33	22.36	60.58	17.83
Qwen3.5-27B	72.00	14.72	70.67	12.63	69.33	15.86	69.67	13.47	70.42	14.17
Task-Trained Baselines										
GCoder	87.67	10.84	69.33	8.96	84.67	14.38	74.33	12.65	79.00	11.71
Graph-TI	79.33	12.36	74.33	10.24	78.67	13.07	69.33	15.92	75.42	12.90
Ours										
CAGR (Qwen3-8B)	94.67	9.95	87.33	3.95	94.33	9.87	92.33	6.02	92.17	7.45
CAGR (Qwen3.5-9B)	95.33	8.24	89.00	3.39	93.67	7.50	93.33	4.51	92.83	5.91

offer computational savings while introducing tool-selection and composition decisions that these baselines do not consistently resolve. CAGR learns these decisions jointly through CCO, while the Compression Gate determines how much Certificate graph construction is worth executing. GPT-5.4-mini achieves the highest success rate among API-based LLMs, whereas CAGR achieves the lowest execution time with either backbone. With Qwen3.5-9B, CAGR reaches 92.83% success at 5.91 seconds per case, retaining its efficiency advantage with a second backbone.

B.3 Observation Experiment Settings

Table 7 lists the compression operations used in Fig. 2. Shortest Path and Max Flow follow the same operation sequence with task-specific contraction and parallel-edge rules. Within each task, the sequence is fixed across all 100 instances. We select the five instances with the smallest and largest Certificate graphs after Depth 5 and retain these groups at every depth. Curves report group means. At depth d, total time includes the first d compression steps and task execution on the resulting graph, relative to Direct execution on the same instance. Depth 0 denotes Direct execution.

Depth 2 retains vertices reachable from the source, and Depth 3 further restricts them to vertices that can reach the target; query terminals are retained. Depth 4 contracts nonterminal vertices with exactly one incoming and one outgoing edge and distinct predecessor and successor vertices. Shortcuts sum nonnegative path weights for Shortest Path or take the minimum of nonnegative capacities for Max Flow. Existing edges are retained alongside shortcuts until Depth 5, which selects the minimum-weight edge for Shortest Path or sums capacities for Max Flow.

Table 7: Ordered compression tools for the observation experiment.

Depth	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5
Operation	Self-Loop Removal	Source-Descendant Filtering	Target-Ancestor Filtering	Degree-2 Vertex Contraction	Parallel-Edge Reduction

C CAGR Method Details

C.1 Graph Representation and Tool Composition

We use a shared CSR-backed graph representation across methods. Adjacency offsets and neighbor identifiers are stored in CSR arrays, with task-relevant properties and schema metadata maintained alongside them. The logical representation follows the Neo4j GDS projection model (Hodler & Needham, 2022). Compression and execution tools operate on this representation through a unified

interface, using CSR implementations or adapters. Given the task arguments, a compression tool returns an executable graph representation with recovery records, while an execution tool returns the task result. All methods share the same backend and execution-tool implementations.

Within its supported domain, each compression tool must preserve both the Certificate condition and compatibility with the selected execution tool:

$$\begin{aligned} &\mathrm{Cert}_q(H; G) \wedge H \in \mathrm{Dom}(E) \\ &\quad \Longrightarrow \mathrm{Cert}_q(C(H); G) \wedge C(H) \in \mathrm{Dom}(E). \end{aligned} \tag{10}$$

Here, $\mathrm{Cert}_q(H; G)$ states that H preserves the answer to task q on the original graph G, and $\mathrm{Dom}(E)$ specifies the representations accepted by the selected execution tool. Answer equivalence includes any recovery required by changes in representation, as defined below. Within its supported domain, a tool applies its transformation when the structural conditions hold and otherwise returns the graph unchanged. Unsupported tasks and incompatible representations are invalid.

Answer Recovery. When compression changes the graph representation, the execution tool may return an answer expressed through compressed nodes or edges. A recovery procedure maps this output to an equivalent answer on the original graph. For a compression tool C, we require

$$\mathrm{Recover}_C(A_q(C(H))) \equiv_q A_q(H). \tag{11}$$

Here, recovery uses the records retained by the compression tool, and $\equiv_q$ denotes equivalence under the task's output requirements. For example, recovery expands a shortcut into its original path or maps compressed node identifiers back to the original graph. Outputs that remain valid without conversion use identity recovery. For optimization tasks, recovery restores a feasible optimal solution to the original problem, including any objective offsets introduced by compression.

For a sequence $H_i = C_i(H_{i-1})$, the recovery procedures are composed in reverse execution order:

$$\mathrm{Recover}_{1:i} = \mathrm{Recover}_{C_1} \circ \cdots \circ \mathrm{Recover}_{C_i}, \qquad \mathrm{Recover}_{1:i}(A_q(H_i)) \equiv_q A_q(G). \tag{12}$$

The procedure recovers the task answer without reconstructing the entire original graph. Recovery time is included in execution-tool time.

Safe Prefix Execution. Let $H_0 = G$ be accepted by the selected execution tool E, and let $H_i = C_i(H_{i-1})$. If each compression tool operates within its supported domain and satisfies Eq. 10, every prefix preserves the Certificate condition and remains executable:

$$\mathrm{Cert}_q(H_i; G), \qquad H_i \in \mathrm{Dom}(E), \qquad i = 0, \ldots, K. \tag{13}$$

This follows by induction: the original graph satisfies both conditions, and each compression step preserves them. The selected execution tool can therefore answer the original task before compression or after any compression step. Appendix E organizes the tool-specific preservation conditions and recovery rules by proof mechanism.

C.2 LLM CONSTRUCTOR

Constructor Input. The Constructor input x comprises the task description, graph information, and tool library. Task information describes the requested computation, including the relevant entities and constraints. Graph information consists of the graph's storage path and structural summary. The path identifies the graph accessed by the tools. We compute the graph summary from the CSR representation and its associated schema metadata, as summarized in Table 8. Tool information describes the available operations and their arguments.

Toolchain Output. Given x, the Constructor selects an execution tool E and organizes compression tools into an ordered chain:

$$\pi = (C_1, \ldots, C_K, E). \tag{14}$$

Here, C_i denotes the i-th compression tool and K is the number of compression steps. The Constructor also supplies the task arguments required by each selected tool. During execution, the tools access the graph through its storage path and operate on successive graph representations. The execution tool computes the answer at the prefix selected by the Compression Gate.

Table 8: Graph metadata and structural statistics supplied to the Constructor.

Category	Fields	Description
Scale	Node and edge counts	Graph size
Storage	Size in bytes	Packed graph storage size
Density	Edge density	Overall connection density
Degree	Min, max, mean; P50, P90, P99	Degree distribution
Orientation	Directed or undirected	Relationship direction
Types	Node labels; edge types	Entity and relation categories
Properties	Property names and data types	Available graph attributes

C.3 Certificate Compression Optimization

Certificate Compression Optimization (CCO) alternates toolchain search and Constructor updates. At iteration r, the current policy θ_r guides search for each training task and graph. Executed candidate chains provide rewards, and the resulting decision values determine which tool choices are reinforced.

Tool Warm Up. Before CCO, we perform Tool Warm Up through SFT on tool documentation and invocation examples to improve toolchain sampling quality. This stage familiarizes the Constructor with tool functions, arguments, and applicability conditions to reduce formatting and invocation errors during sampling. The training data teach correct tool invocation and do not include complete toolchain generation examples. Tools exclusive to OOD tasks are excluded from Tool Warm Up and CCO training, including their documentation, invocation examples, and execution feedback. Their descriptions are provided to the Constructor only at evaluation.

Search Space and Action Prior. Given the input x, the Constructor generates a toolchain by first selecting an execution tool and then composing an ordered sequence of compression tools. CCO uses MCTS to explore alternative choices throughout this generation process. Each search state s represents a partial toolchain. Conditioned on x and s, the Constructor proposes an execution tool at the root, or a compression tool or STOP at subsequent states. Choosing when to stop allows the Constructor to determine the chain length, subject to a shared maximum number of compression tools.

At each toolchain prefix, the Constructor proposes the next action by sampling from its current policy. As MCTS revisits a prefix, additional proposals update the candidate set and action counts. We compute the empirical action prior from these proposal frequencies:

$$\widehat{P}_r(a \mid x, s) = \frac{n_r(s,a)}{\sum_{b \in \mathcal{A}_{\text{prop}}(s)} n_r(s,b)}, \tag{15}$$

where $\mathcal{A}_{\text{prop}}(s)$ is the set of distinct actions proposed at prefix s, and $n_r(s,a)$ counts how often the Constructor proposes action a. MCTS uses this prior together with the returns from evaluated toolchains to select which candidate action to explore.

Toolchain Evaluation. CCO evaluates a candidate chain by its execution validity, final compression, and compression order. We execute the chain on the corresponding training graph. Execution validity ν is 1 if the chain satisfies the task, Certificate, and representation contracts, does not increase graph size at any step, and produces a correct task answer; otherwise, $\nu = 0$.

We measure graph size $U(H)$ by the total number of nodes and edges in H. The cumulative compression after step i is $c_i = 1 - U(H_i)/U(G)$, with $c_0 = 0$, and the final compression is $\gamma = c_K$. CCO considers chains whose graph size does not increase, so each increment $c_i - c_{i-1}$ is nonnegative. To reward earlier compression gains, we weight each increment by the inverse of its position and define:

$$\omega = \begin{cases} \dfrac{1}{\gamma} \displaystyle\sum_{i=1}^{K} \frac{c_i - c_{i-1}}{i}, & \gamma > 0, \\ 0, & \gamma = 0, \end{cases} \qquad R(\pi) = \nu \left[1 + \frac{\gamma(1+\omega)}{2} \right] - 1. \tag{16}$$

Here, γ rewards the total compression achieved, while ω rewards obtaining that compression earlier in the chain. The increments sum to γ, giving $\omega \in [0,1]$ and $R \in [-1,1]$. For invalid chains, we set

$\gamma = \omega = 0$, yielding $R = -1$. Valid chains without compression receive 0, and valid compressing chains receive a positive reward. An empty compression chain has $\gamma = \omega = 0$. A valid step that leaves the graph unchanged contributes no compression gain; appending such a step does not increase the reward.

Search and Decision Values. CCO uses the rewards of completed toolchains to evaluate the decisions along their prefixes. During search, each decision is assigned the average reward of its evaluated continuations:

$$Q_{\text{search}}(s, a) = \frac{1}{N(s, a)} \sum_{j=1}^{N(s,a)} R(\pi_j), \tag{17}$$

where π_j is a completed toolchain evaluated through action a at prefix s, and $N(s, a)$ counts these evaluations. The mean is defined for $N(s, a) > 0$; unvisited edges are initialized with $Q_{\text{search}}(s, a) = 0$. MCTS combines this value with the action prior and visit counts in Eq. 5 to select the next branch to explore.

For policy improvement, we seek decisions that can support high-reward toolchains. Their average returns can be low when subsequent tools are poorly composed, even if search has found an effective continuation. We therefore score each decision by its best observed return:

$$B_r(s, a) = \max_{\pi \in \mathcal{D}_r(s,a)} R(\pi), \tag{18}$$

where $\mathcal{D}_r(s, a)$ contains the toolchains evaluated through (s, a) in iteration r. This score supplies the decision values used to compute the policy-update advantages.

Policy Improvement. We reinforce decisions whose best observed returns exceed the baseline at the same prefix. Let $\mathcal{A}_{\text{obs}}(s)$ contain actions with evaluated continuations. We normalize their proposal probabilities and use the resulting distribution to compute the baseline and positive advantage:

$$\begin{aligned} \widetilde{P}_r(a \mid x, s) &= \frac{\widehat{P}_r(a \mid x, s)}{\sum_{b \in \mathcal{A}_{\text{obs}}(s)} \widehat{P}_r(b \mid x, s)}, \\ V_{B,r}(s) &= \sum_{a \in \mathcal{A}_{\text{obs}}(s)} \widetilde{P}_r(a \mid x, s) B_r(s, a), \\ A_r^+(s, a) &= \max\{B_r(s, a) - V_{B,r}(s), 0\}. \end{aligned} \tag{19}$$

Here, $V_{B,r}(s)$ is the prior-weighted average of the observed decision scores, using proposal counts at the end of search. Decisions above this baseline receive positive training weights.

For each instance, we select the highest-reward valid toolchain found during search, preferring the shorter chain when rewards are equal. Its positive-advantage decisions are collected into the iteration-wide training set $\mathcal{D}_r^+$. We normalize their advantages to have unit mean:

$$\overline{A}_r^+(s, a) = \frac{A_r^+(s, a)}{\frac{1}{|\mathcal{D}_r^+|} \sum_{(x', s', a') \in \mathcal{D}_r^+} A_r^+(s', a')}. \tag{20}$$

These normalized advantages determine the weights used by AWPI in Eq. 8. Each weight is assigned to the tokens expressing the corresponding tool choice or STOP decision; fixed output syntax receives zero weight. The minibatch loss is normalized by the sum of token weights. Instances without positive-advantage decisions contribute no training term, and an empty training set skips the update. The updated Constructor then guides the next round of toolchain search.

C.4 ADAPTIVE EXECUTION WITH A COMPRESSION GATE

The Constructor generates a complete toolchain, but executing the full compression chain does not always minimize execution time. Since Safe Prefix Execution allows the selected execution tool to operate on the Certificate graph at any prefix, the Compression Gate can stop compression before the chain ends. It determines when to execute by weighing further compression costs against subsequent execution savings.

Gate Observation. At depth d, the Gate combines the original graph summary $\mathbf{s}(G)$, query-node statistics $\mathbf{a}(q, G)$, and observed compression state $\mathbf{m}_d$ with the remaining-plan complexity $\boldsymbol{\kappa}_d$. The

two input blocks in Eq. 9 are

$$\mathbf{g}_d = \mathbf{s}(G) \oplus \mathbf{a}(q, G) \oplus \mathbf{m}_d, \qquad \mathbf{p}_d = \boldsymbol{\kappa}_d(\pi, H_d). \tag{21}$$

Table 9: Input features of the Compression Gate.

Category	Fields	Source
Graph summary	Numeric statistics and directionality	Original graph
Compression state	Current node and edge counts	Current graph
	Previous node and edge compression fractions	Tool execution
	Last and cumulative record-access fractions	Tool execution
	Number of completed compression steps	Execution progress
	Number of remaining compression steps	Execution progress
Query-node statistics	In/out-degrees; Presence masks	Query
Plan complexity	Reducer/execution tool values; Presence masks	LLM Constructor

Table 9 summarizes the observation fields. The same observation schema is used throughout execution. Before compression begins, current graph counts equal the original counts, compression-history fields are zero, and the remaining step count equals the generated chain length. Each compression step updates the observed state and execution progress, and recomputes the remaining-plan complexity values at the new graph size, while the original graph summary stays unchanged. Compression fractions are relative to the previous step's input. Record-access fractions are normalized by the original node and adjacency records, with repeated accesses counted repeatedly.

Query-node statistics describe the structural context of nodes specified in the request. Statistics are retrieved from the original graph and arranged in fixed slots according to argument roles, with zero padding and presence masks for missing arguments. They remain unchanged throughout compression, providing a consistent query-specific context for Gate decisions.

Plan complexity describes the computational requirements of the remaining compression tools and the selected execution tool using the complexity expressions recorded in their tool cards. At prefix d, each remaining tool T is encoded as

$$\kappa_d(T) = \log(1 + f_T(n_d, m_d, \boldsymbol{\xi}_q)), \tag{22}$$

where f_T describes the complexity of the implemented algorithm, $n_d = |V(H_d)|$, $m_d = |E(H_d)|$, and $\boldsymbol{\xi}_q$ contains the required query parameters. The vector $\boldsymbol{\kappa}_d$ arranges the remaining compression-tool values in execution order, followed by a fixed execution-tool slot, with zero padding and presence masks for variable-length chains. All expressions are evaluated at the current graph size and updated after each compression step, allowing the Gate to relate the remaining computation to the observed graph reduction.

Training Data and Supervision. After CCO, we freeze the Constructor and generate complete toolchains for training instances. Each valid compression chain is executed to completion, recording the graph state and compression cost at every step. We also run the selected execution tool separately on the original graph and every intermediate graph to measure the cost of answering the task at each prefix. These execution-tool evaluations leave the compression trajectory unchanged. Together, the measurements determine whether continuing from a prefix can reduce the total completion cost.

For each prefix at depth $d < K$, the observed features $\mathbf{z}_d$ form the Gate input. We construct its supervision target y_d from the measured cost of immediate execution, S_d, and the best subsequent completion cost, F_d:

$$\begin{aligned} S_d &= T_{\text{exec}}(E, H_d), \\ F_d &= \min_{d<j\le K} \left[\sum_{i=d+1}^{j} T_{\text{comp}}(C_i, H_{i-1}) + \sum_{\ell=d+1}^{j} T_{\text{gate},\ell} + T_{\text{exec}}(E, H_j) \right], \\ y_d &= \log \frac{S_d + \epsilon}{F_d + \epsilon}. \end{aligned} \tag{23}$$

Here, F_d includes the additional compression, Gate evaluation, and execution-tool costs of finishing at a later prefix of the generated chain. A positive target favors further compression, while a

non-positive target favors immediate execution. Earlier compression costs and the current Gate call are shared by both choices and cancel. Future Gate costs include feature processing and prediction, with $T_{\mathrm{gate},K} = 0$ because the execution tool is invoked directly at the chain end. We set $\epsilon = 10^{-8}$ s to ensure numerical stability. Future-prefix measurements provide the training targets, while observations contain only information available at the current prefix.

Gate Training. We implement the Gate as a compact MLP with two ReLU hidden layers and a scalar output. Nonnegative count features are transformed by $\log(1+x)$, and numerical features are standardized using training-prefix statistics. Complexity values are already log-transformed. The model predicts the relative continuation benefit and is trained with:

$$\mathcal{L}_{\mathrm{Gate}}(\phi) = \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \frac{1}{K_i} \sum_{d=0}^{K_i - 1} \rho(\mathrm{Gate}_\phi(\mathbf{z}_{i,d}) - y_{i,d}) \,, \tag{24}$$

where $\mathcal{I}$ contains training instances with nonempty valid chains, K_i is the compression-chain length, $\mathbf{z}_{i,d}$ is the observation at prefix d, and ρ is the Huber loss. Averaging within each instance gives equal total weight to chains of different lengths. Prefixes from the same instance remain in the same data partition. The validation split determines the checkpoint, and Appendix B provides the architecture settings and measurement protocol.

Adaptive Execution. At inference, the Gate evaluates the generated chain starting from the original graph. A positive prediction executes the next compression tool, after which the Gate reassesses the updated state. A non-positive prediction invokes the selected execution tool on the current graph. Empty chains and completed chains invoke the execution tool directly. The resulting execution depth adapts to the observed compression trajectory.

D GraphLarge Construction Details

D.1 Graph Sources and Task Inventory

GraphLarge pairs real-world graph topologies with domain-grounded algorithmic tasks. Web cases are derived from the WDC Hyperlink Graph (Meusel et al., 2015); Transportation uses Overture Maps Transportation[1]; Social uses Friendster (Yang & Leskovec, 2015); and Citation uses MAG240M (Hu et al., 2021). Requests express graph problems through domain-specific scenarios and parameters. Available source attributes are retained, and missing task-specific costs and capacities are generated as described in Appendix D.4. Table 10 reports the scale of the primary graph source in each domain, and Tables 11–14 list the complete task inventory. The five ID tasks in each domain are visible during training, whereas the two OOD tasks remain sealed until evaluation. The reported task complexities give the worst-case running time of the corresponding exact algorithms. Here, $n = |V|$ and $m = |E|$.

Table 10: Scale of the original graph sources used by GraphLarge.

Domain	Original graph	Edge semantics	Nodes	Edges
Web	WDC Hyperlink Graph (2014)	Hyperlinks	1.7B	64.0B
Transportation	Overture Maps Transportation	Road segments	419.5M	350.5M
Social	Friendster	Friendship links	65.6M	1.8B
Citation	MAG240M	Scholarly relations	244.2M	1.7B

D.2 Source Partitioning and Subgraph Sampling

Source Partitioning. Before sampling, we partition each source graph into three node-disjoint regions for training, validation, and test, with approximately balanced internal edge counts. Starting from three uniformly sampled distinct nodes, we expand regions along the original adjacency structure, prioritizing the region with the fewest internal edges among those with unassigned neighbors. Node assignments remain fixed. When no region can expand, we uniformly sample an unassigned node to resume growth in the region with the fewest internal edges. Each region retains all original

[1] https://docs.overturemaps.org/guides/transportation/

Table 11: Details of Graph Reasoning Tasks in the Web Domain.

Task	Description	Task Split	Time Complexity
Shortest Path	Find the minimum-cost hyperlink path between two pages.	ID	$\mathcal{O}((n+m)\log n)$
Reachability	Determine whether one page can reach another through directed hyperlinks.	ID	$\mathcal{O}(n+m)$
Directed Cycle Detection	Determine whether the hyperlink graph contains a directed cycle.	ID	$\mathcal{O}(n+m)$
Maximum Flow	Compute the maximum flow between two pages under hyperlink capacities.	ID	$\mathcal{O}(n^2m)$
Maximum Bipartite Matching	Find a maximum matching between two disjoint node sets using existing edges as feasible pairs.	ID	$\mathcal{O}(m\sqrt{n})$
Widest Path	Find the path whose minimum edge capacity is maximized.	OOD	$\mathcal{O}((n+m)\log n)$
Minimum Cut	Find the minimum-capacity cut separating two pages.	OOD	$\mathcal{O}(n^2m)$

Table 12: Details of Graph Reasoning Tasks in the Transportation Domain.

Task	Description	Task Split	Time Complexity
Shortest Path	Find the minimum-travel-time route between two locations.	ID	$\mathcal{O}((n+m)\log n)$
Minimum Spanning Forest	Find a minimum-cost spanning forest of the undirected road network.	ID	$\mathcal{O}(n+m\log m)$
Minimum-Cost Flow	Minimize transportation cost while meeting specified supplies, demands, and road capacities.	ID	$\mathcal{O}(m\log n\cdot (m+n\log n))$
Maximum Flow	Compute the maximum evacuation flow under road capacities.	ID	$\mathcal{O}(n^2m)$
Maximum Bipartite Matching	Find a maximum matching between two disjoint node sets using existing edges as feasible pairs.	ID	$\mathcal{O}(m\sqrt{n})$
Replacement Paths	Find an alternative shortest route for every edge failure on the original route.	OOD	$\mathcal{O}(n(n+m)\log n)$
Edge-Disjoint Paths	Find the maximum number of edge-disjoint routes.	OOD	$\mathcal{O}(n^2m)$

Table 13: Details of Graph Reasoning Tasks in the Social Domain.

Task	Description	Task Split	Time Complexity
Shortest Path	Find the minimum-hop connection path between two users.	ID	$\mathcal{O}(n+m)$
Maximum Matching	Find the largest set of social links with no shared users.	ID	$\mathcal{O}(n^3)$
Triangle Counting	Count triangles in the undirected social graph, where Δ is the maximum degree.	ID	$\mathcal{O}(n+m\Delta)$
Maximum Flow	Compute the maximum flow between two users under social-link capacities.	ID	$\mathcal{O}(n^2m)$
Maximum Bipartite Matching	Find a maximum matching between two disjoint node sets using existing edges as feasible pairs.	ID	$\mathcal{O}(m\sqrt{n})$
Minimum Cut	Find the minimum-capacity cut separating two users in the social-link graph.	OOD	$\mathcal{O}(n^2m)$
Widest Path	Find a path between two users that maximizes the minimum edge capacity along the path.	OOD	$\mathcal{O}((n+m)\log n)$

edges between its nodes; cross-region edges are excluded from sampling and counted separately. Partitioning depends only on the source topology.

Table 14: Details of Graph Reasoning Tasks in the Citation Domain.

Task	Description	Task Split	Time Complexity
Node-Disjoint Paths	Find f, the maximum number of citation paths sharing only their source and target papers.	ID	$\mathcal{O}((f+1)(n+m))$
Shortest Path	Find the minimum-hop citation chain between two papers.	ID	$\mathcal{O}(n+m)$
Triangle Counting	Count citation patterns where a paper cites another paper and one of that paper's references.	ID	$\mathcal{O}(n+m+\sum_v d_{\text{in}}(v)d_{\text{out}}(v))$
Maximum Flow	Compute the maximum flow between two papers under citation-link capacities.	ID	$\mathcal{O}(n^2m)$
Maximum Bipartite Matching	Find a maximum matching between two disjoint node sets using existing edges as feasible pairs.	ID	$\mathcal{O}(m\sqrt{n})$
Widest Path	Find a citation path between two papers that maximizes the minimum edge capacity along the path.	OOD	$\mathcal{O}((n+m)\log n)$
Minimum Cut	Find the minimum-capacity cut separating two papers in the citation graph.	OOD	$\mathcal{O}(n^2m)$

Subgraph Sampling. For each graph, we initialize 16 random walks from uniformly sampled distinct nodes within its assigned region and advance them in round-robin order. Each step selects a neighbor uniformly and adds the visited node to a shared node set. A walk restarts from a uniformly sampled node in the same region when it has no available neighbor or completes 128 consecutive steps without adding a new node; previously sampled nodes are retained. For directed graphs, partitioning and sampling ignore edge direction, while the output preserves original directions.

After each round, we check the number of original edges induced by the selected nodes and stop when it first reaches the target. We then read their complete adjacency to retain all original edges between selected nodes, including edges not traversed by the walks. Any excess above the target is retained rather than removed. Target edge counts are fixed per domain and checked against the internal edge counts available in all three splits. The walk count, restart threshold, and target scale are shared across splits within each domain.

Sampled graphs retain available task-relevant source attributes; missing attributes follow Appendix D.4. Each task uses 30 independently sampled graphs: 18/6/6 for ID training/validation/test, or 30 test graphs for OOD evaluation. Graphs may overlap within a split but never across splits.

Domains and Scales. Table 15 reports mean node and edge counts over sampled graphs in each domain. Within each domain, training, validation, and test instances follow the same sampling protocol and target scale.

Table 15: Mean node and edge counts of GraphLarge instances across the four domains.

Web		Transportation		Social		Citation	
Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
30.5M	1.15B	50.9M	56.0M	13.2M	362.95M	40.2M	279.65M

D.3 Requests and Data Splits

Following the intent-driven task design of GraphCogent (Wang et al., 2026), we construct natural-language requests around domain scenarios and user goals without explicitly naming the underlying algorithm. For each task, we design 10 templates that vary the scenario descriptions, wording, and parameter placement. After each sampled graph is fixed, we instantiate all 10 templates with different query nodes and task parameters where applicable, producing 10 requests. Query nodes are sampled uniformly from vertices satisfying the task's type and argument requirements; valid unreachable queries and other negative cases are retained.

Each in-distribution task contains 300 cases, comprising 180 training, 60 validation, and 60 test cases. Each out-of-distribution task contains 300 test cases, with its definition, cases, and execution feedback excluded from training and model selection. All 10 requests generated from a sampled graph belong to the same training, validation, or test split. For OOD evaluation, the Constructor receives the held-out task definition and generates a toolchain from the existing tool library without additional training.

D.4 Reference Answers and Evaluation

Sampled graphs use existing source edges and retain available task-relevant attributes. Missing algorithmic edge attributes, including costs and capacities, are sampled uniformly from $\{1, \ldots, 50\}$. Graph attributes are fixed before method execution and do not depend on the generated toolchain. For bipartite matching, feasible pairs are existing edges between two disjoint vertex sets. All methods and the reference solver use the same graph attributes and query parameters.

For all GraphLarge tasks, we compute reference answers on each sampled graph before compression using the corresponding standard algorithms. Boolean and integer outputs are matched exactly, while floating-point outputs are compared within a predefined numerical tolerance. For structured outputs, such as paths and matchings, we verify their validity and compare their objective values with the reference optimum, accepting equivalent optimal solutions.

We report the fraction of outputs satisfying these task-specific evaluation criteria as the success rate.

E Certificate-Aware Tool Library

E.1 Tool Library Construction and Statistics

CAGR provides a unified library of execution tools and Certificate compression tools for toolchain construction. As summarized in Table 16, the library contains 45 execution tools covering the 28 GraphLarge task settings and 77 Certificate compression tools: 33 task-general and 44 task-specific tools. Each callable entry is described by a tool card specifying its supported tasks, inputs, and outputs. The cards also record the complexity expression of each tool's implemented algorithm and the parameters needed to evaluate it, allowing the Gate to characterize the computational scale of the remaining plan. The catalog is fixed before evaluation.

Table 16: Composition of the Certificate-Aware Tool Library.

Tool Type	Number	Scope	Function
Execution tools	45	28 task settings	Execute the requested task on the selected graph.
Task-general compression tools	33	Reusable structural rules	Apply reusable graph compression operations under their corresponding Certificate conditions.
Task-specific compression tools	44	Individual problem contracts	Exploit the answer structure and graph properties of a specific problem.
Total	122	Four domains and 28 task settings	Construct and execute Certificate-aware toolchains.

We build the library from 614 documented APIs in NetworkX 3.6.1 and 82 Neo4j GDS algorithm or graph-operation families, combining graph primitives with task-specific transformations and recovery rules. We select exact transformations that reduce or reorganize graph representations while preserving task answers under their applicability conditions.

Each selected transformation is paired with a task-preservation contract specifying its supported graph problems, structural preconditions, retained evidence, and answer recovery. The contract turns a general graph operation into a Certificate compression tool. Different task semantics can yield distinct tools from the same operation: parallel-edge aggregation retains the minimum weight for

Shortest Path, the maximum capacity for Widest Path, and the summed capacity for Maximum Flow. Each tool preserves the task answer under its stated conditions.

At evaluation, the Constructor receives the same complete tool library for every task and selects tools according to the request and graph. The following subsections present preservation proofs organized by compression mechanism. A proof applies to multiple tools when they share the same structural preconditions, transformation, aggregation rule, retained evidence, and recovery procedure.

E.2 Certificate Compression Tool Contract

Each Certificate compression tool is supported by a preservation argument following the contract in Figure 6. Its tool card summarizes the supported tasks, invocation requirements, and preservation guarantee for the Constructor.

Certificate Proof Contract

Tool The name of the compression tool.

Applicable tasks The graph problems for which the compression argument holds.

Precondition The structural and task conditions checked before execution.

Transformation The graph records removed, contracted, or replaced, together with any aggregation rules.

Retained evidence The retained or aggregated information sufficient to preserve the task answer.

Execution-tool compatibility The output graph remains in the input domain of the selected execution tool.

Recovery The deterministic procedure that maps an answer or witness back to the original graph.

Figure 6: **Proof contract used for Certificate compression tools in CAGR.**

The following subsections present preservation arguments and execution-tool compatibility conditions for representative compression mechanisms under Eq. 10. Answer recovery follows Appendix C.1.

E.3 Task-General Compression Tools

Connected components and isolated vertices provide two basic structures for graph reduction. The following tools use these structures to remove irrelevant records while preserving answers across multiple graph tasks.

Task-General Compression

Connected-Component Pruning
Based on NetworkX

Applicable tasks Reachability, Shortest Path, Widest Path, Maximum Flow, and terminal Minimum Cut.

Precondition Both query terminals belong to the same connected component W, using weak connectivity for directed graphs.

Transformation Retain the induced subgraph $H[W]$.

Retained evidence All terminal-to-terminal paths and their edge attributes remain in W.

Recovery Values and paths are unchanged. Assign zero flow to removed edges and place each removed component wholly on either side of a cut.

Isolated-Vertex Removal
Based on NetworkX

Applicable tasks Terminal path and flow problems, Cycle Detection, and Maximum Matching.

Precondition Removed vertices have degree zero, are not query terminals, and are not required in the output.

Transformation Remove the selected isolated vertices I, yielding $H[V(H) \setminus I]$.

Retained evidence All edges and all query terminals are preserved.

Recovery Values and returned paths, cycles, flows, or matching edges remain unchanged.

Figure 7: **Task contracts for two task-general compression tools.**

Lemma E.1 (Connected-component pruning)**.** *Let s and t belong to the same connected component W of H, using weak connectivity for directed graphs. Restricting H to $H[W]$ preserves reachability, shortest-path distance, widest-path value, maximum-flow value, and minimum s–t cut capacity.*

Proof. No edge joins W to a removed component. Every s–t path therefore remains in $H[W]$ with unchanged attributes, preserving reachability and both path objectives. Restricting a feasible flow to W preserves its value, and any flow in $H[W]$ extends to H by assigning zero to removed edges. Likewise, any cut in $H[W]$ extends to H at the same capacity by placing each removed component wholly on one side, while restricting a cut of H to W cannot increase its capacity. Both optimal flow and cut values are consequently preserved. □

Lemma E.2 (Isolated-vertex removal)**.** *Let I consist of isolated vertices that are neither query terminals nor required output entries. Removing I preserves terminal path and flow answers, cycle existence, and maximum matchings.*

Proof. Vertices in I have no incident edges and cannot participate in a path between distinct terminals, a cycle, or a matching edge. Their removal leaves all remaining edges, attributes, and flow constraints unchanged. The feasible answer objects and their objective values are therefore preserved, with recovery given by the identity map. □

E.4 Task-Specific Compression Tools

Task-specific compression preserves the graph properties required by a particular task. Transitive reduction removes redundant edges while preserving reachability, whereas low-degree pruning removes vertices that cannot participate in triangles. Both operations have corresponding NetworkX primitives. Figure 8 summarizes their contracts, and the following lemmas establish answer preservation.

Task-Specific Compression

Transitive Reduction *Based on NetworkX*	**Low-Degree Pruning** *Based on NetworkX*
Applicable tasks Reachability.	**Applicable tasks** Triangle Counting.
Precondition The input is a directed acyclic graph.	**Precondition** The input is an undirected simple graph.
Transformation Remove redundant arcs while preserving all directed reachability relations.	**Transformation** Repeatedly remove vertices of current degree less than two and their incident edges.
Retained evidence Each removed arc has an alternative directed path in the reduced graph.	**Retained evidence** All original triangles are preserved.
Recovery Reachability answers are unchanged. Returned paths map directly to original arcs.	**Recovery** The triangle count is returned unchanged.

Figure 8: **Task contracts for two task-specific compression tools.**

Lemma E.3 (Transitive reduction)**.** *Let H be a directed acyclic graph and H^- its transitive reduction. For any $s, t \in V(H)$, t is reachable from s in H^- if and only if it is reachable from s in H.*

Proof. Every arc of H^- belongs to H, so every path in H^- is also a path in H. Conversely, each removed arc has an alternative directed path between its endpoints in H^-. Replacing removed arcs along any path in H therefore preserves reachability in H^-. Retaining original arc identifiers allows returned paths to be mapped directly to H. □

Lemma E.4 (Low-degree pruning)**.** *Let H be an undirected simple graph and let H_2 result from repeatedly deleting vertices of current degree less than two. Then H and H_2 contain exactly the same triangles.*

Table 17: Preservation and recovery mechanisms for the 77 Certificate compression tools.

Mechanism	Tools	Preservation and recovery
Irrelevant-record removal	9	Remove task-irrelevant components, dead ends, and inert records. Retain task-relevant evidence; assign zero flow to removed edges and restore removed components to compatible cut sides.
Component multiplicity reduction	1	Keep one representative of disconnected, attribute-isomorphic components. Recover counts and objective values using task-specific multiplicity rules, and map witnesses to the original components.
Path simplification	13	Task-specific dominance, bounds, twin quotients, and series or forced-segment rules preserve optimal path values after recovery. Expand stored paths and apply recorded distance offsets or bottleneck caps.
Reachability and cycles	8	Retain alternative routes, SCC information, or witness graphs. SCC sparsification preserves internal reachability; stored cycle information is retained where required.
Parallel-edge aggregation	3	Retain minimum weight, maximum capacity, or summed flow capacity according to the task. Select an original edge or split flow during recovery.
Flow and cut reduction	6	Series contraction, terminal normalization, certified high-capacity classes, and direct-arc offsets preserve optimal flow and cut values after recovery. Restore original flows or cut sides under the conditions below.
Boundary summaries	4	Store exact task-specific boundary relations, values, or matching states with corresponding witnesses. Expand selected summaries and restore any recorded objective offset.
Matching and disjoint paths	11	Exchange rules, bottleneck witnesses, and task-specific gadgets preserve cardinality. Query-bound sparse certificates preserve terminal edge connectivity; recover original matching edges or disjoint paths.
Triangle counting	8	Remove zero-contribution records or retain exact triangle offsets. Add each contribution once; directed checks cover all vertex roles in the specified triangle pattern.
Spanning forests	3	Cut and cycle properties justify contraction or deletion; zero-weight components retain a spanning forest. Restore selected original edges and weight offsets.
Minimum-cost flow	8	Preserve feasibility and optimal cost after recovery under the conditions below. Zero-demand pendant pruning permits zero-flow extension; other rules restore local flows and apply cost offsets.
Replacement paths	3	Preserve original failure IDs, shortcut failure masks, and scenario-valid alternatives. Recover the complete indexed answer vector, including unreachable scenarios.
Total	**77**	

Proof. Suppose a vertex belonging to an original triangle is deleted, and consider the first deleted vertex of that triangle. Its two triangle neighbors and the edges connecting them to it are still present, giving it degree at least two, a contradiction. Thus every original triangle survives. Since H_2 is an induced subgraph of H, no new triangles are introduced, and the triangle count is unchanged. □

E.5 Coverage and Composition

Table 17 groups the 77 Certificate compression tools into 12 categories and summarizes their preservation and recovery mechanisms. Each tool applies under its task-specific conditions.

Task-specific conditions. For directed flow and cut tasks, high-capacity cut-equivalence quotienting requires strongly connected classes. Recovery decomposes the reduced flow into terminal paths and routes them through the stored internal paths, whose capacities exceed the certified cut bound. Terminal-side fixing preserves the optimal cut value and supports cut recovery, but does not provide

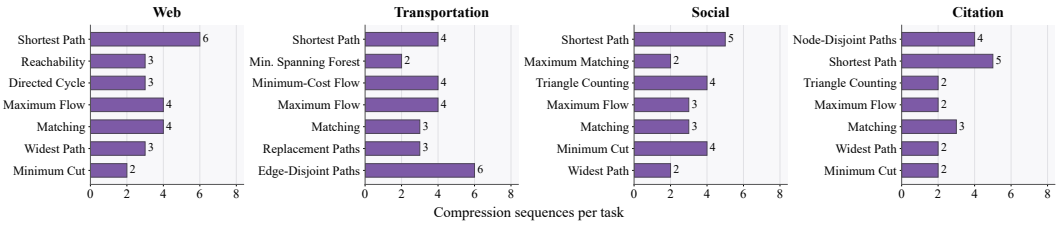


Figure 9: Instance diversity of compression chains on GraphLarge. Each bar reports the number of distinct compression toolchains generated for 60 cases of one task.

flow recovery. Directed flow-series contraction replaces a nonterminal vertex with one incoming and one outgoing arc by a shortcut with their minimum capacity. Existing parallel arcs remain separate. Recovery assigns the shortcut flow to both original arcs and places the removed vertex on a cut side consistent with their capacities.

Boundary summaries retain different information for different tasks. An undirected two-boundary flow module with no internal query terminal is replaced by an edge whose capacity equals its exact maximum-flow value. For Edge-Disjoint Paths, original edges have unit capacity, and recovery uses an integral path packing. This scalar summary does not cover arbitrary directed or vertex-capacitated flow. A one-boundary matching module instead records the maximum matching sizes μ_0 and μ_1 with the boundary vertex unmatched or matched internally. The reduced graph retains μ_0 as an offset and adds a leaf only when $\mu_1 = \mu_0 + 1$. For bipartite matching, the leaf lies on the side opposite the boundary vertex.

For Minimum-Cost Flow, zero-cost SCC contraction requires internal capacities sufficient to route every feasible residual boundary imbalance, with nonnegative costs and no lower bounds. A zero-demand series vertex is replaced by an arc with summed unit cost and minimum capacity. A feasible, bounded one-boundary module is solved locally; its total demand is transferred to the boundary, and its optimal cost and flow are retained for recovery. An expensive parallel arc is removed only when a retained arc of no greater unit cost can carry a certified upper bound on the entire bundle's feasible flow. Recovery restores the original flow balances and accounts for stored cost offsets.

Safe composition. Each compression tool checks its applicability on the current graph and preserves compatibility with the selected execution tool. For a supported task, an inapplicable structural transformation leaves the graph unchanged; unsupported tasks or incompatible representations invalidate the toolchain. Recovery proceeds in reverse order as in Eq. 12, restoring original identifiers and applying each offset once. Under the conditions in Eq. 10, Safe Prefix Execution follows by induction. Condition checks, Certificate graph construction, and local solving are included in compression cost; answer recovery is included in execution cost.

F Supplementary Experiments

F.1 Instance Diversity of Compression Chains

We examine the diversity of compression toolchains generated by the Constructor across instances of the same task. We generate one toolchain per case for 60 test cases per in-distribution task and a fixed subset of 60 cases per out-of-distribution task. Among valid toolchains, we count distinct compression-tool sequences, retaining tool order and repetitions but excluding argument values and execution-tool identities; an empty sequence counts as one sequence. Figure 9 shows multiple sequences for all 28 task settings. These counts characterize the variation in generated tool compositions, while Appendix F.2 further evaluates the execution benefits of instance-dependent toolchain construction.

F.2 Adaptive Execution and Chain Construction

We evaluate adaptive execution and instance-dependent chain construction on the GraphLarge in-distribution test cases. Time is macro-averaged across tasks and averaged over five runs for all

Table 18: Execution policies and chain construction on four-domain GraphLarge. Time reports mean execution time over five runs. CAGR entries are taken from Table 1.

Variant	Time (s)	Relative Cost
Execution Policy		
Direct	39.34	1.000
Full Chain	10.04	0.255
Static Prefix	12.48	0.317
CAGR Gate	7.45	0.189
Best Measured Prefix	6.76	0.172
Chain Construction		
Per-task Static Chain	10.73	0.273
CAGR	7.45	0.189

variants. The CAGR Gate and CAGR rows use the five-run mean reported in Table 1. Relative Cost is normalized by Direct.

Execution Policy. All policies reuse the same toolchain and execution tool generated by the frozen Constructor for each case, varying only the execution depth. Direct performs no compression; Full Chain executes all compression tools. Static Prefix uses a task-level depth selected by minimum mean validation time, with depth zero and full-chain execution included as candidates; chains shorter than the selected depth execute in full. CAGR Gate selects when to execute for each instance. Best Measured Prefix retrospectively selects the fastest prefix, including depth zero, without enumeration cost. All policies include Constructor inference time and, where applicable, Gate feature acquisition and prediction.

Chain Construction. Both variants use the same selected execution tool for each case, frozen Compression Gate, tool contracts, and graph backend. Per-task Static Chain selects one compression sequence per task from training-search candidates by minimum mean validation time with the Gate, binding its arguments to each test request. CAGR generates a sequence for each instance. The Gate controls execution depth in both variants.

Table 18 shows that CAGR Gate reduces execution time by 25.80% relative to Full Chain and 40.30% relative to Static Prefix, while remaining 0.69 seconds per case above Best Measured Prefix. Adaptive stopping therefore improves execution efficiency even when toolchain construction is fixed. With the execution tool and Gate held fixed, CAGR further reduces time by 30.57% relative to Per-task Static Chain. Instance-dependent tool composition and adaptive execution depth provide complementary benefits: the Constructor determines which compression tools to compose, while the Gate determines how much of the resulting chain to execute.

F.3 Execution Overhead without Compression Gains

We construct 60 cases for each of Cycle Detection, Maximum Flow, and Matching to evaluate execution cost when compression provides no graph reduction. Each case has a nonempty compression chain whose tools satisfy their task and representation contracts but leave graph size unchanged. Direct, Full Chain, and CAGR Gate reuse the same toolchain and selected execution tool generated by the frozen Constructor. Direct executes without compression, Full Chain invokes every compression tool, and CAGR Gate selects when to execute. All policies include Constructor inference time and, where applicable, condition checks and Gate feature acquisition and prediction. Relative Cost compares each policy's mean execution time with Direct within each task; Depth-0 Stop reports the percentage of cases where the Gate invokes the execution tool without compression.

Table 19 shows that Full Chain increases execution time by 18.0–57.0% relative to Direct without reducing graph size. CAGR Gate stops before compression on most cases, limiting the additional cost to 0.7–3.9%. Although Gate evaluation and the remaining compression attempts still incur costs, adaptive stopping substantially reduces the expense of executing compression tools without graph reduction.

Table 20: Gate feature ablation across requests on fixed graphs. Time reports execution time; Excess Time measures the additional time relative to Best Measured Prefix (both in s/case). Results are averaged over five runs.

Gate Variant	Time	Excess Time
w/o Query-Node Statistics	12.57	1.66
w/o Plan Complexity	13.33	2.42
Full Gate	11.67	0.76
Best Measured Prefix	10.91	0.00

Table 19: Execution cost on constructed cases where the selected compression chain leaves graph size unchanged. Relative Cost is mean execution time relative to Direct; Depth-0 Stop is the percentage of cases where the Compression Gate invokes the execution tool without compression.

	Relative Cost			Depth-0 Stop (%)
Task	Direct	Full Chain	CAGR Gate	CAGR Gate
Cycle Detection	1.000	1.570	1.039	85.00
Maximum Flow	1.000	1.278	1.034	81.67
Matching	1.000	1.180	1.007	93.33

F.4 Gate Features for Different Requests on the Same Graph

We evaluate the contribution of query-node statistics and plan complexity to Compression Gate decisions on fixed graphs. We use one sampled test graph from each of Web and Social, with 20 terminal-pair requests per graph for each of Shortest Path and Maximum Flow, giving 80 requests. Graph structure and task attributes remain fixed within each graph–task group. The frozen Constructor generates one toolchain per request, shared across all Gate variants. We compare the full Gate with variants omitting either feature group, each trained separately on the same prefix-cost examples using the same procedure.

Time follows the common timing protocol, including Constructor inference, compression, Gate evaluation, and task execution. Best Measured Prefix retrospectively selects the fastest prefix of each chain, including depth zero, without Gate or enumeration costs. Excess Time measures the mean per-request difference from this reference. Results are averaged equally across the four graph–task groups and over five runs.

Table 20 shows that removing query-node statistics or plan complexity increases execution time from 11.67 seconds to 12.57 and 13.33 seconds per case, respectively. Both feature groups improve stopping decisions: query-node statistics describe the structural context of the request, while plan complexity describes the computational requirements of the remaining tools. The full Gate remains 0.76 seconds per case above Best Measured Prefix, including Gate evaluation and the additional cost of its selected prefixes.

F.5 MCTS Search Budget

Evaluating multiple MCTS search budgets across all domains is computationally expensive, so we use the Transportation domain of GraphLarge to guide budget selection. We compare 32, 64, 128, 256, and 512 completed toolchain evaluations per case, including invalid toolchains, while keeping the tool library and maximum compression-chain length fixed.

Search Quality. We freeze the same Tool Warm Up checkpoint and run MCTS on a fixed set of Transportation training cases using paired search seeds. For each case, we record the highest reward among valid evaluated toolchains, assigning -1 when no valid toolchain is found. Results are averaged equally across tasks. Holding the Constructor fixed isolates the effect of additional search on the quality of discovered toolchains.

Policy Improvement. We train a separate Constructor for each budget from the same initialization, keeping training data, CCO iterations, policy-update steps, and other training settings fixed. Each

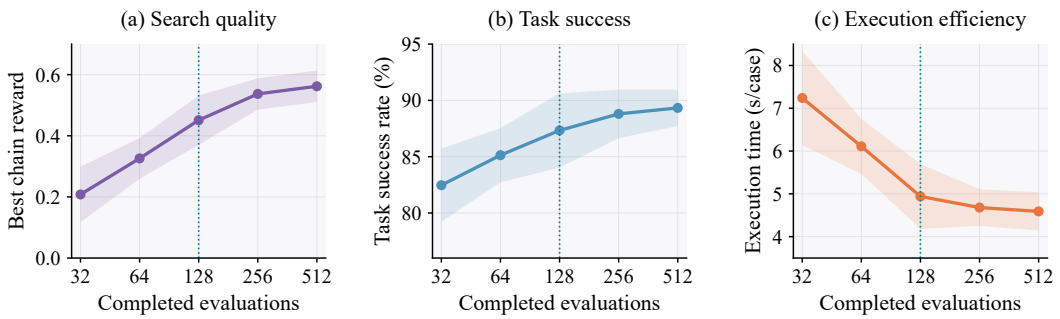


Figure 10: MCTS search-budget results on Transportation. (a) Best observed toolchain reward with a fixed Constructor. (b) Task success rate and (c) execution time after CCO training, using full-chain execution. Curves and shaded bands show five-run means and standard deviations.

Constructor is evaluated on the same 300 Transportation validation cases, comprising 60 cases per task. Toolchains are executed in full without the Compression Gate. Success and execution time follow the common evaluation protocol and are averaged over five runs.

Figure 10 shows that larger budgets improve the best observed toolchain reward and yield Constructors with higher task success and lower execution time. Execution time decreases most between budgets of 64 and 128, while further increases yield smaller gains in task success and execution efficiency. Based on the Transportation validation results, we select 128 completed evaluations per case as the default MCTS budget, balancing search cost with the quality of the learned toolchains.

G Prompt Templates

G.1 Complete Toolchain Construction

System instructions: Constructor

Construct a complete toolchain for the given graph task using the provided tool library. Select one execution tool and an ordered sequence of at most 10 compression tools. Use documented tool IDs and argument fields, and bind arguments to the task inputs, respecting supported tasks and compatibility requirements.

Return only a JSON object in the specified format. Place the executor field before the compression field. Compression tools execute in list order, followed by the execution tool. The compression list may be empty and may contain repeated tools. Do not output the task answer or explanatory prose.

Instance input: Constructor

Task: [task definition and instance request]
Graph handle: [graph reference]
Public graph summary: [structural-summary fields]
Tool library: [complete serialized tool descriptions]

Required output format: Constructor

```
{
  "executor": {"tool": "[execution tool ID]",
    "arguments": [argument object]},
  "compression": [
    {"tool": "[compression-tool ID]",
      "arguments": [argument object]}
  ]
}
```

Figure 11: Constructor prompt and required output format. Bracketed fields are filled for each request.

Bracketed values are placeholders for schema-conforming tool IDs and argument objects. The compression array contains zero to ten calls; one entry is shown to specify its format. The environment binds each call to the current graph and maintains identifier mappings required after compression, as described in Appendix C. Tool Warm Up uses individual tool descriptions and invocation requests to train single-tool calls in the same tool-and-arguments format.

G.2 MCTS Prefix Continuation

MCTS reuses the Constructor instructions and instance input, with the serialized partial toolchain supplied as an assistant prefix. At the root, generation begins in the executor field. After execution-tool selection, each continuation appends a compression-tool call or closes the compression array, corresponding to STOP. Sampled action frequencies define the search prior. Search values and measured execution costs are used by the training procedure, not included in the prompt.

H Case Studies

H.1 Shared Settings

Shared Graph. Both queries use the same WDC induced graph with 29,438,617 nodes and 1,054,030,380 directed edges, sampled using the multi-start random-walk procedure in Appendix D.2. Edge costs are fixed positive integers in $\{1, \ldots, 50\}$, assigned before query timing.

Interface. Both cases follow the Constructor prompt in Appendix G. Tool names identify the corresponding compression and execution operations, and the runtime binds each call to the current graph.

Tool library: representative descriptions

Format: ID | Name | Description

Execution Tool

`[ID]` | `shortest_path(source, target, weight)` | Return a minimum-cost directed path and its cost; requires nonnegative weights.

Compression Tools

`[ID]` | `source_scope(source)` | Retain source-reachable vertices and their induced edges.

`[ID]` | `remove_self_loops()` | Remove positive-cost self-loops; preserve vertices and shortest-path answers.

The three entries illustrate the tool-description format. The complete toolchains for cases are shown below.

Figure 12: Representative tool descriptions used to present the case studies.

Table 21: Compression operations shared by both cases.

Depth	Presentation alias	Measured operation
Depth 1	`remove_self_loops`	Remove positive-cost self-loops
Depth 2	`source_scope`	Source-descendant restriction
Depth 3	`target_scope`	Target-ancestor intersection
Depth 4	`series_contract`	Nonterminal series contraction; sum costs
Depth 5	`parallel_minimum`	Minimum-cost edge per ordered parallel pair

Timing. Compression tools use compiled CSR implementations, and the execution tool uses NetworkX 3.6.1's `single_source_dijkstra` through a packed-CSR adapter. We profile each prefix separately, reporting cumulative compression time plus task execution time on the resulting graph. These graph-side times exclude initial graph loading, intermediate serialization, Constructor inference, and Gate evaluation.

H.2 Graph Summary and Gate Inputs

Both cases use the graph summary in Table 22. Table 23 summarizes the Gate inputs, including measured record-access fractions and complexity features computed from the current graph size and remaining tools. Table 24 reports the recorded Gate scores and stopping decisions alongside cost-derived log-ratios comparing immediate execution with the best later completion. These ratios exclude previously incurred compression costs and future Gate evaluation time.

Table 22: Shared graph summary supplied to the Constructor for both cases.

Field	Value
Node count	29,438,617
Edge count	1,054,030,380 directed adjacency records
Packed graph storage size	5,630,151,908 bytes
Edge density (derived)	$1.21623735 \times 10^{-6}$, using $m/[n(n-1)]$
Out-degree min / max / mean	0 / 8,479 / 35.804344
Out-degree P50 / P90 / P99	13 / 84 / 257
In-degree min / max / mean	0 / 960,978 / 35.804344
In-degree P50 / P90 / P99	0 / 5 / 120
Orientation	Directed
Node labels	`Page`
Relationship types	`HYPERLINK`
Node properties / types	None
Edge properties / types	`cost`: integer, 1–50

Table 23: Compression Gate inputs for the two cases.

Field group	Description
Original graph summary	Measured counts, density, and degree statistics above; directed flag 1.
Current graph counts	n_d, m_d from each case's measured prefix table.
Previous compression fractions	$1 - n_d/n_{d-1}$ and $1 - m_d/m_{d-1}$ for $d > 0$; both 0 at Depth 0.
Record-access fractions	Last and cumulative node- and edge-record accesses, normalized by the original counts; repeated accesses are counted.
Execution progress	Completed steps d; remaining steps $5 - d$.
Query-node statistics	Original-graph in/out-degrees of the source and target, with presence masks.
Plan complexity	Complexity values of the remaining compression tools and selected execution tool, evaluated at the current graph size using Eq. 22, with presence masks.

Table 24: Compression Gate scores and stopping decisions for the two cases.

Case	Depth	Cost-derived log-ratio	Gate score	Decision
A	Depth 0	$+1.692$	$+1.44$	Continue
A	Depth 1	$+2.034$	$+1.60$	Continue
A	Depth 2	$+2.138$	$+1.71$	Continue
A	Depth 3	-0.005	-0.02	Stop
B	Depth 0	-0.018	-0.03	Stop

H.3 CASE A: COMPRESSION IS WORTHWHILE

This case illustrates how compression reduces the cost of answering a shortest-path request on a billion-edge graph. Figure 13 presents the request, and Figure 14 shows the toolchain. The Compression Gate selects execution at Depth 3 (Table 24), where the Certificate graph contains 7.48 million edges. Cumulative compression plus execution-tool time decreases from 101.143 seconds for Direct execution to 18.635 seconds, while further compression leaves the graph size unchanged and slightly increases total time (Table 25). Table 26 reports the corresponding record-access fractions and execution-tool complexity. The recovered path preserves the original shortest-path cost of 332.

Constructor Input

Task: Starting from page 22558015, follow directed links to page 1145864. Each link has a `cost`. Return a route with the lowest total cost and report that cost.
Graph handle: `random_walk_st_002`
Public graph summary: Shared graph summary in Table 22.
Tool library: The full compression and execution tool library, with representative descriptions in Appendix H.1.

Figure 13: Constructor input for Case A.

Output toolchain

```
{
  "executor": {"tool": "shortest_path",
    "arguments": {"source": 22558015, "target": 1145864, "weight": "cost"}},
  "compression": [
    {"tool": "remove_self_loops",
     "arguments": {}},
    {"tool": "source_scope",
     "arguments": {"source": 22558015}},
    {"tool": "target_scope",
     "arguments": {"target": 1145864}},
    {"tool": "series_contract",
     "arguments": {"terminals": [22558015, 1145864], "weight": "cost"}},
    {"tool": "parallel_minimum",
     "arguments": {"weight": "cost"}}
  ]
}
```

Figure 14: Output toolchain for Case A. Tool IDs are replaced with tool names for readability.

Table 25: Graph sizes and execution costs at each compression depth for Case A. Total includes cumulative compression and execution-tool time; bold marks Best Measured Prefix.

Depth	Nodes	Edges	Step comp. (s)	Cum. comp. (s)	Execution tool (s)	Total (s)
Depth 0	29,438,617	1,054,030,380	0.000	0.000	101.143	101.143
Depth 1	29,438,617	1,049,722,844	5.454	5.454	100.790	106.244
Depth 2	2,413,443	124,749,415	1.462	6.916	99.364	106.281
Depth 3	**228,614**	**7,481,265**	**1.352**	**8.268**	**10.367**	**18.635**
Depth 4	228,614	7,481,265	0.177	8.446	10.237	18.683
Depth 5	228,614	7,481,265	0.168	8.614	10.495	19.108

Table 26: Record access and complexity at each compression depth for Case A. Last access and Cum. access denote edge-access counts for the latest step and all completed steps, respectively, divided by the original edge count. Complexity denotes the execution-tool complexity proxy.

Depth	Last access	Cum. access	Complexity
Depth 0	0.000	0.000	24.02
Depth 1	1.000	1.000	24.01
Depth 2	0.355	1.355	21.72
Depth 3	0.258	1.614	18.74
Depth 4	0.021	1.635	18.74
Depth 5	0.007	1.642	18.74

H.4 Case B: Direct Execution Is Preferable

This case illustrates how compression can reduce graph size without reducing the cost of answering a shortest-path request on the same billion-edge graph. Figure 15 presents the request, and Figure 16 shows the toolchain. The Compression Gate selects Direct execution at Depth 0 (Table 24). Although compression reduces the graph to 68.18 million edges at Depth 3, its cost of 8.764 seconds exceeds the execution-tool savings of 7.643 seconds. Cumulative compression plus execution-tool time is therefore 64.514 seconds, compared with 63.393 seconds for Direct execution, and later prefixes further increase total time (Table 27). Table 28 reports the corresponding record-access fractions and execution-tool complexity. The recovered path preserves the original shortest-path cost of 283.

Constructor Input

Task: Starting from page 10195586, follow directed links to page 24180157. Each link has a `cost`. Return a route with the lowest total cost and report that cost.
Graph handle: `random_walk_st_002`
Public graph summary: Shared graph summary in Table 22.
Tool library: The full compression and execution tool library, with representative descriptions in Appendix H.1.

Figure 15: Constructor input for Case B.

Output toolchain

```
{
  "executor": {"tool": "shortest_path",
    "arguments": {"source": 10195586, "target": 24180157, "weight": "cost"}},
  "compression": [
    {"tool": "remove_self_loops",
     "arguments": {}},
    {"tool": "source_scope",
     "arguments": {"source": 10195586}},
    {"tool": "target_scope",
     "arguments": {"target": 24180157}},
    {"tool": "series_contract",
     "arguments": {"terminals": [10195586, 24180157], "weight": "cost"}},
    {"tool": "parallel_minimum",
     "arguments": {"weight": "cost"}}
  ]
}
```

Figure 16: Output toolchain for Case B. Tool IDs are replaced with tool names for readability.

Table 27: Graph sizes and execution costs at each compression depth for Case B. Total includes cumulative compression and execution-tool time; bold marks Best Measured Prefix.

Depth	Nodes	Edges	Step comp. (s)	Cum. comp. (s)	Execution tool (s)	Total (s)
Depth 0	**29,438,617**	**1,054,030,380**	**0.000**	**0.000**	**63.393**	**63.393**
Depth 1	29,438,617	1,049,722,844	5.870	5.870	63.238	69.107
Depth 2	2,053,744	103,209,302	1.305	7.175	61.170	68.345
Depth 3	1,365,965	68,181,633	1.589	8.764	55.750	64.514
Depth 4	1,362,776	68,178,444	2.360	11.124	54.456	65.580
Depth 5	1,362,776	68,176,044	2.374	13.498	54.074	67.572

Table 28: Record access and complexity at each compression depth for Case B. Last access and Cum. access denote edge-access counts for the latest step and all completed steps, respectively, divided by the original edge count. Complexity denotes the execution-tool complexity proxy.

Depth	Last access	Cum. access	Complexity
Depth 0	0.000	0.000	24.02
Depth 1	1.000	1.000	24.01
Depth 2	0.294	1.294	21.52
Depth 3	0.390	1.684	21.07
Depth 4	0.194	1.878	21.07
Depth 5	0.065	1.943	21.07

I Reproducibility Statement

We have provided implementation and measurement details in Appendix B, complete method specifications in Appendix C, the GraphLarge construction and evaluation pipeline in Appendix D, Certificate compression-tool contracts and preservation arguments in Appendix E, additional experiments in Appendix F, and prompt templates in Appendix G. These materials document the training, evaluation, data construction, and tool execution procedures used in our experiments. Our anonymous repository[2] provides the core CAGR workflow code, along with preservation proofs, supported tasks and applicability conditions, and mappings between tool IDs and implementations for all Certificate compression tools.

[2] https://anonymous.4open.science/r/ICLR2027-CAGR