ALGOSAGE: FORMAL PROBLEM REPRESENTATION AND GROUNDED EXPERTISE FOR ALGORITHMIC CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have demonstrated strong capabilities in automated code generation; however, achieving reliable performance on algorithmic programming problems remains challenging, as implicit problem abstraction and free-form planning can lead to inappropriate algorithm choices, underspecified solution plans, and constraint-violating implementations. Our key insight is that reliable code generation requires making the computational structure of a problem explicit, using it to select appropriate algorithmic expertise, and translating that expertise into concrete plans before coding. We propose AlgoSAGE, which constructs a formal problem representation and retrieves expertise at two complementary levels: reusable Algorithmic Knowledge and Algorithmic Application Exemplars that demonstrate how the retrieved knowledge is applied. AlgoSAGE then adapts this evidence into target-specific, executable, and auditable plans. Extensive experiments across multiple benchmarks and LLM backbones demonstrate that AlgoSAGE consistently improves algorithmic code generation. With Qwen3.8-27B-Coder, AlgoSAGE achieves 70.3%, 76.0%, and 78.3% Pass@1 on CodeContests, APPS, and xCodeEval, respectively, outperforming existing approaches by up to 6.7 percentage points. Experiments with additional backbones and function-level programming benchmarks further demonstrate the robustness and general applicability of AlgoSAGE. Code is available at `https://anonymous.4open.science/r/AlgoSage-82F9/`.

1 INTRODUCTION

LLMs have become increasingly capable of generating executable programs from natural-language specifications. Their strong performance on function-level benchmarks such as HumanEval and MBPP demonstrates that modern models can synthesize short programs from docstrings or concise task descriptions (Chen et al., 2021; Austin et al., 2021). However, this success does not transfer reliably to algorithmically demanding programming problems. Competition-style tasks require models to infer the computational structure underlying a problem, select suitable algorithms and data structures, and implement solutions that remain correct and efficient under large input sizes and explicit time and memory limits (Li et al., 2022). Prior work further shows that failures arise both from specifying incorrect or incomplete algorithms and from incorrectly implementing otherwise valid solution ideas (Hadhoud et al., 2026). These limitations are more fully exposed by broader execution-based benchmarks such as APPS, CodeContests, and xCodeEval (Hendrycks et al., 2021; Li et al., 2022; Khan et al., 2023).

Recent work has improved code generation through explicit planning, multi-agent collaboration, retrieval augmentation, execution feedback, simulation, and iterative repair (Dong et al., 2024; Huang et al., 2024; Islam et al., 2024; 2025; Wang et al., 2025). However, these methods still rely largely on the model's free-form reasoning to select an appropriate algorithm for the target problem. When this upstream decision is implicit, subsequent planning, coding, and debugging may continue refining a realization of an inappropriate algorithm rather than reconsidering the underlying algorithm choice.

To illustrate this limitation, we trace CodeSIM's (Islam et al., 2025) reasoning, simulation, and repair process on ChainWord (Appendix A). CodeSIM recalls a related fixed-string segmentation

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.

problem and uses simulation to correct a counting error. However, the correction focuses on counting correctness without translating the target problem's computational constraints into a feasible algorithm. The revised program explicitly enumerates generated strings, passing the public samples but remaining exponential in the worst case and failing the final evaluation.

This failure highlights that planning and sample-level simulation alone do not ensure an appropriate algorithmic choice for the target problem. We therefore make the computational information governing algorithm selection explicit and use it to retrieve applicable algorithmic expertise. Specifically, AlgoSAGE constructs a formal problem representation that makes the computational information relevant to algorithm selection explicit. It then retrieves applicable Algorithmic Knowledge and, conditioned on the retrieved knowledge, identifies Application Exemplars showing how that knowledge is instantiated in problems with similar operations and constraints. The resulting evidence is used to synthesize target-specific and auditable solution plans.

Extensive experiments across multiple LLM backbones show that AlgoSAGE consistently outperforms all compared baselines across HumanEval, MBPP, xCodeEval, CodeContests, and APPS, with particularly large gains on the more challenging algorithmic benchmarks xCodeEval, CodeContests, and APPS. The results on HumanEval and MBPP further show that these improvements do not come at the expense of performance on simpler function-level code-generation tasks. Ablation studies further analyze the contributions of different components.

Our main contributions are threefold:

- We introduce a formal representation for algorithmic programming problems and construct an expert memory that connects formalized problem characteristics to applicable algorithmic expertise, enabling structure-grounded algorithm selection.
- We propose a multi-agent framework that leverages this memory to retrieve appropriate Algorithmic Knowledge and Application Exemplars, and transforms them into target-specific and auditable solution plans for code generation.
- We conduct extensive experiments across multiple LLM backbones and code-generation benchmarks, demonstrating consistent improvements over strong baselines, particularly on challenging algorithmic problems.

2 Related Work

2.1 Problem Representation and Solution Planning

Recent work has increasingly made intermediate problem-solving artifacts explicit before code generation. Hadhoud et al. use natural-language editorials to separate algorithmic reasoning from code implementation, revealing that specifying a correct and complete algorithm remains a major bottleneck (Hadhoud et al., 2026). StoryCoder instead reformulates fragmented problem statements into coherent narratives containing task overviews, constraints, and examples, which can guide subsequent reasoning toward better algorithmic strategies (Jang et al., 2026). Different from these approaches, AlgoSAGE formalizes the computational information relevant to algorithm selection and uses it to retrieve applicable Algorithmic Knowledge and corresponding Application Exemplars for target-specific solution planning.

2.2 Multi-Agent Code Generation, Verification, and Repair

Multi-agent code-generation systems decompose problem solving into specialized roles for analysis, planning, coding, testing, and repair (Dong et al., 2024; Huang et al., 2024; Islam et al., 2024; 2025; Wang et al., 2024). Recent methods further adapt when planning and debugging are invoked, allow plans and programs to be revised jointly, or apply multiple verification perspectives to intermediate reasoning (Zhu et al., 2025; Yoon et al., 2026; Doan et al., 2026; Xu et al., 2026). Together, these approaches improve workflow orchestration, execution feedback, and failure recovery. Unlike workflow-oriented methods, AlgoSAGE grounds upstream algorithmic decisions in structured intermediate artifacts and audits generated implementations against their intended plans before final acceptance.

2.3 Retrieval-Augmented Code Generation

Retrieval-augmented methods supplement model-internal reasoning with external code, documentation, tutorials, solved problems, or reusable procedural knowledge. CodeRAG-Bench shows that retrieval quality and effective context utilization are both critical to downstream code generation (Wang et al., 2025). To reduce dependence on surface similarity, PERC retrieves examples through pseudocode-level plans, while SolveRank learns solution-aware representations for retrieving problems with related algorithmic logic (Yoo et al., 2025; Zhang et al., 2025). Other methods organize reusable expertise into procedural abstractions for program optimization or algorithm design (Wang et al., 2026; Song et al., 2026). Different from prior retrieval-based methods, AlgoSAGE first retrieves applicable Algorithmic Knowledge and then corresponding Application Exemplars from the formalized problem structure. The retrieved expertise is adapted into target-specific and auditable plans before code generation.

We provide a more detailed review of related work and additional comparisons with representative approaches in Appendix B.

3 Methodology

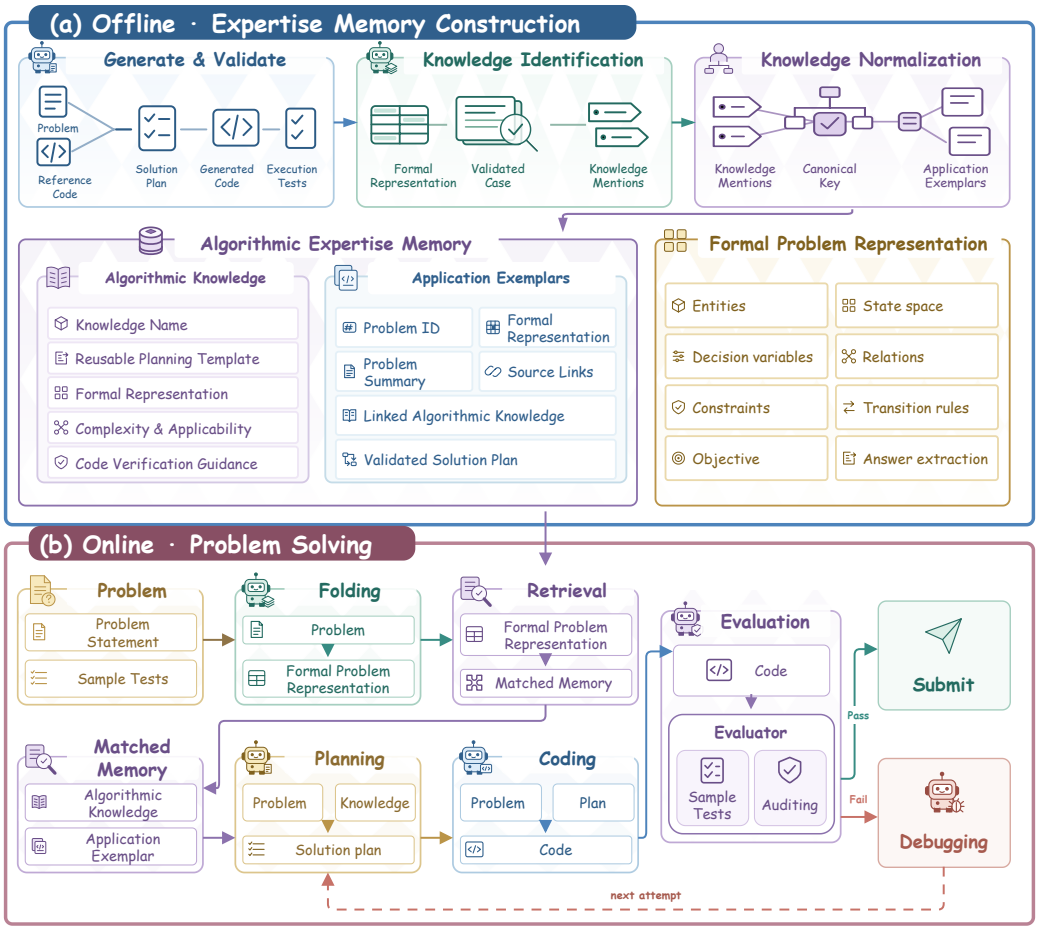


Figure 1: Overview of AlgoSAGE

This section presents the design of AlgoSAGE. We first introduce the formal representation used to expose the computational structure of algorithmic programming problems, and then describe the

expert memory that links these representations to reusable Algorithmic Knowledge and concrete Application Exemplars. Based on this memory, AlgoSAGE performs structure-grounded algorithm selection by retrieving expertise that matches the target problem. We then present the multi-agent framework that turns the retrieved expertise into target-specific solution plans, generates executable code, and evaluates the code together with its associated plan.

3.1 Formal Problem Representation

Algorithm selection for programming problems relies on identifying the computational characteristics that make certain methods applicable. However, natural-language problem statements often mix narrative descriptions, input-output specifications, and constraints, making these characteristics implicit. Directly matching raw problem text may therefore emphasize surface-level similarity rather than the structural factors required for expertise retrieval. AlgoSAGE introduces a structured representation to capture the algorithm-relevant properties of each problem:

$$\mathcal{M}_P = (\mathcal{E}_P, \mathcal{X}_P, \mathcal{V}_P, \mathcal{R}_P, \mathcal{C}_P, \mathcal{T}_P, \mathcal{O}_P, \mathcal{A}_P), \tag{1}$$

where the eight fields represent entities, state space, decision variables, relations, constraints, transition rules, objective, and answer extraction. Together, these fields describe the objects involved in computation, their structural dependencies, state evolution, feasibility requirements, and output semantics. Each field contains a list of controlled structural labels, interpreted as a set for structural matching:

$$\mathcal{M}_P^{(f)} = \boldsymbol{\tau}_P^{(f)}, \qquad f \in \mathcal{F}. \tag{2}$$

The structural labels provide consistent matching signals across problems. The representation may include inferred computational mechanisms but does not constitute a complete solution plan. It exposes the computational structure relevant to algorithm applicability, providing a common basis for retrieving reusable Algorithmic Knowledge and Application Exemplars. Detailed field definitions, vocabularies, and examples are provided in Appendix D.

3.2 Algorithmic Expertise Memory and Structure-Grounded Retrieval

The formal problem representation introduced in Section 3.1 establishes a structured connection between problem characteristics and algorithm applicability. However, such a representation only provides a basis for matching; effective algorithm selection additionally requires a corresponding repository of algorithmic knowledge. AlgoSAGE therefore constructs an *Algorithmic Expertise Memory*, where Algorithmic Knowledge and Algorithmic Application Exemplars are organized under the same structural representation schema. The memory contains two complementary components: *Algorithmic Knowledge* and *Algorithmic Application Exemplars*. Algorithmic Knowledge represents reusable algorithmic concepts, including their applicable problem patterns, planning templates, complexity requirements, and Evaluation guidance. Algorithmic Application Exemplars preserve concrete problem instances where specific Algorithmic Knowledge has been successfully applied, including their problem representation, associated knowledge categories, and execution-verified solution plans.

Construction of Algorithmic Expertise Memory. We next describe the construction process of the Algorithmic Expertise Memory. The construction starts from a predefined Algorithmic Knowledge taxonomy and extends it with knowledge extracted from execution-verified solution cases.

We first construct a fixed Algorithmic Knowledge taxonomy from OI Wiki, covering commonly used algorithms and data structures in competitive programming. Each entry corresponds to a canonical knowledge key, which provides a stable vocabulary for organizing algorithmic expertise. Given programming problems from the CodeContests training set, AlgoSAGE performs solution planning with access to the official reference solutions. The reference solutions are only used during memory construction to provide guidance for generating reliable solution plans and are unavailable during online problem solving. Each generated plan is implemented and evaluated through execution tests, and only problems whose implementations pass the evaluation are retained. For each retained solution plan, AlgoSAGE identifies the algorithmic concepts used in the solution. This step, referred to as *Algorithmic Knowledge Identification*, extracts the knowledge categories required by the solution plan. Since the same algorithmic concept may appear under different names or descriptions,

the extracted concepts are represented as *Algorithmic Knowledge Mentions* and are subsequently normalized to the canonical knowledge keys defined in the taxonomy.

The normalized knowledge keys associate each successful problem with its corresponding Algorithmic Knowledge entries. Each retained problem is stored as an *Algorithmic Application Exemplar*, containing its problem representation, linked Algorithmic Knowledge entries, and validated solution plan. Before exporting the memory, we remove Application Exemplars whose problem statements overlap with evaluation benchmarks, including APPS and xCodeEval, to prevent information leakage.

Algorithmic Knowledge Identification and Knowledge Normalization are only used during memory construction. During online inference, AlgoSAGE retrieves existing Algorithmic Knowledge and Application Exemplars based on the formal problem representations.

Hierarchical structure-grounded retrieval. After constructing the Algorithmic Expertise Memory, AlgoSAGE retrieves relevant Algorithmic Knowledge and Algorithmic Application Exemplars during problem solving. Given a target problem, its formal representation $\widehat{\mathcal{M}}_P$ serves as the retrieval query for matching both knowledge entries and application examples.

Before retrieving concrete examples, AlgoSAGE first identifies applicable Algorithmic Knowledge. This avoids selecting Application Exemplars solely because they exhibit similar problem structures while belonging to inappropriate algorithmic categories. Therefore, AlgoSAGE performs hierarchical retrieval: it first retrieves Algorithmic Knowledge entries to determine candidate solution methods, and then searches for Application Exemplars associated with these methods.

First, AlgoSAGE compares $\widehat{\mathcal{M}}_P$ with the structural representation of each Algorithmic Knowledge entry. The similarity is computed through weighted overlap among the eight fields, where fields containing more discriminative structural signals contribute larger weights. After filtering entries below the similarity threshold, the top $K_{\text{skill}} = 3$ Algorithmic Knowledge entries are retained:

$$\mathcal{K}_P = \text{TopK}\left\{\text{Sim}(\widehat{\mathcal{M}}_P, \mathcal{M}_i^{\text{alg}})\right\}. \tag{3}$$

The retrieved Algorithmic Knowledge defines a constrained search space for Application Exemplar retrieval. Instead of comparing against all stored examples, AlgoSAGE only considers exemplars associated with the retrieved knowledge entries. This design ensures that retrieved examples are not only structurally similar to the target problem but also implement relevant algorithmic methods.

The candidate exemplars are ranked according to the similarity between their problem representations and the target representation:

$$\mathcal{I}_P = \text{TopK}\left\{\text{Sim}(\widehat{\mathcal{M}}_P, \mathcal{M}_j^{\text{app}}) \mid \mathcal{M}_j^{\text{app}} \in \mathcal{I}(\mathcal{K}_P)\right\}. \tag{4}$$

When selecting multiple Application Exemplars, AlgoSAGE further applies knowledge coverage diversification to avoid concentrating all retrieved examples on a single Algorithmic Knowledge category. This encourages the retrieved evidence to cover different applicable solution methods when multiple algorithmic directions are relevant to the target problem. The detailed similarity computation and retrieval procedure are provided in Appendix C.

This hierarchical retrieval separates algorithm selection from solution instantiation. Algorithmic Knowledge retrieval identifies reusable methods whose applicability matches the target structure, while Application Exemplar retrieval provides concrete examples showing how these methods can be adapted under specific operations and constraints.

For each planning trajectory, AlgoSAGE forms a compact retrieval context by pairing one selected Application Exemplar with its associated Algorithmic Knowledge entries rather than exposing the entire retrieved memory. The use of these retrieved artifacts for solution planning is described in Section 3.3.

3.3 Expertise-Grounded Multi-Agent Code Generation

With the formal problem representation and Algorithmic Expertise Memory introduced above, we next describe how AlgoSAGE integrates these components into a complete code generation frame-

work. The framework consists of Folding, Retrieval, Planning, Coding, Evaluation, and Debugging, where each component produces an intermediate artifact for the subsequent stage.

Folding. Since natural-language problem statements do not explicitly expose the computational structure required for expertise matching, AlgoSAGE first employs Folding to convert a target problem P into the formal representation $\widehat{\mathcal{M}}_P$ introduced in Section 3.1. The resulting representation makes the algorithm-relevant characteristics of the problem explicit and serves as the input to Retrieval.

Retrieval. Since the formal representation itself does not specify how the problem should be solved, AlgoSAGE uses Retrieval to connect $\widehat{\mathcal{M}}_P$ with the Algorithmic Expertise Memory. Retrieval transforms the structural query into relevant Algorithmic Knowledge and Algorithmic Application Exemplars, which provide the algorithmic evidence used by the subsequent Planning stage.

Planning. Since retrieved knowledge and exemplars describe reusable methods and previous problem instances rather than a solution directly applicable to the target problem, AlgoSAGE employs Planning to adapt the retrieved expertise into a target-specific solution plan. For each planning trajectory, Planning takes the original problem together with one retrieval context, consisting of an Application Exemplar and its associated Algorithmic Knowledge, and produces a complete solution plan specifying the algorithmic procedure, maintained states, required operations, complexity, and implementation considerations. Different retrieval contexts are processed independently, allowing multiple candidate solution trajectories to be explored.

Coding. Since a solution plan remains a high-level specification and cannot be executed directly, AlgoSAGE employs Coding to translate each generated plan into an executable program. Coding takes the original problem and its corresponding plan as input, implements the specified algorithmic procedure, and passes the resulting program to Evaluation.

Evaluation and Auditing. Since generated code may produce incorrect outputs or contain algorithmic and implementation issues that are not exposed by the available examples, AlgoSAGE evaluates each initial implementation through two complementary criteria: execution checking and Auditing. Execution checking runs the program against the available examples, while Auditing inspects the program together with its plan for potential inconsistencies in algorithmic logic, complexity, boundary handling, and implementation. The initial implementation is accepted only when both checks succeed; otherwise, the observed execution feedback and Auditing results are passed to Debugging.

Debugging. Since a failed initial implementation may still be based on a valid solution plan, AlgoSAGE employs Debugging to repair the current program rather than immediately discarding the entire planning trajectory. Debugging takes the original problem, current plan, failed implementation, execution feedback, and Auditing information as input and produces a revised program. The repaired program is rechecked through execution checking only, without repeating Auditing. If it remains unsuccessful after the predefined debugging budget is exhausted, AlgoSAGE returns to Planning and proceeds with another retrieval context associated with a different Application Exemplar.

4 Experiments

4.1 Benchmarks and Metrics

We evaluate AlgoSAGE on both algorithmic programming benchmarks and function-level code-generation benchmarks.

Our main evaluation focuses on three execution-based algorithmic programming benchmarks: CodeContests, APPS, and xCodeEval. CodeContests contains competitive programming problems collected from online platforms (Li et al., 2022). APPS covers programming tasks ranging from introductory exercises to interview and competition-level problems (Hendrycks et al., 2021). xCodeEval provides execution-based programming problems collected from multiple online platforms and programming environments (Khan et al., 2023). These benchmarks contain a substantial number of problems requiring algorithm selection, data-structure design, and reasoning under explicit computational constraints, which matches the target setting of AlgoSAGE.

We additionally evaluate HumanEval and MBPP as complementary function-level code-generation benchmarks. HumanEval consists of manually designed Python programming tasks with executable unit tests (Chen et al., 2021), while MBPP contains short Python programming problems with corresponding test cases (Austin et al., 2021). These benchmarks mainly evaluate code implementation ability rather than complex algorithm selection, and therefore serve as an additional analysis of the general code-generation capability of AlgoSAGE.

Following previous code-generation studies (Islam et al., 2024; 2025), we use Pass@1 as the primary evaluation metric. For each problem, a generated program is considered successful only if it passes all official test cases provided by the benchmark.

$$\text{Pass@1} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{I} \left[\text{Test}(C_i) = \text{pass} \right]. \tag{5}$$

where C_i denotes the final program returned for problem i.

In addition to correctness, we report average token consumption to analyze the efficiency trade-off of different generation frameworks. Token consumption is measured as the average number of tokens consumed per problem during inference, including all prompts and generated responses. For agent-based methods, the measurement includes all intermediate interactions required by their complete workflows and is reported in thousands of tokens.

4.2 Baselines

We compare AlgoSAGE with representative approaches covering direct generation and recent agent-based code-generation frameworks.

- **Direct Generation.** Directly prompts the backbone model to generate the final program from the problem statement and available examples. It does not introduce explicit retrieval, planning, or iterative repair.
- **MapCoder.** MapCoder decomposes competitive programming into multiple stages, including example retrieval, solution planning, code generation, and debugging through a multi-agent workflow (Islam et al., 2024).
- **CodeSIM.** CodeSIM introduces solution planning and simulation before implementation, using simulation feedback to improve code generation and debugging (Islam et al., 2025).

These baselines represent different paradigms of code generation: direct LLM generation, multi-agent generation with planning and repair, and simulation-based agent workflows. All baseline methods are evaluated under the same backbone models and execution environments whenever applicable.

4.3 Setting

We evaluate AlgoSAGE using three backbone models: Qwen3.8-27B-Coder, Claude Haiku 4.5, and GPT-5.4 mini. These models cover different model families and capability levels, allowing us to examine the robustness of AlgoSAGE across backbone models. For all experiments, different methods use the same benchmark splits, programming language, decoding configuration, execution environment, and evaluation harness. All approaches generate Python 3 programs and do not access hidden benchmark tests during generation. The temperature and top-p values are set to 0 and 0.95, respectively, for all methods. The execution-based evaluation follows the testing framework adopted by CodeSIM. For AlgoSAGE, we set the number of retrieved Algorithmic Knowledge entries and Algorithmic Application Exemplars to 3. The minimum similarity thresholds for knowledge retrieval and exemplar retrieval are set to 0.19 and 0.38, respectively. The maximum planning and debugging attempts are configured as follows. Direct Generation performs a single code generation attempt without explicit planning or debugging. MapCoder follows its original configuration with $k = 3$ retrieved examples and $t = 4$ debugging attempts. CodeSIM allows up to 3 planning attempts and 4 debugging attempts for each plan. AlgoSAGE allows up to 3 planning trajectories, and each trajectory can be refined by up to 2 debugging attempts.

Table 1: Pass@1 (%) and token consumption comparison on algorithmic programming benchmarks. Tokens in the table denote token consumption, i.e., the average number of tokens consumed per problem during inference. Each value is reported as Prompt tokens / Completion tokens, with all numbers measured in thousands of tokens. For agent-based methods, the measurement includes all intermediate interactions required by their complete inference workflows.

Method	CodeContests		APPS		xCodeEval	
	Pass@1	Tokens	Pass@1	Tokens	Pass@1	Tokens
Qwen3.8-27B-Coder						
Direct Generation	38.8	0.69/4.04	46.0	0.67/3.86	65.1	0.55/2.31
MapCoder	56.4	40.15/25.26	51.3	29.16/19.37	67.0	26.39/17.58
CodeSIM	68.5	30.74/32.38	69.3	16.89/19.36	74.5	13.63/14.33
AlgoSAGE	**70.3**	38.07/25.44	**76.0**	27.91/18.28	**78.3**	20.95/13.56
Claude Haiku 4.5						
Direct Generation	27.3	0.75/0.99	42.0	0.71/0.85	49.1	0.57/0.89
MapCoder	44.2	35.03/15.16	59.3	20.86/9.85	57.5	18.79/9.73
CodeSIM	46.1	25.21/15.09	61.3	13.16/8.16	58.5	11.03/7.25
AlgoSAGE	**48.5**	27.08/10.91	**62.7**	18.27/7.30	**62.3**	16.26/6.96
GPT-5.4 mini						
Direct Generation	27.9	0.65/0.81	22.7	0.63/0.69	43.4	0.50/0.53
MapCoder	46.7	24.86/9.38	47.3	19.48/7.12	52.8	14.38/5.78
CodeSIM	48.5	16.67/10.63	48.7	10.30/6.50	59.4	7.66/5.46
AlgoSAGE	**52.7**	24.51/7.38	**49.3**	19.13/5.24	**60.4**	13.87/3.94

Table 2: Results on function-level benchmarks using Claude Haiku 4.5. Pass@1 (%) is reported for correctness evaluation. Tokens denotes the average prompt and completion tokens per problem, measured in thousands.

Method	HumanEval		MBPP	
	Pass@1	Tokens	Pass@1	Tokens
Direct Generation	51.8	0.214/1.580	43.1	0.118/1.401
MapCoder	78.7	20.093/5.925	70.5	23.144/5.534
CodeSIM	97.6	3.763/4.705	88.9	4.409/4.075
AlgoSAGE	**98.2**	7.686/5.040	**90.9**	7.699/4.364

4.4 Results and Analysis

Overall performance across backbone models. Table 1 reports both Pass@1 and average token consumption on algorithmic programming benchmarks. AlgoSAGE consistently improves execution correctness across different backbone models while maintaining a comparable token consumption with existing agent-based approaches. With Qwen3.8-27B-Coder, AlgoSAGE achieves the best performance on all three benchmarks, obtaining 70.3%, 76.0%, and 78.3% Pass@1 on CodeContests, APPS, and xCodeEval, respectively. Compared with the strongest baseline CodeSIM, AlgoSAGE improves Pass@1 by 1.8, 6.7, and 3.8 percentage points. Similar improvements are observed with Claude Haiku 4.5 and GPT-5.4 mini, where AlgoSAGE consistently outperforms existing approaches on all benchmarks. These results demonstrate that the effectiveness of AlgoSAGE is not tied to a specific backbone model, but comes from the additional algorithmic guidance provided by structured problem representation and expertise retrieval. Moreover, the improvements are not obtained simply by increasing generation length: despite introducing retrieval and planning stages, AlgoSAGE maintains a comparable token consumption with other agent-based methods and avoids the repeated simulation trajectories used by simulation-based approaches such as CodeSIM.

Generalization to function-level programming tasks. Table 2 evaluates AlgoSAGE on HumanEval and MBPP. Although these benchmarks mainly focus on function implementation rather than complex algorithm selection, AlgoSAGE still achieves the best performance among all com-

Table 3: Comparison between Direct Generation and AlgoSAGE using GPT-6 Astra on algorithmic programming benchmarks. Results are reported as Pass@1 (%).

Method	CodeContests	APPS	xCodeEval
Direct Generation	72.67	63.58	83.96
AlgoSAGE	**90.00**	**79.01**	**89.62**

Table 4: Ablation study of AlgoSAGE. Results are reported as Pass@1 (%).

Method	CodeContests	APPS	xCodeEval
w/o Audit	66.1	69.3	74.5
w/o Folding	67.3	70.0	72.6
w/o Skill	67.9	72.0	74.5
w/o Example	63.6	68.0	72.6
AlgoSAGE	**70.3**	**76.0**	**78.3**

pared methods. The results show that introducing expertise retrieval and structured planning does not harm general code-generation ability. Meanwhile, the larger gains on algorithmic benchmarks indicate that the proposed framework is particularly effective when successful generation requires selecting and adapting appropriate algorithms.

Evaluation with stronger foundation models. To examine whether stronger foundation models can already solve algorithmic programming problems through direct generation, we further evaluate GPT-6 Astra with Direct Generation and AlgoSAGE. Due to the higher inference cost, we only compare these two settings on the main algorithmic benchmarks. Table 3 shows that GPT-6 Astra achieves strong performance with direct generation but remains below complete correctness on the evaluated benchmarks, indicating that challenging algorithmic programming problems are not fully solved by stronger foundation models alone. AlgoSAGE further improves the results across all benchmarks, demonstrating that expertise-grounded retrieval and structured planning provide additional guidance beyond the capability of the underlying model, even with a stronger backbone.

4.5 Ablation Study

We conduct ablation studies using Qwen3.8-27B-Coder on CodeContests, APPS, and xCodeEval. We remove each component from the full system to examine the contribution of Auditing, formal problem representation, and the two components of Algorithmic Expertise Memory. Table 4 shows that all components contribute to the final performance of AlgoSAGE. Removing Folding consistently decreases performance, especially on xCodeEval, demonstrating that structural problem representation provides more reliable expertise retrieval signals than direct textual matching. Removing either Algorithmic Knowledge or Algorithmic Application Exemplars also reduces performance. Algorithmic Knowledge provides reusable algorithmic principles for selecting solution strategies, while Application Exemplars provide concrete demonstrations of how these principles are instantiated in specific problems. Removing Auditing decreases performance across all benchmarks, indicating that plan-aware verification provides complementary feedback beyond execution checking and improves the reliability of subsequent debugging. Overall, the ablation results confirm that structured representation, expertise retrieval, and reliability-oriented evaluation jointly contribute to the effectiveness of AlgoSAGE.

5 Conclusion and Limitation

We propose AlgoSAGE, an expertise-grounded framework for algorithmic code generation that combines structured problem representation with reusable algorithmic knowledge. Experiments across multiple benchmarks and backbone models demonstrate consistent improvements over existing generation and agent-based approaches.A limitation of AlgoSAGE is that its effectiveness depends on the coverage and quality of the Algorithmic Expertise Memory. Future work will explore more adaptive expertise acquisition and extension to broader software engineering tasks.

AI USE STATEMENT

In this work, we used generative AI tools only to assist with language polishing and translation during manuscript preparation. We did not use generative AI tools for research ideation, experiment design or execution, related work discovery, drafting of paper sections, synthetic data generation, mathematical proof, or code generation. All AI-assisted text was carefully reviewed and revised by the authors, who take full responsibility for the final content, including all claims, analyses, and experimental results.

ETHICS STATEMENT

This work studies automated code generation using publicly available programming benchmarks and language models. It does not involve human subjects, private user data, or the collection of sensitive information. We follow responsible research practices by reporting experimental settings, evaluation procedures, and limitations to facilitate transparent assessment of the proposed approach.

REPRODUCIBILITY STATEMENT

We provide an anonymous GitHub repository containing the implementation, experimental scripts, and configuration files required to reproduce our results. The paper and appendix describe the proposed framework, Algorithmic Expertise Memory construction, retrieval procedure, experimental settings, and evaluation protocols. We use publicly available benchmarks and report all model configurations and hyperparameters necessary for reproduction.

REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Codeforces. 1511F: Chainword. https://codeforces.com/contest/1511/problem/F, n.d.a.

Codeforces. 185A: Plant. https://codeforces.com/problemset/problem/185/A, n.d.b.

Duy Tung Doan, Quang Huy Phung, Dzung Nguyen, and Khac-Hoai Nam Bui. CollabCoder: Plan-code co-evolution via collaborative decision-making for efficient code generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 27964–27985, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1393. URL https://aclanthology.org/2026.findings-acl.1393/.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 33(7), September 2024. ISSN 1049-331X. doi: 10.1145/3672459. URL https://doi.org/10.1145/3672459.

Sama Hadhoud, Alaa Elsetohy, Frederikus Hudi, Jan Christian Blaise Cruz, Steven Halim, and Alham Fikri Aji. Idea first, code later: Disentangling problem solving from code generation in evaluating LLMs for competitive programming. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 35708–35747, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1778. URL https://aclanthology.org/2026.findings-acl.1778/.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 4912–4944, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.269. URL https://aclanthology.org/2024.acl-long.269/.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. CodeSim: Multi-agent code generation and problem solving through simulation-driven planning and debugging. In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 5128–5154, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.285. URL https://aclanthology.org/2025.findings-naacl.285/.

Geonhui Jang, Dongyoon Han, and YoungJoon Yoo. StoryCoder: Narrative reformulation for structured reasoning in LLM code generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 19092–19112, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.872. URL https://aclanthology.org/2026.acl-long.872/.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30, 2024.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023. URL https://arxiv.org/abs/2303.03004.

Chao Lei, Yanchuan Chang, Nir Lipovetzky, and Krista A Ehinger. Planning-driven programming: A large language model programming workflow. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12647–12684, 2025.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

James Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15(1):122–127, 1969.

Tal Ridnik, Dedy Kredo, and Itamar Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

Xinyuan Song, Zekun Cai, and Liang Zhao. Algoskill: Learning to design algorithms by scheduling human-like skills, 2026. URL https://arxiv.org/abs/2606.29999.

Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 2081–2107, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-acl.124. URL https://aclanthology.org/2024.findings-acl.124/.

Zimu Wang, Yuling Shi, Mengfan Li, Zijun Liu, Jie M. Zhang, Chengcheng Wan, and Xiaodong Gu. Effiskill: Agent skill based automated code efficiency optimization, 2026. URL https://arxiv.org/abs/2603.27850.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. CodeRAG-bench: Can retrieval augment code generation? In Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3199–3214, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.176. URL https://aclanthology.org/2025.findings-naacl.176/.

ZhenChun Xu, Yi Cai, Dajun Zheng, li Yuan, Mengchen Zhao, Qixiang Wang, and Jiexin Wang. MavenCoder: Competitive code generation via model adaptive planning strategies and multi-perspective verification enhancement. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 30415–30439, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1403. URL https://aclanthology.org/2026.acl-long.1403/.

Jaeseok Yoo, Hojae Han, Youngwon Lee, Jaejin Kim, and Seung-won Hwang. PERC: Plan-as-query example retrieval for underrepresented code generation. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert (eds.), *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 7982–7997, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.534/.

Youngsik Yoon, Sungjae Lee, Seockbean Song, Siwei Wang, Wei Chen, and Jungseul Ok. PaT: Planning-after-trial for efficient test-time code generation. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 36738–36755, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.1703. URL https://aclanthology.org/2026.acl-long.1703/.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023.

Shiwen Zhang, Lingxiang Wang, Hainan Zhang, Ziwei Wang, Sijia Wen, and Zhiming Zheng. Beyond the surface: A solution-aware retrieval model for competition-level code generation. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5237–5246, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.281. URL https://aclanthology.org/2025.findings-emnlp.281/.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870, 2024.

Yueheng Zhu, Chao Liu, Xuan He, Xiaoxue Ren, Zhongxin Liu, Ruwei Pan, and Hongyu Zhang. Adacoder: An adaptive planning and multi-agent framework for function-level code generation. *IEEE Transactions on Software Engineering*, 52:631–650, 2025. URL https://api.semanticscholar.org/CorpusID:277621741.

A Motivating Case Study: Chainword

A.1 Problem and Assessment Criteria

A.1.1 Original Problem Statement

Chainword originates from Codeforces problem 1511F (Codeforces, n.d.a). The following statement, input/output specifications, and examples reproduce the experimental dataset. Mathematical delimiters have been adapted for typesetting. The illustration referenced in the original notes is absent from the dataset.

Chainword

Time limit: 3 seconds
Memory limit: 256 megabytes
Input: standard input
Output: standard output

A chainword is a special type of crossword. As most of the crosswords do, it has cells that you put the letters in and some sort of hints to what these letters should be.The letter cells in a chainword are put in a single row. We will consider chainwords of length m in this task.A hint to a chainword is a sequence of segments such that the segments don't intersect with each other and cover all m letter cells. Each segment contains a description of the word in the corresponding cells.The twist is that there are actually two hints: one sequence is the row above the letter cells and the other sequence is the row below the letter cells. When the sequences are different, they provide a way to resolve the ambiguity in the answers.You are provided with a dictionary of n words, each word consists of lowercase Latin letters. All words are pairwise distinct.An instance of a chainword is the following triple: a string of m lowercase Latin letters; the first hint: a sequence of segments such that the letters that correspond to each segment spell a word from the dictionary; the second hint: another sequence of segments such that the letters that correspond to each segment spell a word from the dictionary. Note that the sequences of segments don't necessarily have to be distinct.Two instances of chainwords are considered different if they have different strings, different first hints or different second hints.Count the number of different instances of chainwords. Since the number might be pretty large, output it modulo $998\,244\,353$.

Input

The first line contains two integers n and m ($1 \le n \le 8$, $1 \le m \le 10^9$) — the number of words in the dictionary and the number of letter cells. Each of the next n lines contains a word — a non-empty string of no more than 5 lowercase Latin letters. All words are pairwise distinct.

Output

Print a single integer — the number of different instances of chainwords of length m for the given dictionary modulo $998\,244\,353$.

Examples

Example	Input	Output
Example 1	3 5 ababa ab a	11
Example 2	2 4 ab cd	4
Example 3	5 100 a aa aaa aaaa aaaaa	142528942

Table 5: Sample inputs and outputs for Chainword.

Notes

Here are all the instances of the valid chainwords for the first example: The red lines above the letters denote the segments of the first hint, the blue lines below the letters denote the segments of the second hint.In the second example the possible strings are: ”abab”, ”abcd”, ”cdab” and ”cdcd”. All the hints are segments that cover the first two letters and the last two letters.

A.1.2 Counting Objective and Scale Constraints

Let D be the dictionary, and let $f_D(s)$ denote the number of segmentations of a string s into words from D. Each Chainword instance consists of a string and an ordered pair of segmentations of that same string. The required answer is therefore

$$\mathrm{Ans}(D, m) = \left(\sum_{s \in \Sigma^m} f_D(s)^2 \right) \bmod P, \qquad P = 998244353, \tag{6}$$

where Σ is the lowercase Latin alphabet. Strings without a valid segmentation contribute zero. The same-string requirement is essential: squaring the total number of segmentations would also count pairs that produce different strings.

The input constraints create a second requirement beyond correct counting semantics. The dictionary contains at most $n = 8$ words of maximum length $\ell = 5$, whereas the target length can be as large as $m = 10^9$. Under the stated time and memory limits, a practical solution must exploit the small dictionary and short words instead of processing every target position or enumerating generated strings. For example, the valid dictionary $D = \{a, b\}$ generates 2^m distinct strings of length m.

The central modeling challenge is thus to preserve compatibility between two segmentations while aggregating generation histories into a representation whose size does not grow with the target length.

A.1.3 Assessment Criteria

We assess both traces against four requirements derived from the counting objective and input constraints. These criteria do not prescribe a particular implementation.

Counting semantics. Count ordered segmentation pairs that produce the same string, as required by Equation 6.

State sufficiency. Retain enough information to determine compatible extensions while preserving the multiplicity of distinct segmentation choices.

Boundary handling. Define initialization, word reselection, and answer extraction so that both segmentations end exactly at length m.

Computational feasibility. Account for state size and evaluation cost under the full constraints. Naming a fast evaluation technique is insufficient unless it is applied to a suitably bounded state representation.

A.2 Experimental Setting and Trace Overview

We analyze the recorded trajectories of AlgoSAGE and CodeSIM on Chainword, following planning, code generation, and debugging through final evaluation. In this case study, AlgoSAGE uses Qwen3.8-27B-Coder with at most three planning attempts and two debugging attempts per plan, whereas CodeSIM uses Claude Haiku 4.5 with at most three planning attempts and four debugging attempts per plan. Both runs use a temperature of 0 and a top-p value of 0.95. These settings describe the recorded case-study runs; the different backbones and debugging budgets preclude a controlled performance comparison.

A.3 Retrieval and Target Adaptation

A.3.1 CodeSIM: Semantic Repair without State Compression

What the recalled problem provides. CodeSIM recalls Word Break Partitioning, which counts the valid segmentations of a given string using a dictionary. This reference is relevant to dictionary segmentation, but its input fixes the character sequence in advance.

For an explicit input string s of length L, a prefix dynamic program can aggregate segmentations reaching the same position because they refer to the same input prefix. The logged reference implementation checks n words at each position. Accounting for substring slicing and comparison, its time complexity is $O(Ln\ell)$ and its auxiliary space is $O(L)$.

Chainword does not provide an explicit string. It asks for a count over all length-m strings and their compatible segmentation pairs. Consequently, prefixes of equal length cannot be treated as the same character prefix.

The initial semantic mismatch. CodeSIM's initial plan counts concatenation sequences by total length and squares the resulting count. This replaces

$$\sum_{s\in\Sigma^m} f_D(s)^2 \quad \text{with} \quad \left(\sum_{s\in\Sigma^m} f_D(s)\right)^2.$$

The second expression includes pairs of segmentations producing different strings.

For example, with $D=\{a,b\}$ and $m=1$, the correct answer is $1^2+1^2=2$, whereas squaring the total gives $(1+1)^2=4$. This constructed example isolates the information lost by aggregating only by length.

The repair and its computational cost. The recorded simulation identifies the incorrect square and requests a revision. CodeSIM's final program restores the same-string requirement by storing a separate segmentation count for each generated string and then summing the squared counts. The following noncontiguous excerpts show this representation:

```
dp = [defaultdict(int) for _ in range(m + 1)]
dp[0][] = 1

for i in range(m + 1):
    if not dp[i]:
        continue
    for string, count in list(dp[i].items()):
        for length in lengths:
            if i + length <= m:
                for word in words_by_len[length]:
                    new_string = string + word
```

```
                    dp[i + length][new_string] = (
                        dp[i + length][new_string] + count
                    ) % MOD

answer = 0
for string, partition_count in dp[m].items():
    answer = (answer + partition_count * partition_count) % MOD
```

This revision repairs the counting semantics, but its state space grows with the number of generated strings. For $D = \{a, b\}$, it stores 2^i distinct strings at length i, giving

$$\sum_{i=0}^{m} 2^i = 2^{m+1} - 1$$

string states across all lengths. State enumeration alone therefore requires $\Omega(2^m)$ time. Including the construction and hashing of length-i strings, the string-processing work in this implementation is

$$\Theta\left(\sum_{i=1}^{m} i\, 2^i\right) = \Theta(m\, 2^m).$$

The length-indexed array also retains a direct dependence on m.

The unresolved adaptation problem is therefore state compression: the program preserves string identity explicitly instead of retaining only the local information needed to extend a compatible pair of segmentations. Although the trace mentions matrix exponentiation, the final implementation does not apply fast evaluation to a bounded counting system.

A.3.2 AlgoSAGE: From a Retrieved Framework to Joint-State Modeling

The query and retrieved context. AlgoSAGE constructs an eight-field representation of the target problem to guide expertise retrieval. The representation encodes structural properties and inferred candidate computational mechanisms, while leaving the target-specific state definitions and transition equations to the subsequent planning stage.

In the recorded run, Plan 1 is paired with the knowledge-base exemplar *185A: Plant* (Codeforces, n.d.b) and the Algorithmic Knowledge entry on matrix exponentiation. The retrieved context provides state definitions, recurrences, solution steps, and complexity analysis for the reference problem, together with reusable guidance on the computational method. It supplies a framework for adaptation rather than a ready-made solution to Chainword.

As described in that context, Plant aggregates triangles by orientation. Writing U_t and D_t for the two orientation counts, its evolution is

$$\begin{pmatrix} U_{t+1} \\ D_{t+1} \end{pmatrix} = \begin{pmatrix} 3 & 1 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} U_t \\ D_t \end{pmatrix}, \qquad \begin{pmatrix} U_0 \\ D_0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

A fixed two-dimensional transition permits evaluation after a large number of steps by matrix exponentiation.

The transferable element is the organization of the computation: aggregate objects with identical future behavior, accumulate their one-step contributions through a fixed linear transition, and evaluate the resulting process efficiently. The reference does not supply Chainword's states or compatibility conditions.

Constructing target-specific joint states. AlgoSAGE's target plan introduces states (u, p, v, q), where u and v identify the current words in the two segmentations and p and q record their progress. Under the consumed-character convention,

$$1 \le p \le |w_u|, \qquad 1 \le q \le |w_v|.$$

Each step appends one character on both sides. An unfinished word supplies its next character; a completed word permits selection of a new dictionary word. A transition is allowed only when the two emitted characters agree.

Initialization selects an ordered pair of words with matching first characters and consumes that character. After m characters, only states in which both current words have ended contribute to the answer. The final program includes the following noncontiguous excerpts:

```
if words[u][0] == words[v][0]:
    s = (u, 1, v, 1)
    # ... omitted ...
# ... omitted ...
if p == len(words[u]) and q == len(words[v]):
    terminal_indices.append(i)
```

These rules maintain agreement incrementally. Each complete path represents an ordered pair of segmentations of one generated string. Different paths reaching the same state can be added because their legal future extensions depend only on the state and dictionary. Their multiplicities remain in the state count; complete strings do not need to remain as keys.Each aggregated history already represents a compatible pair of partial segmentations. The algorithm adds these pair counts; it never squares an aggregate of single-segmentation counts from different strings.

Let K denote the number of joint states retained in the transition system. It satisfies

$$K \le \left(\sum_{w \in D} |w| \right)^2 \le (n\ell)^2 \le 1600. \tag{7}$$

Removing unreachable or incompatible states can further reduce K. This upper bound is independent of m.

Adapting the evaluation method. Let x_1 be the joint-state count vector after initialization, which consumes the first character, and let c indicate states where both current words have ended. For a fixed dictionary, let $A_{j,i}$ count the legal one-character transitions from state i to state j. Then, over $\mathbb{F}_P$,

$$x_{t+1} = Ax_t \quad (t \ge 1), \qquad a_t = c^\top x_t, \qquad \mathrm{Ans}(D, m) = c^\top A^{m-1} x_1.$$

Transition entries retain the multiplicity of legal choices.

This formulation connects the target model to the retrieved framework, but does not by itself establish practical feasibility. Dense matrix powering costs $O(K^3 \log m)$ time and $O(K^2)$ space, which is too expensive to serve as a satisfactory guarantee at the largest candidate state bound.

The selected program generates initial terms using finite-state transitions, recovers a linear recurrence over $\mathbb{F}_P$ using the Berlekamp–Massey algorithm (Massey, 1969), and evaluates the target term through polynomial reduction and binary exponentiation. By the Cayley–Hamilton theorem, the sequence $b_r = c^\top A^r x_1$ for $r \ge 0$ satisfies a linear recurrence of order at most K. Its first $2K$ terms suffice for recurrence recovery by Berlekamp–Massey over $\mathbb{F}_P$, after which the required answer is obtained as b_{m-1}.

Let E be the number of stored transitions and $d \le K$ the recovered recurrence order. After state-graph construction, generating $O(K)$ terms, recovering the recurrence, and evaluating the target term has the standard bound

$$O(KE + K^2 + d^2 \log m) \quad \text{time}, \qquad O(K + E + d) \quad \text{space}, \tag{8}$$

using sparse transitions and linear-size polynomial work arrays.The time bound counts arithmetic operations over the fixed field $\mathbb{F}_P$ and excludes state-graph construction.

This removes iteration over all m positions, although actual runtime still depends on the reachable state count, transition density, recurrence order, and implementation constants. The bound alone does not certify the three-second limit.

The recorded audit also flags an indexing issue in recurrence evaluation within `combine`, followed by a debug artifact. Thus, the trace documents three distinct steps: retrieval of a linear counting framework, construction of a target-specific joint-state model, and implementation-level correction of its fast evaluation.

A.4 What the Evaluation Establishes

CodeSIM's public-sample success is compatible with the complexity defect above. The first two examples have target lengths 5 and 4. The third has length 100, but its dictionary $D = \{a, aa, aaa, aaaa, aaaaa\}$ produces only one distinct string at each length. A string-keyed dynamic program can aggregate that string's many segmentations without encountering the exponential number of strings generated by $D = \{a, b\}$.

The samples therefore exercise counting behavior without excluding worst-case state growth. The trace shows a successful local correction of counting semantics, followed by a representation that remains incompatible with the full constraints. Static analysis establishes this incompatibility, but failure in the final evaluation does not by itself identify a timeout as the specific failure reason.

AlgoSAGE's saved successful outcome applies to the selected debugged program as a whole. It is consistent with the joint-state model and fast recurrence evaluation, but cannot isolate the contribution of the retrieved reference from planning, code generation, or debugging.

The evaluation evidence therefore supports two separate observations: sample agreement does not establish computational feasibility, and final program success does not establish the causal contribution of an individual pipeline component.

A.5 Discussion and Limitations

Chainword illustrates why retrieval relevance must be assessed relative to both the counting objective and the target constraints. Word Break shares the dictionary-segmentation setting, but its fixed input string makes prefix aggregation possible in a way that does not transfer directly to Chainword. Plant has a different narrative, yet its finite-state linear evolution provides a computational framework compatible with large-step evaluation.

The AlgoSAGE trace records a connection between that retrieved framework and subsequent target modeling: the reference is bound to the first plan, joint states enforce character compatibility, and fast recurrence evaluation handles the large target length. This is qualitative evidence of framework adaptation in the recorded run, rather than proof that retrieval caused the successful outcome.

The comparison is limited by the different backbones and debugging budgets, and by its focus on a single problem and recorded trajectories. Moreover, the structured representation already contains algorithm-relevant labels, so its role cannot be separated from algorithm inference in this case. Same-backbone comparisons, matched budgets, and controlled ablations are needed to assess the independent contributions of representation design, retrieval, and target planning.

B Extended Related Work

This appendix supplements Section 2 with additional discussion of explicit planning, verification and debugging, and reusable algorithmic expertise. It focuses on how these approaches organize intermediate solution structure, use feedback, and provide evidence for target-specific planning.

B.1 Explicit Planning and Task Decomposition

Self-planning separates solution planning from code implementation: the model first generates solution steps and then uses them to guide program generation (Jiang et al., 2024). Parsel represents algorithmic tasks through hierarchical natural-language function descriptions and searches for combinations of implementations that satisfy tests (Zelikman et al., 2023). Both approaches make intermediate solution structure explicit to support reasoning and implementation.

AlgoSAGE additionally uses a structured representation of algorithm-relevant problem features to retrieve reusable expertise before constructing a target-specific plan. This representation provides a shared basis for matching the target problem with Algorithmic Knowledge and Application Exemplars. The retrieved evidence then supports the development of concrete states, operations, and complexity arguments during planning.

B.2 Plan Verification and Execution-Based Debugging

Planning-Driven Programming verifies solution plans using visible tests and uses the resulting guidance during implementation and refinement (Lei et al., 2025). This work is closely related to the use of plans as explicit artifacts for checking and correction. AlgoSAGE complements this direction by retrieving algorithmic knowledge and associated application exemplars to inform target-specific planning, and by auditing the generated implementation together with its plan.

Other approaches emphasize feedback from generated programs. AlphaCodium develops a test-driven, multi-stage code-generation workflow (Ridnik et al., 2024). LDB examines intermediate runtime states to localize errors and guide debugging (Zhong et al., 2024). These methods provide complementary mechanisms for identifying and correcting program errors.

Passing visible tests provides evidence about the tested inputs, but does not by itself establish that the chosen algorithm meets the full input constraints. The Chainword trace in Appendix A illustrates this distinction: the revised program corrects the counting semantics but explicitly enumerates generated strings, leaving computational feasibility unresolved. AlgoSAGE combines execution feedback with auditing of algorithmic logic, complexity, boundary conditions, and consistency between the plan and its implementation. Auditing is applied to the initial implementation of each planning trajectory; debugged programs are rechecked through execution tests. Such auditing remains an LLM-based assessment rather than a formal correctness proof.

B.3 Algorithm-Aware Retrieval and Reusable Expertise

Algorithm-aware retrieval uses solution-related information to identify relevant references. PERC uses algorithmic plans as retrieval queries, while SolveRank learns solution-aware representations for retrieving relevant problems (Yoo et al., 2025; Zhang et al., 2025). AlgoSAGE shares this emphasis on computational relevance and organizes retrieval around two linked types of evidence: reusable Algorithmic Knowledge and concrete Application Exemplars.

A shared representation schema supports matching between the target problem and memory entries. Retrieved knowledge entries restrict the exemplar candidate pool through explicit knowledge links. Each planning attempt receives one selected exemplar, when available, and at most two retrieved knowledge entries referenced by that exemplar. The planning context contains the exemplar's summary and stored plan together with the names and templates of the paired knowledge entries. Appendix C specifies the filtering, ranking, and binding rules.

Reusable expertise can also be organized as skills. EffiSkill explores skills for improving the efficiency of existing programs, while AlgoSkill studies algorithm design through verification-guided skill scheduling (Wang et al., 2026; Song et al., 2026). The distinguishing design of AlgoSAGE is its use of linked knowledge and exemplar retrieval to support solution construction from a target problem statement. The knowledge provides reusable algorithmic guidance, while the exemplar illustrates a concrete application that must be assessed and adapted during planning.

B.4 Comparison with Representative Approaches

Table 6 summarizes the principal mechanisms of representative approaches and their relation to AlgoSAGE. The comparison describes methodological emphases rather than controlled performance differences.

Table 6: Comparison of representative approaches. The entries summarize methodological differences rather than controlled performance comparisons.

Method	Principal mechanism	Relation to AlgoSAGE
MapCoder (Islam et al., 2024)	Example recall followed by planning, coding, and debugging.	AlgoSAGE retrieves evidence from an explicit memory containing linked algorithmic knowledge and application exemplars.
CodeSIM (Islam et al., 2025)	Simulation-driven planning and debugging.	AlgoSAGE grounds planning in structurally retrieved expertise and audits the implementation and its associated plan.
Planning-Driven Programming (Lei et al., 2025)	Visible-test verification of plans guides implementation and refinement.	AlgoSAGE additionally retrieves algorithmic knowledge and associated exemplars to support target-specific planning.
PERC (Yoo et al., 2025)	Algorithmic plans serve as queries for example retrieval.	AlgoSAGE uses a shared field schema and knowledge-conditioned exemplar selection.
SolveRank (Zhang et al., 2025)	Solution-aware representations support retrieval of problems with related algorithmic logic.	AlgoSAGE uses explicit structural matching over linked knowledge and exemplar entries.
EffiSkill (Wang et al., 2026)	Reusable optimization skills support code-efficiency improvement.	AlgoSAGE retrieves expertise for constructing solutions from problem statements.
AlgoSkill (Song et al., 2026)	Verification-guided skill scheduling supports algorithm design.	AlgoSAGE retrieves linked knowledge and exemplars for target-specific planning.

C Structural Similarity and Hierarchical Retrieval

AlgoSAGE first retrieves Algorithmic Knowledge and then selects Application Exemplars linked to the retrieved knowledge. Both stages use the eight-field representation described in Appendix D.

C.1 Weighted Structural Similarity

Let $\mathcal{F}$ denote the eight representation fields and $\mathcal{B}$ the retrieval corpus for the current stage. Let Q_f and C_f denote the normalized query and candidate label sets for field $f \in \mathcal{F}$. Structural similarity is computed solely from these label sets. Labels match only when they are identical and belong to the same field. Their weights combine the field coefficients and corpus-specific IDF values defined in Appendix D:

$$\omega_{\mathcal{B},f}(\lambda) = w_f \ \mathrm{idf}_{\mathcal{B},f}(\lambda).$$

IDF statistics are computed separately for the loaded knowledge and exemplar collections, before query-specific filtering.

Define

$$W_{\mathcal{B}}(Q) = \sum_{f\in\mathcal{F}} \sum_{\lambda\in Q_f} \omega_{\mathcal{B},f}(\lambda), \qquad I_{\mathcal{B}}(Q,C) = \sum_{f\in\mathcal{F}} \sum_{\lambda\in Q_f\cap C_f} \omega_{\mathcal{B},f}(\lambda).$$

Here, $W_{\mathcal{B}}(C)$ is defined analogously to $W_{\mathcal{B}}(Q)$. The structural similarity is the weighted Dice overlap:

$$\mathrm{Sim}_{\mathcal{B}}(Q,C) = \begin{cases} \dfrac{2I_{\mathcal{B}}(Q,C)}{W_{\mathcal{B}}(Q)+W_{\mathcal{B}}(C)}, & W_{\mathcal{B}}(Q)+W_{\mathcal{B}}(C) > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

The score lies in $[0,1]$ and balances coverage of query labels against unmatched candidate labels.

C.2 Eligibility and Hierarchical Retrieval

Structural eligibility. Candidates must satisfy three checks: (i) at least one shared entity label when both representations contain entity labels; (ii) at least one shared label across Decision Variables, Relations, Transition Rules, and Objective, considered collectively, when the query contains any labels in these fields; and (iii) at least one matching distinctive label when the query contains labels from the implementation's predefined distinctive set, covering mechanisms such as flow, linear

recurrences, and digit dynamic programming. All matches require identical labels within the same field.

Application Exemplars additionally require at least one shared label across State Space, Relations, Transition Rules, and Objective, considered collectively. These checks exclude weak structural matches but do not establish target-specific correctness or feasibility.

Knowledge retrieval. Only published knowledge entries enabled for retrieval are considered. Candidates passing the structural eligibility checks and the similarity threshold $\tau_K = 0.19$ are ranked in descending order of structural similarity, and at most $K_{\text{skill}} = 3$ entries are retained.

Exemplar retrieval. Let $\mathcal{K}_Q$ be the selected knowledge names and $L(e)$ the knowledge references attached to exemplar e. The candidate pool is restricted to exemplars satisfying

$$L(e) \cap \mathcal{K}_Q \neq \varnothing.$$

Each exemplar must be published and contain a problem identifier, knowledge references, a representation, and a usable stored plan. Candidates must pass the structural checks and similarity threshold $\tau_E = 0.38$.

Exemplars are ranked by structural similarity. Exact ties are resolved by lexical Dice overlap between token sets of the target and exemplar problem statements. This textual score does not contribute to the structural threshold. No exemplars are retrieved when $\mathcal{K}_Q$ is empty.

C.3 Exemplar Selection and Plan Binding

At most $k_E = 3$ exemplars are selected. Let $\mathcal{U}$ denote the union of knowledge references attached to selected exemplars, initialized as $\mathcal{U} = \varnothing$. If the candidate pool is nonempty, the highest-ranked exemplar is selected first. Each subsequent selection takes the highest-ranked remaining exemplar satisfying

$$L(e) \cap \mathcal{U} = \varnothing,$$

when available, and otherwise takes the highest-ranked remaining candidate. After each selection, $\mathcal{U} \leftarrow \mathcal{U} \cup L(e)$. Selection stops when k_E exemplars have been selected or no candidates remain. This heuristic favors disjoint knowledge-reference sets without guaranteeing coverage of every retrieved knowledge entry.

The i-th planning attempt is paired with the i-th selected exemplar and up to two retrieved knowledge entries referenced by that exemplar, preserving their retrieval order. The context contains the exemplar's problem summary and stored plan together with the corresponding knowledge guidance. If no exemplar is available for that attempt, the binding routine uses the highest-ranked retrieved knowledge entry, when available.

Retrieval therefore supplies a compact reference context, while planning establishes the target-specific states, transitions, boundary conditions, and computational complexity.

D Formal Representation Specification

This section specifies the eight-field formal representation used in AlgoSAGE, including its design motivation, field semantics, label conventions, and weighting scheme.

D.1 Design Motivation

The formal representation captures algorithm-relevant problem features and may include inferred computational mechanisms, but does not constitute a complete solution plan. It provides a shared basis for matching a target problem with reusable algorithmic expertise.

For example, graph problems may require different methods depending on their objectives, structural properties, and input constraints: shortest-path queries and tree-structured optimization problems can call for different algorithmic strategies. Effective expertise retrieval therefore requires representations that capture objects, relations, operations, and constraints rather than textual similarity alone.

The representation follows three design principles:

- **Algorithm relevance.** Fields should capture information that influences algorithm applicability, such as structural relations, state transitions, and computational constraints.
- **Cross-problem comparability.** Different problems should be represented in a shared space to enable matching with reusable Algorithmic Knowledge and Application Exemplars.
- **Separation from complete solution specification.** The representation may include candidate algorithmic mechanisms, while concrete state definitions, transition equations, boundary conditions, and implementation details are established during planning.

D.2 Eight-Field Representation

For a programming problem P, its formal representation is defined as

$$\mathcal{M}_P = \big(\mathcal{E}_P, \mathcal{X}_P, \mathcal{V}_P, \mathcal{R}_P, \mathcal{C}_P, \mathcal{T}_P, \mathcal{O}_P, \mathcal{A}_P\big), \tag{10}$$

where $\mathcal{E}_P$, $\mathcal{X}_P$, $\mathcal{V}_P$, $\mathcal{R}_P$, $\mathcal{C}_P$, $\mathcal{T}_P$, $\mathcal{O}_P$, and $\mathcal{A}_P$ correspond to Entities, State Space, Decision Variables, Relations, Constraints, Transition Rules, Objective, and Answer Extraction, respectively.

Each field is represented by a list of controlled labels, interpreted as a set for structural matching:

$$\mathcal{M}_P^{(f)} = \tau_P^{(f)}, \qquad f \in \mathcal{F}.$$

We denote the representation after label filtering, deduplication, and redundant-parent removal by $\widehat{\mathcal{M}}_P$. Structural retrieval uses its normalized label sets. Table 7 details the meaning of each field and its role in retrieval.

Field	Meaning	Role in Retrieval
Entities	Primary computational objects involved in the problem.	Provides object-level signals for identifying relevant algorithm families.
State Space	Candidate categories of information used to summarize partial solutions or computational configurations.	Indicates possible forms of state organization, such as prefix states or dynamic programming states.
Decision Variables	Choices whose alternatives define candidate solutions or constructions.	Distinguishes selection, assignment, ordering, and construction patterns.
Relations	Dependencies and compatibility conditions among objects or choices.	Provides structural signals such as connectivity, ordering, hierarchy, and equality.
Constraints	Input bounds and explicit feasibility or resource requirements.	Provides scale and resource signals for assessing algorithm applicability.
Transition Rules	Rules or inferred mechanisms for extending or updating a computational configuration.	Indicates relevant operations and state-evolution mechanisms.
Objective	The mathematical quantity or property required by the task.	Distinguishes optimization, counting, construction, and decision problems.
Answer Extraction	The condition or mapping that identifies the required output from a completed computation.	Matches output requirements to the result produced by a candidate method.

Table 7: Definition and role of the eight representation fields.

D.3 Controlled Structural Labels

Target problems, Algorithmic Knowledge, and Application Exemplars share predefined, field-specific vocabularies for the eight fields. Labels encode both problem properties and inferred computational mechanisms.

Folding is instructed to select a minimal set of supported labels, normally one to three per field, and to use empty arrays for unsupported or inapplicable fields. The implementation removes duplicates

and labels outside the assigned field's vocabulary. It also removes predefined redundant parent labels from the query when their more specific counterparts are present.

Matching uses exact label equality within each field, without implicit ancestor expansion or semantic equivalence between related labels.

D.4 Field and Label Weighting

Each label's contribution combines a fixed field coefficient with a corpus-specific inverse document frequency (IDF). In the field order of the eight-field representation, the coefficients are

$$\mathbf{w} = (1.20,\ 1.00,\ 0.50,\ 1.30,\ 0.90,\ 1.25,\ 1.10,\ 0.70).$$

Relations and Transition Rules receive the largest coefficients, while Decision Variables and Answer Extraction receive smaller ones.

For a loaded retrieval corpus $\mathcal{B}$, let $N = |\mathcal{B}|$ and let $\mathrm{df}_{\mathcal{B},f}(\lambda)$ denote the number of entries containing label λ in field f. Each entry contributes at most once to this count. The IDF weight is

$$\mathrm{idf}_{\mathcal{B},f}(\lambda) = \begin{cases} \min\left\{3,\ 1 + \ln \dfrac{N+1}{\mathrm{df}_{\mathcal{B},f}(\lambda)+1}\right\}, & \mathrm{df}_{\mathcal{B},f}(\lambda) > 0, \\ 3, & \text{otherwise.} \end{cases}$$

Thus, common labels receive less weight than rare labels, with an upper bound of 3. Valid query labels absent from the corpus also receive weight 3.

IDF statistics are computed separately for the loaded Algorithmic Knowledge and Application Exemplar collections, before query-specific filtering, and remain fixed during ranking and selection. The combined contribution is

$$\omega_{\mathcal{B},f}(\lambda) = w_f\ \mathrm{idf}_{\mathcal{B},f}(\lambda),$$

which is used in Equation 9.