Enhancing Competitive-level Code Generation by Utilizing Natural Language Reasoning

by

Jierui Li

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2025

Abstract

Enhancing Competitive-level Code Generation by Utilizing Natural Language Reasoning

Jierui Li, PhD
The University of Texas at Austin, 2025

SUPERVISOR: Raymond Mooney

Recent progress in large language models (LLMs) has shown strong performance in code generation. Models trained with long reasoning chains achieve promising results on complex competitive programming (CP) tasks. However, it remains unclear where the main bottlenecks in solving such problems lie. This dissertation studies these obstacles and explores how leveraging LLMs' natural language reasoning abilities can improve code generation for CP.

This proposal highlights three completed contributions:

- **Explanation and Distilling**: LLMs are effective at explaining solution code(Li et al., 2023), and their ability to implement a verbal solution is stronger than solving a problem directly. Based on this, we developed a supervised fine-tuning method that distills LLM-generated explanations into chain-of-thought style problem-solving steps(Li and Mooney, 2024).
- **Agent-Guided CodeTree Search**: We introduced CodeTree(Li et al., 2025), an agent system for code generation that iteratively thinks, solves, reflects, refines, and verifies through an auto-expanded tree search until reaching the final solution.

- **AlgoSimBench Benchmark**: We built AlgoSimBench(Li and Mooney, 2025), a benchmark for evaluating LLMs' ability to identify algorithmically similar problems. We found that using attempted solutions to match problems improves both end-to-end LLM selection and cosine similarity-based retrieval.

Finally, we outline two directions for future work.

- Task-Aware Code Representation: Develop a zero-shot code embedding method that weighs tokens based on the task-specific prompt, focusing the representation on distinct aspects such as algorithm, functionality, and semantics.
- Retriever–LLM Training: Investigate why Retrieval-Augmented Generation (RAG) shows limited improvement in coding tasks, with two hypotheses: (a) retrievers fail to find useful context, and (b) LLMs struggle to use retrieved information effectively. To address this, we plan to jointly train retrievers and LLMs on context-dependent coding tasks.

Chapter 1: Introduction

Programming has become a central tool for solving problems across science, engineering, and everyday applications. With the rise of large language models (LLMs), code generation has advanced rapidly, making it possible to automate tasks that once required expert knowledge. Yet competitive programming remains a particularly challenging domain: problems are complex, solutions demand precise reasoning, and small errors often lead to failure. Tang et al. (2023b) argued that LLMs rely heavily on semantic associations rather than formal symbolic reasoning, making them stronger at NL-style reasoning but poor at true symbolic logic. Understanding how LLMs can bridge the gap between natural language reasoning and executable code is therefore both a practical and scientific question, and it forms the focus of this proposal.

1.1 Background and Motivation

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022a; Lyu et al., 2023), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Chen et al., 2022; Zhou et al., 2023). Chain-of-thought (CoT) prompting (Wei et al., 2022b) addresses limitations of previous LLMs by instructing them to generate intermediate steps towards the final answer, thereby decomposing complex problems step-by-step.

LLMs trained on large corpora of natural language and code demonstrate strong performance across a range of software engineering tasks, from writing utility scripts to solving introductory programming challenges (Chen et al., 2021; Hendrycks et al., 2021a). These advances highlight the potential of LLMs to serve as general assistants for reasoning about and generating functionally correct code.

However, challenges remain, particularly in complex reasoning tasks like algo-

rithmic programming. For example, the majority of human competitors still outperform advanced models like GPT-4 in Codeforces contests (OpenAI, 2023b). Complex programming problems have stringent time and space complexity constraints, where straightforward implementation methods often yield time-consuming brute-force solutions.

This gap suggests that current LLMs, while highly capable in many natural language tasks, continue to face bottlenecks in structured algorithmic reasoning and reliable program synthesis. This dissertation explores a central thesis: the path to superhuman performance in competitive programming lies in explicitly structuring and leveraging the natural language reasoning capabilities of LLMs as a scaffold for code generation.

1.2 Research Questions

To investigate this thesis, my research follows a deliberate methodological arc, where the findings from each stage motivate the inquiry of the next. This progression is guided by five core research questions.

1.2.1 Diagnose the Bottleneck

To generate the correct solution for a complex problem, we propose three stages: 1)Find the appropriate strategy and solution to tackle the programming challenge. 2)Apply the strategy and finalize details of the exact method. 3)Implement the solution as an executable program.

In (Li et al., 2023), we decompose LLMs' capabilities in problem-solving code generation to perform these 3 stages sequentially, and evaluate them separately. This helps identify the bottleneck of solving complex competitive-level programming problems(CP). We want to answer the following research question: **RQ1: In the process of solving a complex programming problem, where is the primary bottleneck for LLMs?**

1.2.2 Utilizing Natural Language Reasoning as a Scaffold

This diagnostic results from 3.1 show that LLMs are good at explaining code, and can faithfully implement natural-language-described verbal solutions into executable programs, despite their poor abilities to directly solve them. This answers to **RQ1** and directly leads to **RQ2: Can we use LLMs' strength in explaining and NL reasoning to improve their problem-solving ability?** Human-written rationales for solving algorithmic reasoning problems, known as *editorials*, are hard to collect as they are often posted on personal blogs or as tutorial videos. An alternative is to distill such natural-language-described problem-solving strategies from larger models. Distilling explicit chain-of-thoughts (CoT) reasoning processes has been shown to be an effective method to learn multi-step reasoning from larger models (Hsieh et al., 2023; Yue et al., 2023). Usually, a teacher model is required to solve a set of problems while giving CoT reasoning paths at the same time. However, when facing challenging tasks where state-of-the-art models struggle to generate effective solutions, it becomes infeasible to gather reasoning processes at scale.

We found from our previous research (Li et al., 2023) that LLMs are good at explaining code, and Tang et al. (2023b) found that LLMs can learn Natural language (NL) reasoning abilities more easily than they learn symbolic language (e.g., programming language) reasoning.

In our work (Li and Mooney, 2024), we synthesize explanations of human-written code and utilize them as the chain-of-thoughts for a student model to learn. In our experiments, we found that it's more effective for LLMs to learn from natural language descriptions than to directly learn from code, which answers to **RQ2**.

1.2.3 Structuring NL Reasoning Within an Agentic Framework

Having validated the principle of separation, the next logical challenge was to build a system that could perform this process autonomously and robustly.

Modern code generation pipelines (Shinn et al., 2023; Wang et al., 2022b) have

adapted a general "generate → execute → refine" framework towards more robust code generation. In the previous works, we separated steps for complex CP problem-solving code generation. In addition, we are interested in: **RQ3: Can we always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline?** and **RQ4: How can we build an agent that systematically explores this decomposed reasoning space?**

1.2.4 Probing the Generalization of Algorithmic Reasoning

After developing a powerful problem-solving agent, the investigation naturally turns to the limits of its understanding. While solving CP problems is a challenging benchmark (Li et al., 2022b; Shi et al., 2024), it is not clear if the skills learned generalize beyond code generation. This raises our next question, **RQ5: Does a model's ability to solve specific problems reflect a deeper, generalizable understanding of the underlying algorithms?**

1.3 Proposal Outline

The remainder of this proposal details the completed and future work that addresses these research questions.

- Chapter 2 provides background on large language models for code generation and frames the core tasks.
- Chapter 3 details the completed works and experimental results, presented in the same order as the research questions described above.
- Chapter 4 outlines two proposed directions for future work, which directly address the limitations on generalization identified in our investigation of RQ5.
- Chapter 5 concludes with a summary and a proposed timeline for completing the dissertation.

Chapter 2: Background

2.1 Emergence of Large Language Models

Vaswani et al. (2017) proposed the Transformer architecture to only use the self-attention mechanism to obtain context-rich representations of tokens while encoding all tokens simultaneously. This design has opened the doors for efficient training and inference of LLMs. Efforts have been made to improve the structure in terms of attention mechanisms (Dao et al., 2022; Choromanski et al., 2022), more expressive position embeddings (Shaw et al., 2018; Dai et al., 2019; Su et al., 2023b), and efficiency (Shoeybi et al., 2020; Kwon et al., 2023).

Devlin et al. (2019) follows the Transformer architecture and learns language representation through pre-training on a large corpus. Other masked language models have been developed to improve upon BERT (Liu et al., 2019; Beltagy et al., 2020; Sanh et al., 2020; Clark et al., 2020).

In parallel, a lineage of causal language models, pre-trained with an autoregressive objective, gained prominence. This began with the Generative Pre-trained Transformer (GPT)(Radford et al., 2018) and evolved with GPT-2, which demonstrated that a sufficiently scaled model could perform a variety of tasks without explicit supervision, acting as an unsupervised multitask learner (Radford et al., 2019). This trajectory culminated in models like GPT-3, where immense scale unlocked the emergent ability of in-context learning (ICL), allowing the model to perform tasks given only a few demonstrations in the prompt, without any gradient updates(Brown et al., 2020; Gao et al., 2021). Further research revealed that complex reasoning could be elicited from these large models through Chain-of-Thought (CoT) prompting, which encourages the model to generate intermediate reasoning steps before providing a final answer (Wei et al., 2022b,a; Wang et al., 2023a).

The rapid growth in model size and capability was systematized by the dis-

covery of neural scaling laws, which established that model performance, measured by cross-entropy loss, scales predictably as a power-law with model size, dataset size, and the compute used for training (Kaplan et al., 2020; Hoffmann et al., 2022). These findings provided a principled framework for efficiently allocating computational budgets, demonstrating that larger models are more sample-efficient and that the optimal strategy involves training very large models on relatively modest amounts of data.

To adapt these powerful pre-trained models for specific downstream applications, various fine-tuning methodologies were developed. The Text-to-Text Transfer Transformer (T5) introduced a unified framework that treats every NLP task as a text-to-text problem, enabling a single model to be fine-tuned on a diverse mixture of tasks (Raffel et al., 2023). Building on this, instruction tuning was proposed to enhance zero-shot generalization by fine-tuning models on a collection of tasks described via natural language instructions (Wei et al., 2022a; Chung et al., 2022; Sanh et al., 2022; Wang et al., 2023b). Concurrently, knowledge distillation emerged as a key technique for model compression, enabling the transfer of capabilities from a large "teacher" model to a smaller "student" model to create more efficient deployments (Hinton et al., 2015; Sun et al., 2019; Xu et al.).

As LLMs became more capable, aligning their behavior with human intent became a critical challenge. Reinforcement Learning from Human Feedback (RLHF) was established as a powerful, multi-stage process to steer models toward generating outputs that are helpful, honest, and harmless. This process involves supervised fine-tuning (SFT), training a reward model on human preference data, and then optimizing the language model policy using reinforcement learning (Ouyang et al., 2022; Stiennon et al., 2020). To simplify this complex and often unstable pipeline, subsequent work introduced more direct alignment methods. Constitutional AI proposed using AI-generated feedback (RLAIF) based on a set of principles to automate the preference labeling process (Bai et al., 2022). More recently, Direct Preference Optimization (DPO) and its variants have emerged as a stable and computationally lightweight alternative that bypasses the explicit reward modeling step, optimizing the policy

directly on preference data through a simple classification loss (Rafailov et al., 2023; Ethayarajh et al., 2024; Shao et al., 2024).

Finally, to address the static nature of parametric knowledge and the tendency for models to hallucinate, Retrieval-Augmented Generation (RAG) was developed. This framework grounds LLMs in external, verifiable knowledge by combining a pre-trained parametric model with a non-parametric memory, such as a dense vector index of a text corpus. By retrieving relevant documents and providing them as context in the prompt, RAG improves factual accuracy, allows for knowledge updates without retraining, and enhances the interpretability of model outputs (Lewis et al., 2020b; Guu et al., 2020; Karpukhin et al., 2020).

2.2 Symbolic Reasoning with Large Language Models

"*Symbolic reasoning represents concepts or objects as symbols instead of numbers and manipulates them according to logical rules. Neuro-symbolic AI combines the deep learning capabilities of neural networks with symbolic reasoning for more robust decision-making. This is a fairly recent advancement and is still an emerging area of research.*" (Caballar and Stryker, 2025)

Recent Large Language Models (LLMs) have shown impressive capabilities for various reasoning tasks, including multi-hop question answering (Wang et al., 2022a; Lyu et al., 2023), commonsense reasoning (Zelikman et al., 2022), symbolic reasoning (Hua and Zhang, 2022), and math word problem-solving (Zhou et al., 2023; Chen et al., 2022).

Gendron et al. (2024); Tang et al. (2023b) argue that LLMs rely heavily on semantics in tokens and contexts, and struggle more when semantics are inconsistent or when symbolic/counter-commonsense reasoning is needed. Besides training LLMs on symbolic reasoning tasks(MA et al., 2024), many methods are proposed to enhance LLMs' capabilities to do symbolic reasoning.

The chain of thought prompting method (Wei et al., 2022b) explicitly instructs LLMs to generate intermediate steps until the final answer is reached, enabling the model to decompose problems and solve them step by step. Wang et al. (2023a); Zhou et al. (2023); Yao et al. (2024) propose to approach the correct answer through multipath. They leverage multiple generations on the same question to find the correct answer. Yao et al. (2023); Schick et al. (2023) decompose the reasoning process into a chain of actions and utilize tools to perform actions. Similarly, Chen et al. (2022); Zhu et al. (2023) shift from textual steps to executable programs, leveraging interpreters for exactness.

2.3 Code Generation and Programming Tasks

Programming has become a central tool for solving problems across science, engineering, and industry. With the rise of large language models (LLMs), automatic code generation has advanced rapidly, making it possible to automate tasks that previously required expert programming knowledge. LLMs trained on large corpora of natural language and code demonstrate strong performance across a range of software engineering tasks, from writing utility scripts to solving introductory programming challenges (Chen et al., 2021; Hendrycks et al., 2021a). These advances highlight the potential of LLMs to serve as general assistants for reasoning about and generating functionally correct code.

While generating correct code is a central goal, recent research has started to explore the broader space of what "understanding code" means. One direction focuses on code efficiency—generating programs that are not just correct but also optimal in time or space. Works like Shypula et al. (2024), BigOBench (Chambon et al., 2025), and ECCO (Waghjale et al., 2024) evaluate models' ability to reason about and improve algorithmic efficiency in time and space. Similarly, NoFunEval (Singhal et al., 2024) proposed to evaluate LLMs' programming abilities beyond functional correctness. Another important area is code reasoning—testing whether models can

simulate, explain, or analyze program behavior. CRUXEval (Gu et al., 2024) tests models' ability to predict program outputs given specific inputs, revealing weaknesses in dynamic reasoning. Pei et al. (2023) study whether models can identify invariants and reason about variables during code execution. EquiBench (Wei et al., 2025) poses the deceptively simple task of determining functional equivalence between two programs. Efforts have also gone into understanding how well models can explain programs. Li et al. (2023) test LLMs on providing human-readable explanations of competitive programming solutions, evaluating both clarity and faithfulness. CodeMind (Liu et al., 2024) proposes the reverse task—converting code into its natural language specification (Code2NL)—as a counterpart to the well-studied NL2Code direction. CodeRAGBench(Wang et al., 2025) has studied if code retrieval can augment better code generation tasks.

2.4 Competitive-level Programming Problem Solving

Despite these successes, competitive programming (CP) remains a particularly difficult frontier. Problems in this domain are designed to test algorithmic thinking under strict time and space constraints, where even small reasoning errors or inefficient implementations lead to failure. While models like GPT-4 (OpenAI, 2023b) can occasionally produce correct solutions, they are still far from matching strong human competitors in CP contests. This gap suggests that current LLMs, while highly capable in many natural language tasks, continue to face bottlenecks in structured algorithmic reasoning and reliable program synthesis.

One fundamental limitation lies in the type of reasoning LLMs perform. As argued by Tang et al. (2023b), LLMs rely heavily on semantic associations and are often stronger at natural language style reasoning than at strict symbolic logic. This difference is important in programming: solving a problem requires identifying the right algorithmic strategy, working out the technical details, and only then producing correct code. Failures in any of these stages lead to incorrect solutions. Existing stud-

ies in multi-step reasoning, such as chain-of-thought prompting (Wei et al., 2022b), show that explicitly structuring intermediate reasoning steps improves accuracy on math and symbolic tasks (Zhou et al., 2023; Hua and Zhang, 2022), but CP poses additional difficulties due to large search spaces and strict efficiency constraints.

At the same time, the strengths of LLMs provide an opportunity. Our prior work has shown that models are surprisingly effective at generating explanations of code, even when they fail to produce correct solutions (Li et al., 2023). This indicates that natural language reasoning, when separated from direct code generation, can be a powerful resource. Recent work in distillation (Hsieh et al., 2023; Yue et al., 2023) suggests that transferring reasoning traces into smaller models can improve their problem-solving ability. These findings raise the possibility of harnessing LLMs' explanatory abilities to overcome their weaknesses in direct program synthesis.

Beyond reasoning at the single-problem level, there is growing interest in agentic approaches to code generation (Shinn et al., 2023; Wang et al., 2022b). Instead of producing one-shot outputs, LLM agents can generate candidate programs, execute them, reflect on the results, and iteratively refine their solutions. Such frameworks address the brittleness of single-pass generation and move closer to the problem-solving process used by human programmers. However, designing agent systems that can efficiently explore large spaces of potential solutions remains challenging, especially for competitive programming where the number of possible strategies is vast.

Another open question is whether models trained for CP-style reasoning can generalize to broader forms of algorithmic understanding. Identifying algorithmically similar problems (ASPs)—that is, recognizing when two problems can be solved with the same underlying approach—represents one such capability. ASP identification is important both for retrieval-based systems and for assessing whether models capture deeper algorithmic structure rather than surface-level patterns. Early benchmarks suggest that even state-of-the-art LLMs struggle in this setting, performing well below human intuition.

Taken together, these observations motivate a systematic study of how LLMs can bridge the gap between natural language reasoning and executable program synthesis. The central theme of this dissertation is to leverage the strengths of LLMs in explanation and natural language reasoning to improve their weaknesses in structured problem solving, symbolic reasoning, and reliable code generation for competitive programming tasks.

We pursue this theme through a sequence of investigations that form a coherent ladder:

- First, we diagnose the bottlenecks in LLMs' ability to solve CP problems by decomposing the problem-solving process into strategy identification, detail elaboration, and implementation.
- Second, we develop methods that harness LLMs' explanatory and natural language reasoning abilities, distilling them into problem-solving hints and agent-driven reasoning pipelines.
- Third, we extend beyond single-problem solving to benchmarks that test whether models generalize algorithmic reasoning across tasks, focusing on ASP identification and code retrieval.
- Finally, we outline future directions that build on these findings, including task-aware code representation and integrated retriever–LLM training for context-dependent coding.

By following this trajectory, the dissertation aims to both shed light on the fundamental limitations of current LLMs in algorithmic reasoning and to propose practical pathways for improvement. In the next section, we formalize the guiding research questions that drive this agenda.

Chapter 3: Completed Works

3.1 Explaining CP Solutions Using LLMs

In this section, we (Li et al., 2023) aim to identify the gaps in LLMs' capabilities and answer **RQ1: In the process of solving a complex programming problem, where is the primary bottleneck for LLMs?**

3.1.1 Overview

We approach competitive-level programming problem-solving as a composite task of reasoning and code generation. We propose a novel method to automatically annotate natural language explanations to < *problem, solution*> pairs. We show that despite poor performance in solving competitive-level programming problems, state-of-the-art LLMs exhibit a strong capacity in describing and explaining solutions. Our explanation generation methodology can generate a structured solution explanation for the problem containing descriptions and analysis. To evaluate the quality of the annotated explanations, we examine their effectiveness in two aspects: 1) satisfying the human programming expert who authored the oracle solution, and 2) aiding LLMs in solving problems more effectively. The experimental results on the CodeContests dataset demonstrate that while LLM GPT3.5's and GPT-4's abilities in describing the solution are comparable, GPT-4 shows a better understanding of the key idea behind the solution.

3.1.2 Method

In the process of problem-solving, a human typically constructs a solution by progressing from a general idea to a detailed code implementation. Thus, we propose General-to-Specific(G2S) prompting, guiding LLMs to come to a solution from general aspects like the algorithm to specific implementation details. However,

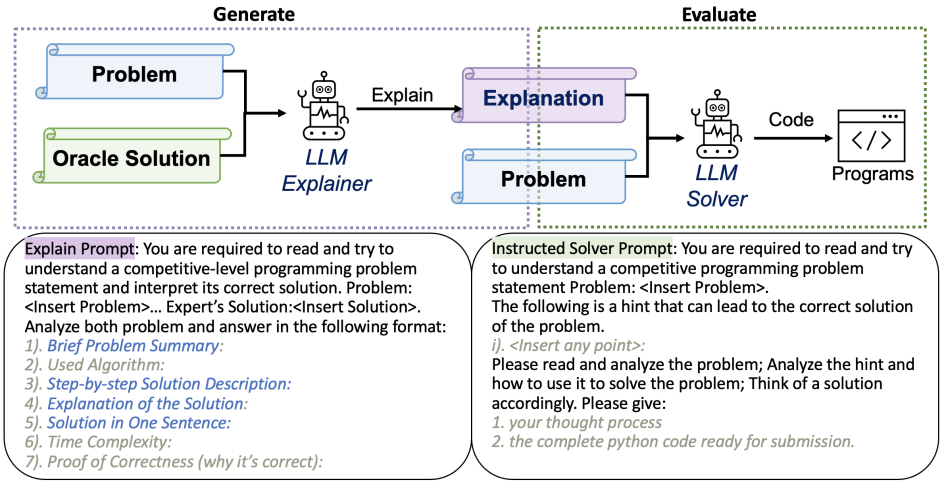


Figure 3.1: The explanation generation and evaluation framework and corresponding prompts (Top). An example of the full explain prompt (Bottom Left). The blue points are descriptions, while the grey points are analysis. We give the explanation based on the oracle solution to the instructed solver as a hint (Bottom Right) to evaluate the quality of the generated explanation.

explaining that solution involves a reverse approach. This entails examining the code on a line-by-line basis, interpreting the role of each function, and then rationalizing the algorithmic steps in relation to the original problem. Therefore, we design a specific-to-general explanation generation method.

While useful explanations might be generated with simple prompts such as "explain the solution", they often lack crucial information and are difficult to evaluate due to the diversity in output. To address this issue, we specifically control the aspects of explanations to include a "problem summary" showing its understanding of the problem; 3 levels of "natural language description of the problem" showing its ability to comprehend the solution from a low-level perspective to a high-level one. Those can be regarded as *Description-level* explanation. The points "used algorithm", "time complexity", "proof of correctness" are *Analysis-level* explanations, showcasing the LLM's general analysis and understanding of the solution. The specific-to-general explaining prompt is described in the left part of Figure 3.1.

3.1.3 Main Experimental Results

In this section, we aim to evaluate the specific-to-general explanations from two different angels: 1). How much does it satisfy the authors who wrote the solution? 2). How easily can LLMs implement the solution back to the correct python code?

Experimental Settings and Evaluation Metrics We use GPT-3.5 and GPT-4 as our main models to evaluate. We use the CodeContests(Li et al., 2022b) test set as our main dataset in this paper. It contains 165 real online contest problems from *Codeforces*, the earliest of which dates back to Oct 2021, which is after the knowledge cutoff of GPT-3.5 and GPT-4 (Sep. 2021). We employ pass@k (Chen et al., 2021) as our evaluation metric for solve rate. For each problem p_i, we sample k programs generated, and evaluate them using **Solve Rate@k** metric: the percentage of programs that pass all hidden test cases, when submitted to Codeforces' online judge. We first filter the programs by their output on the public test cases before submitting them, and also measure **Pass Public@k**: the percentage of programs that pass the public test cases given in the examples. The above metrics are abbreviated as 'solve@k' and 'public@k'. Baseline is the original 0-shot CoT prompting; G2S prompt is General-to-Specific prompting where it demonstrates the model to think in a reverse order of the aspects in 3.1. "w/" is with a single aspect from the explanation as the hint.

Human Evaluation: Author Likert Scores We measured the quality of LLM-generated explanations using human evaluation. We collect 50 <*problem, solution*> pairs from Codeforces, ensuring that their format remained consistent with those in CodeContests. Recognizing that understanding and explaining others' solutions can be a challenging task for programmers, we employed an annotator-centered evaluation approach. We extracted solutions and corresponding problems from Codeforces for an expert annotator. The Explainer then generates an explanation for the annotator's

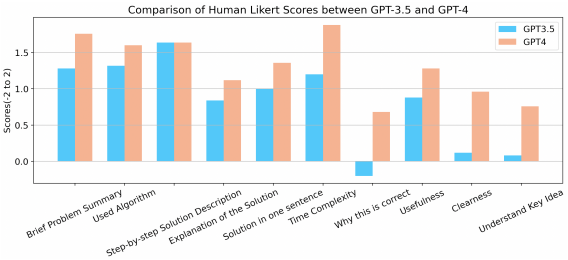


Figure 3.2: Human Likert scores (-2: very poor to 2: excellent) evaluating various aspects of the explanations.

solution, which was subsequently scored by the author of the explained solution. Note that each explanation is scored by the author of the solution being explained.

We generated explanations for 50 problems with ratings ranging from 800 to 2000, along with their corresponding solutions, and provided these explanations to human experts. They were asked to assign a Likert score from -2 (very poor) to 2 (excellent).

The evaluation consists of ten questions, each one corresponding to a specific aspect of the explanation. We separately assess the quality of the response to each point of our S2G prompt. Furthermore, we developed three criteria to evaluate various aspects of the overall explanation:

1. *Usefulness*: How useful is the explanation as guidance to solve the problem?
2. *Clearness*: How good is the explanation in terms of describing everything clearly and avoiding ambiguity?
3. *Understanding*: How much does the LLM understand the key idea behind the solution?

The average Likert scores over 50 problems are shown in Figure 3.2. Regarding the scores for the solution descriptions (*Step-by-Step Solution Description, Explanation of the Solution, Solution in One Sentence*) and usefulness, both GPT-3.5 and GPT-4 Explainer are positively rated by expert annotators, with an average of 1.16 and 1.36

respectively. However, GPT-3.5 receives near-zero or negative scores on questions including *why it's correct, clearness, and understanding*, showing its inadequate ability to grasp the key idea behind the solution, while GPT-4 performs better ($0.68 \sim 0.88$ score higher) on these aspects. This reveals a clear difference in the abilities of GPT-3.5 and GPT-4 to reason and analyze competitive-level programming solutions. For GPT-3.5, we measure pass@k for $k = \{1, 5, 10\}$, but only pass@1 for GPT-4 due to access limits. To sample k programs, we sample k different human solutions for Explainer and then generate a program for each explanation.

GPT-3.5 Solver				
	solve@1	solve@5	solve@10	public@10
Baseline	1.8	3.6	6.1	13.9
G2S prompt	2.4	5.4	9.1	18.8
GPT-3.5 Solver With Silver Explanation				
w/ Used Algorithm	1.8 *(1.2)*	4.2	6.1	13.3
w/ Step-by-Step Solution Description	13.3 *(15.8)*	32.2	42.4	47.9
w/ Explanation of Solution	6.1 *(4.8)*	17.6	23.6	32.7
w/ One Sentence Solution Description	4.2 *(4.2)*	9.1	13.9	26.1
w/ Time Complexity	1.8 *(2.4)*	3.6	6.7	13.3

Table 3.1: Different aspects of the explanation's effect on improving program generation. Values are percentage % and 'solve' and 'public' are short for 'Solve Rate' and 'Pass Public Tests'. Solve@1 results in *parentheses* are from GPT-4's generated explanations. The bottom 5 rows correspond to Figure 3.1's points 2,3,4,5, and 6 in the left prompt.

Automatic Evaluation Results are shown in Table 3.1. Different *Description-level* aspects of explanations improve both the solve rate and pass public rate. The most detailed aspect, *Step-by-Step Solution Description*, which provides a detailed natural language description of the implementation, offers the most significant benefit to problem-solving, resulting in a solve rate @1 that is 7.4 times higher than the baseline. The impact of *Explanation of the Solution* and *Solution in One Sentence* is comparatively lower due to their concise nature, which offers a less clear path towards the solution. However, providing information on the algorithms used or the expected time complexity does not improve GPT-3.5's problem-solving capabilities.

One observation from Table 3.1 is that solve@10 is significantly less than public@10. For a program that passes the public tests but fails the hidden tests, there are two possibilities: 1) It is incorrect and only applies to a subset of test data, including the public tests; 2) It is inefficient. As discussed before, in competitive-level programming, a "correct" but slow implementation does not count as a solution, as there are constraints on time and space complexity. Therefore, we further study programs that pass the public tests but may fail hidden tests. As shown in Table 3.2, the baseline has 48.9% of its programs rejected by the online judge due to inefficiency, indicating that GPT-3.5 tends to generate inefficient implementations (e.g., brute force solutions).

	Solve	Wrong Answer	TLE	Other
Baseline	35.1%	15.6%	48.9%	0%
G2S prompt	38.3%	14.1%	47.6%	0%
w/ UsedAlg	39.1%	18.9%	42.1%	0%
w/ S-by-S	75.6%	11.4%	11.4%	1.6%
w/ Exp-Sol	73.6%	11.9%	11.1%	1.4%
w/ OneSent	56.6%	27.9%	14.0%	1.5%

Table 3.2: Final judgement of generated programs that pass the public tests. TLE means time limit exceeded, and *other* includes memory limit exceeded and runtime error.[1]

When provided hints from the solution description, the portion of Time Limit Exceeded(TLE) programs drops significantly. Although GPT-3.5 may still make mistakes in some details or fail to consider corner cases even with hints from the explanation, it is better at avoiding inefficient solutions.

Another interesting observation is that the wrong answer rate for one-sentence explanation-instructed solving is higher than the baseline. One possible explanation is that it is challenging to incorporate corner case handling in a one-sentence solution description, which makes GPT-3.5 more likely to implement an almost-correct program.

[1]This is for all submissions, i.e., one problem might have up to k submissions, which is different from the problem-wise solve rate.

3.1.4 Conclusion

We conclude the following findings from the experimental results above.

- Despite the poor abilities to solve CP, LLMs exhibit reasonable capabilities in generating faithful and clear descriptions and explanations of the solutions.
- GPT-4 outperforms GPT-3.5 significantly in analyzing the problem and solution, as well as capturing the key ideas behind the solution.
- Given the step-by-step description of how to solve the problem, even the weaker LLM GPT-3.5 can generate the correct solution. This highlights the answer to RQ1, that LLMs are strong implementers.

When a hint is based on an oracle human solution, it effectively guides the LLM to generate improved programs for solving problems. However, a system should be able to learn from human programming solutions to improve its own problem-solving on novel problems without guidance from a human solution. This raises the question: Can the LLM-generated explanations be utilized to improve subsequent problem-solving? This leads to the next completed work.

3.2 Distilling Algorithmic Reasoning from LLMs via Explaining Solution Programs

In this section, we(Li and Mooney, 2024) utilize the findings from our last work, and answer **RQ2: Can we use LLMs' strength in explaining and NL reasoning to improve their problem-solving ability?**

3.2.1 Overview

Distilling explicit chain-of-thought reasoning paths has emerged as an effective method for improving the reasoning abilities of large language models (LLMs) across various tasks (Wang et al., 2022a; Chen et al., 2022; Zhou et al., 2023). However, when

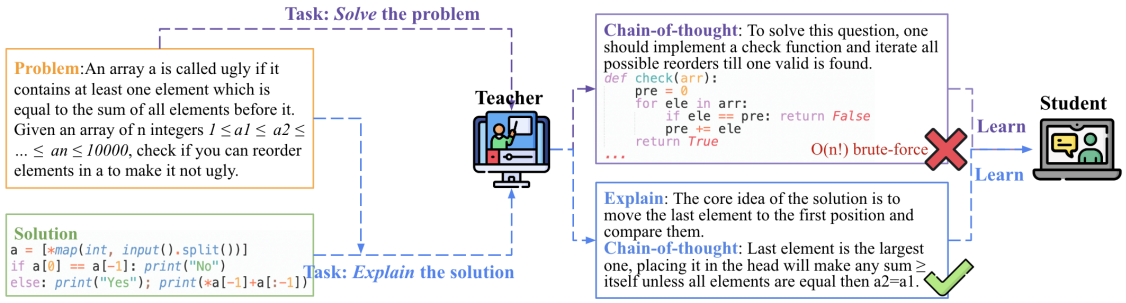


Figure 3.3: Comparison between Solve-based and Explain-based chain-of-thoughts distilling. Top: Solve-based CoT distilling is likely to generate incorrect or inefficient solutions. Bottom: Explain-based CoT distilling can generate high-quality reasoning processes by explaining the oracle solution.

tackling complex tasks that pose significant challenges for state-of-the-art models, this technique often struggles to produce effective chains of thought that lead to correct answers. In this work, we propose a novel approach to distilling reasoning abilities from LLMs by leveraging their capacity to explain solutions. We apply our method to solving competitive-level programming challenges. More specifically, we employ an LLM to generate explanations for a set of <*problem, solution-program*> pairs, then use <*problem, explanation*> pairs to fine-tune a smaller language model, which we refer to as the Reasoner, to learn algorithmic reasoning that can generate "how-to-solve" hints for unseen problems. Our experiments demonstrate that learning from explanations enables the Reasoner to more effectively guide the program implementation by a Coder, resulting in higher solve rates than strong chain-of-thought baselines on competitive-level programming problems. It also outperforms models that learn directly from <*problem, solution-program*> pairs. We curated an additional test set in the CodeContests format, which includes 246 more recent problems posted after the models' knowledge cutoff.

3.2.2 Method

Human-written rationales for solving algorithmic reasoning problems, known as *editorials*, are hard to collect as they are often posted on personal blogs or as

tutorial videos. An alternative is to distill such natural-language-described problem-solving strategies from larger models. Distilling explicit chain-of-thoughts (CoT) reasoning processes has been shown as an effective method to learn multi-step reasoning from larger models (Hsieh et al., 2023; Yue et al., 2023). Usually, a teacher model is required to solve a set of problems while giving CoT reasoning paths at the same time, as illustrated in Figure 3.3. However, when facing challenging tasks where state-of-the-art models struggle to generate effective solutions[2], it becomes infeasible to gather reasoning processes at scale.

To tackle this problem, we propose to utilize large language models' capacities in understanding and explaining solutions rather than solving problems. Specifically, we leverage a state-of-the-art LLM to read both the problem statement and an oracle human-written solution program, and generate an *editorial*-style chain-of-thought on how to solve the problem by explaining the solution. Then, a student LLM learns the algorithmic reasoning processes from the explicit chain-of-thought paths. We compare our explain-based CoT distilling method with solve-based CoT distilling in Figure 3.3. While solve-based CoT distilling requires the teacher model to reach the correct and efficient solution to hard problems, our explain-based distilling only requires the teacher model to faithfully explain the correct solution. Since explaining competitive-level code is a feasible task for strong LLMs 3.1, explain-based distilling can yield CoT reasoning processes with high quality and less noise.

We further proposed a reason-then-implement framework to solve algorithmic reasoning problems, utilizing the proposed explain-based CoT distilling. The framework consists of 3 major components: 1) an **Explainer** to annotate explanations for a set of <*problem, solution-program*>; 2) a **Reasoner** to learn to generate intermediate reasoning processes for a given problem; and 3) a **Coder** to implement the solution for an unseen problem *given* the output from the **Reasoner**. The framework and its

[2]Experiments show that only 12% of GPT-4's generated solutions are correct given 200 problems randomly sampled from the CodeContests training set.

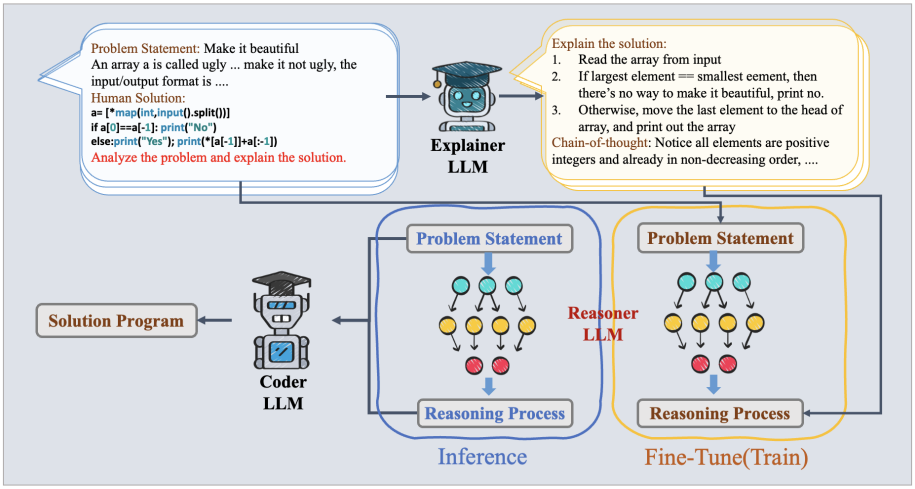


Figure 3.4: The framework of our approach. We use **Explainer** LLM to generate explanations then train **Reasoner** LLM to generate explanations given problem statements. During inference time, given the problem, the **Reasoner** can generate a reasoning process in the same format as solution explanations, which could be provided to the **Coder** as a hint to solve the problem better.

fine-tuning and inference stages are presented in Figure 3.4.

Explainer: Extracting reasoning processes through explaining solution programs The **Explainer** is tasked with explaining a solution program. It serves the role of Teacher in Solve-based CoT distilling, as described in Figure 3.3. Given a pair, $< p_i, s_i >$, it generates an explicit reasoning process e_i in natural language. This is inspired by how human competitors learn problem-solving skills from past problems: they learn by reading editorials, step-by-step guidelines on approaching and solving the problems. While human-written editorials are hard to collect or annotate at scale, we ask an LLM to automatically generate them. We design an editorial-style chain-of-thought reasoning template and ask the **Explainer** to follow it to explain solutions. Specifically, an editorial for an algorithmic reasoning problem refers to a comprehensive explanation or walk-through on how to solve a problem, which includes problem analysis, strategy development, solution explanation, time/space complexity analysis, etc. We leverage the capabilities of LLMs to explain solution code to gen-

erate automated explanations for problem-solution pairs $< p_i, s_i >$. The **Explainer** is prompted to create a detailed explanation, d_i, for each pair. Our focus is predominantly on the reasoning process, encompassing the following critical aspects that are often included in human-expert-written editorials:

1. Problem Restatement: Summary and analysis of the problem.
2. Conceptual Evolution: How one approaches the problem and the development of a problem-solving strategy.
3. Key to Solution: The key idea behind the solution.
4. Solution Description: Brief, verbal description of the solution, focusing on the high-level algorithm.
5. Step-by-Step Solution Explanation: A more detailed description, focusing on the steps in the implementation.
6. Common Pitfalls: Some common mistakes one could make when approaching the problem, or edge/corner cases to be considered.

Based on the length of p_i and s_i, as well as the difficulty ratings (ranging from 800 to 3600), we filter the training set from Li et al. (2022a) to curate a dataset $\{< p_1, s_1 >, \cdots, < p_n, s_n >\}$. Utilizing GPT-4-0613 as the Explainer, we generate "silver" explanations for these pairs, resulting in a comprehensive problem-solution-explanation dataset $< p_1, s_1, d_1 >$, ..., $< p_n, s_n, d_n >$. After further cleaning, we have 8248 triplets in total.

Reasoner: Fine-tuned to generate reasoning processes for problems Given the inherent diversity and potential lack of readability of the code available for algorithmic reasoning problems, our approach focuses on fine-tuning the Reasoner on problem statements p_i and reasoning processes d_i, which are ¡p_0, d_0¿, ¡p_1, d_1¿ $\cdots, < p_n, d_n >$. These problems and reasoning processes, rich in semantics, encapsulate the essential steps for problem resolution, including the algorithms used and

specific problem-solving approaches employed. The **Reasoner** serves the role of Student in Solve-based CoT distilling, as illustrated in Figure 3.3. We adopt a weighted fine-tuning strategy, with simpler, more recent problems weighted more heavily during training. Overly challenging problems might lead to low-quality noisy explanations that hurt the training of the **Reasoner**. Recent solutions usually feature more in-date implementations (e.g., Python 3 rather than Python 2), enhancing the code's interpretability. We fine-tune an LLM on 8,248 $\{p_i, d_i\}$ pairs, and then use the fine-tuned model as the final Reasoner. At the inference time, it generates $\hat{d}_j$ for the jth problem statement p_j. The generated reasoning process $\hat{d}$ can be considered as a natural-language "hint" given to the **Coder** to help solve the problem.

Coder: Reasoner-Hinted Code Implementer Finally, we have a zero-shot Coder to generate code utilizing hints from the **Reasoner**. Since the focus of this work is enhancing *algorithmic reasoning* rather than *code implementation* abilities, we do not further fine-tune the **Coder**. As described earlier, the reasoning process $\hat{d}_j$ for problem p_j contains a problem restatement, conceptual evolution, key to the solution, solution description, step-by-step solution explanation, and common pitfalls, making it a detailed hint to assist the implementation of the solution. We concatenate $< p_j, \hat{d}_j >$ as the input to produce the input for the Coder, which it then uses to analyze the problem together with the reasoning hint, to generate programs that solve the problem.

3.2.3 Experimental Results

In this section, we would like to know, how much performance gain can distilling from explanations bring to the CP solving task.

Experimental Settings and Evaluation Metrics We use GPT-4-0613 (OpenAI, 2023a,b) as the **Explainer** since it's the strongest model that we have access to. We also choose the strongest models with fine-tuning access as the **Reasoner** and

Coder. For a closed model, we choose GPT-3.5-turbo-1106 (henceforth GPT-3.5); for an open model, we choose Deepseek Coder 7B (henceforth Deepseek 7b) (Guo et al., 2024).[3] The temperature t is set to 0.5 when sampling multiple(¿1) is needed in our main experiments. The context window is set to 4096 and we truncate single examples with more than 4096 tokens. We use **solve@k** from the previous section as our main evaluation metric.

Dataset	**CF-Prob**						**CodeContests**					
Coder/Reasoner Model	GPT-3.5			Deepseek 7b			GPT-3.5			Deepseek 7b		
Zero-shot Methods												
	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10	solve@1	solve@5	solve@10
Direct Prompt	1.1	2.7	3.3	0.4	0.9	1.6	1.9	3.5	4.2	0.8	1.2	1.8
Naive CoT	1.2	2.7	3.3	0.7	1.4	2.0	1.8	3.7	4.8	0.8	1.4	1.8
Editorial CoT	1.1	2.7	3.6	0.9	2.0	2.4	1.5	3.7	4.8	1.0	1.8	2.4
0-Reasoner Coder	1.2	2.5	3.3	0.7	1.9	2.4	1.4	3.5	4.2	1.1	2.0	2.4
Fine-tuning Methods												
Fine-tune Coder	0.5	0.8	1.6	0.6	1.7	1.8	0.8	1.2	1.8	1.1	1.4	1.9
Fted Reasoner w/Full	1.1	3.2	4.9	0.6	2.1	3.7	1.5	4.2	6.1	1.4	2.8	3.6
Fted Reasoner w/Best	1.1	3.7	6.1	1.0	2.9	4.1	2.4	5.6	7.2	1.4	2.9	4.2
Relative Increment		+37%	+69%		+53%	+71%		+51%	+50%		+45%	+75%

Table 3.3: Performance of baselines and our methods. Fine-tuned Reasoner (Full) refers to using all aspects from the reasoning process, while Fine-tuned Reasoner (Best) only uses the best aspect. Relative Increment(by percentage) is what's over the best baseline. Solve@k: Solve rates(Percentage) when sampling k.

Main Results The results are presented in Table 3.3. With our fine-tuned **Reasoner**, the **Coder** achieves the best solve@10 and solve@5 rates across open/closed models and two datasets, supporting the effectiveness of our method. Compared to using the original model as the Reasoner (0-shot Reasoner), the best aspect (Step-by-Step) from the reasoning process alone can boost solve@10 from 3.3% to 6.1%, suggesting that our fine-tuning method does distill some algorithmic reasoning ability into the student model. Note that solely fine-tuning <*problem, solution-program*>pairs actually hurts performance. One reason might be that solutions to Codeforces problems are often written under a time constraint with poor readability, making it difficult for the model to generalize given the limited amount of fine-tuning data. Since

[3] https://huggingface.co/deepseek-ai/deepseek-coder-7b-instruct-v1.5

instruction-tuned models would generate intermediate thought processes by default, we experimented to explicitly forbid the model from any reasoning/analysis preceding implementation. Compared to the direct prompt, the solve@10 rate on CF Prob w/GPT-3.5 drops from 3.3% to 2.0%. Even with this setting, the generated code often contains comments and meaningful variable/function naming that are less in human competitors' code. This observation further supports our claim that meaningful, semantic-rich natural language can help the model generalize to unseen problems.

3.2.4 Conclusion

In this study, we propose to distill large language models' ability to explain solutions into reasoning abilities used to help solve problems. Using a two-phase framework of reason-then-implement where the student model plays the role of a **Reasoner** to instruct a zero-shot **Coder** to generate the implementation. Experiments on real problems from Codeforces demonstrate that our explain-based distilling outperforms several strong 0-shot baselines as well as fine-tuning with code solutions alone. This answers RQ2: Yes, we can use LLMs' strength in explaining and NL reasoning to improve their problem-solving ability.

3.3 Codetree: Agent-guided tree search for code generation with large language models

In the previous sections 3.1, 3.2, we find that LLMs are better at natural language reasoning and verbal solution generation than symbolic reasoning and direct program generation. We aim at utilizing NL reasoning in a zero-shot setting by answering the following two questions. **RQ3: Can we always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline?** and **RQ4: How can we build an agent that systematically explores this decomposed reasoning space?**

3.3.1 Overview

Pre-trained on massive amounts of code and text data, large language models (LLMs) have demonstrated remarkable achievements in performing code generation tasks. With additional execution-based feedback, these models can act as agents with capabilities to self-refine and improve generated code autonomously. However, on challenging coding tasks with extremely large search space, current agentic approaches still struggle with multi-stage planning, generating, and debugging. To address this problem, we propose CodeTree, a framework for LLM agents to efficiently explore the search space in different stages of the code generation process. Specifically, we adopted a unified tree structure to explicitly explore different coding strategies, generate corresponding coding solutions, and subsequently refine the solutions. In each stage, critical decision-making (ranking, termination, expanding) of the exploration process is guided by both the environmental execution-based feedback and LLM-agent-generated feedback. We comprehensively evaluated CodeTree on 7 code generation benchmarks and demonstrated the significant performance gains of CodeTree against strong baselines. Using GPT-4o as the base model, we consistently achieved top results of 95.1% on HumanEval, 98.7% on MBPP, and 43.0% on CodeContests. On the challenging SWEBench benchmark, our approach led to significant performance gains.

3.3.2 Method

Yao et al. (2024) proposed to improve LLMs by adopting a tree-based structure to explicitly simulate the exploration of thoughts in a tree. We are motivated by this line of research and proposed CodeTree, a new generation framework to effectively explore the search space of code generation tasks through a tree-based structure(Li et al., 2025). An overview of CodeTree is given in Figure 3.5.

We define 3 standard agents, Thinker, Solver, and Debugger, to equip the strategy-planning, solution implementation, and solution improvement correspond-

ingly, posing comprehensive roles needed for code generation. A CodeTree starts from the input problem as the tree root and subsequent nodes represent code solutions. At any node of the tree, one can either explore sibling nodes (other strategies from the same parent node) or its children (refinements of this node). Within CodeTree, agents can interact with each other through a tree expansion guided by a Critic Agent, searching for the optimal code solution.

Please refer to Figure 3.5 for an overview of our method and Figure 3.6 for a simplified version of instruction prompts to our LLM agents.

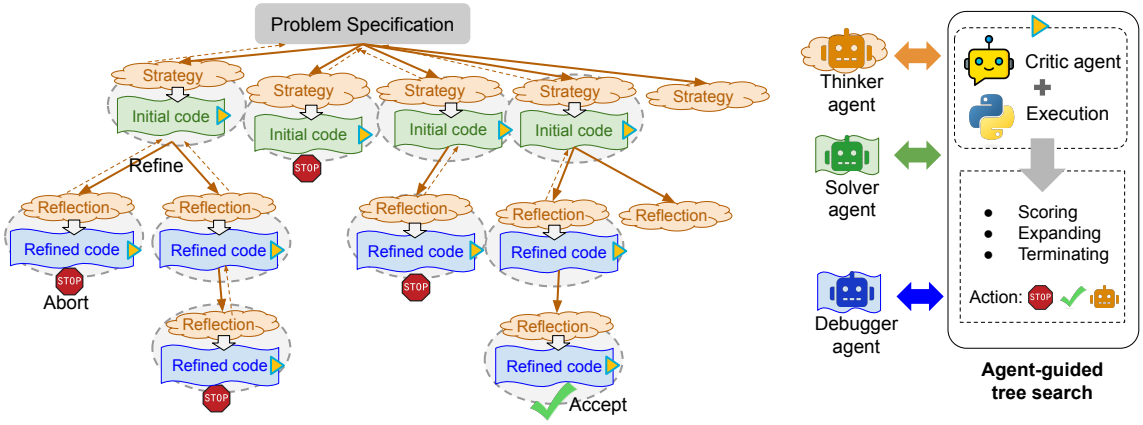


Figure 3.5: CodeTree creates a unified search space for exploration throughout the multi-stage code generation process: strategy generation by a "Thinker" agent, initial code generation by a "Solver" agent, and code improvement by a "Debugger" agent. To effectively perform exploration within the tree structure, we incorporate both environmental execution-based feedback as well as AI-generated feedback (generated by a "Critic" LLM agent).

We first introduce three unit LLM agents, specifically targeting different parts of the code generation process, including strategy thinking, code implementation, and code debugging.

Strategy Generation with Thinker Agent Following the setting in Yao et al. (2024), requesting LLMs to generate a list of different natural language *thoughts* can enhance the diversity of solutions. We propose to adapt this technique to allow an

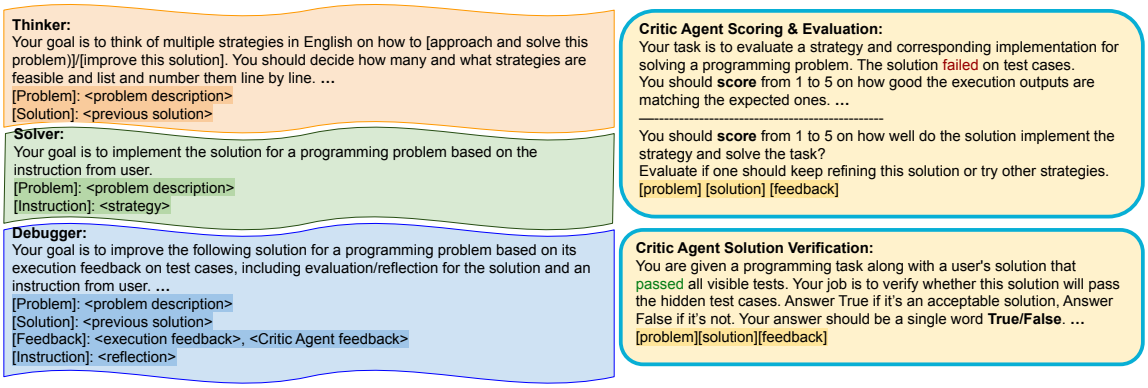


Figure 3.6: Simplified versions of instruction prompts used for Thinker, Solver, Debugger, and Critic agents. Some details are omitted for illustration purposes.

LLM "Thinker" agent to sequentially generate a set of high-level strategies given an input coding problem. Each strategy is generated autoregressively conditioned on previously generated strategies. By allowing models to first generate coding strategies, we enable LLMs to tackle coding problems using their reasoning capabilities learned from the text domain, which is answering **RQ3: We can always separate natural language reasoning and verbal solution generation from program implementation in each step of the pipeline.** The expressiveness of generated strategies in a natural language can potentially guide the code-generation process toward more diverse exploration paths. Notably, we let Thinker Agent dynamically decide the number of generated coding strategies, given the fact that different coding problems can have more or fewer feasible strategies.

Solution Generation with Solver Agent Given a complete generated strategy from Thinker Agent, we let a "Solver" agent LLM implement a set of initial code solutions. By including the strategy as part of the input instruction, we can condition Solver Agent to produce strategy-specific code candidates. From our previous research(Li et al., 2023; Li and Mooney, 2024), LLMs exhibit abilities to implement solutions based on verbal descriptions of solutions.

Solution Refining with Thinker & Debugger Agents Prior approaches such as (Chen et al., 2023a,b; Shinn et al., 2023; Madaan et al., 2023) found that syntactic mistakes or even logical flaws in generated code can be fixed by allowing LLMs to iteratively refine and regenerate the code. This self-refinement capability is typically strengthened by some forms of feedback about the code qualities (e.g. execution results, compiler signals): We name two types of feedback F_i, 1. execution results of the visible test set with the compiler's output and expected output. 2. Self-critique given by the critic agent.

Node Evaluation with Critic Agent CodeTree builds a heterogeneous tree for each problem, where the tree root represents a problem specification $(D, \{(i_j, o_j)\})$ and every subsequent tree node represents a generated candidate code solution Each node has relevant attributes including its collective code feedback F_i and its corresponding strategy and reflections: S_i and R_i. Typically, adding a tree node is a two-step process: 1) generate a code solution from the corresponding strategy, 2) evaluate the generated solution and obtain execution feedback. For a given solution and corresponding F_{exe}, Critic Agent performs an evaluation, measuring how promising it is, which results in F_{cri}. We separately evaluate how well: 1) the execution outputs of test cases match expected outputs on visible test cases; and 2) the solution robustly implements its corresponding strategy towards problem-solving. For one program and its corresponding feedback F_i, the Critic Agent will evaluate whether the current solution is worth refining, or it should not be explored, making decision between refinement and abort. For one solution that passes all visible test cases, it might potentially over-fit the visible test cases and could fail hidden test cases. Hence, the critic agent will verify if this solution is robust and generalizable to unseen test cases.

Tree Expanding with Critic Agent Unlike previous tree-structure search methods (Yao et al., 2024; Islam et al., 2024), we do not predefine the entire tree with fixed width and depth. Instead, we introduce a Critic Agent to dynamically expand

the tree based on potential strategies. It will guide the expansion and spanning of the tree, taking actions based on its evaluation of the current node. To guide the search for a correct solution, at each node, Critic Agent has an action space of three actions: **Refine**: Continue exploring from the current node by generating multiple reflections for this node; **Abort**: Prune this node due to its low evaluation score, and retrace the exploration to its sibling nodes; and **Accept**: Accept the current node as the final solution and terminate the search early.

Agent Collaboration Throughout the expansion of the tree, the task-specific agents collaborate with Critic Agent, utilizing its feedback, and follow its guidance to perform exploration. The flexibility of the tree expansion and search is determined by LLM agents' decision-making, e.g. determining the number of strategies and deciding the search path. During inference time, practically, we limit the number of exploration steps to avoid large computation overhead. Whenever a termination signal (i.e. to accept a code solution) is found or the maximum number of exploration steps is reached, a code candidate is selected based on its evaluation score.

3.3.3 Experimental Results

In this section, we aim to evaluate if CodeTree can efficiently search for the correct solution and which agent is playing the most crucial role.

Experimental Settings and Evaluation Metrics We applied pass@1(Chen et al., 2021) as our evaluation metric: only one code candidate can be selected and submitted for the final evaluation with hidden test cases. We set the generation budget to be 20 samples per coding task. To fairly compare our approach with other baselines, we adopted the same generation budget in all methods. For ablation experiments without using Critic Agent, we followed similar strategies from (Shinn et al., 2023; Chen et al., 2023b): we select a solution that passes all visible test cases as the final solution to be evaluated with hidden test cases. We conducted experiments on 2

categories of code generation tasks: 1) Function implementation where a coding task is to complete a single function following a specific function signature: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and their EvalPlus variants from (Liu et al., 2023), denoted as HumanEval+ and MBPP+; and 2) Program implementation where a coding task is to solve an algorithmic problem: CodeContests (Li et al., 2022b) and APPS (Hendrycks et al., 2021b). The sizes of test set are 164, 378 and 165 for HumanEval(+), MBPP(+) and CodeContests respectively. For APPS, we randomly sampled 150 samples (50 for each level of difficulty) from the test split.

We introduce the following baselines: **Direct** instructs the model to generate code directly from the input problem; **CoT** (Wei et al., 2022b) instructs the model to provide chain-of-thought reasoning before giving the solution program; **Reflexion** (Shinn et al., 2023) utilizes solution's execution feedback to generate self-reflections. The reflections are used to iteratively refine the solution; **MapCoder** (Islam et al., 2024) proposes an agent collaboration system to plan, solve, test, and refine the solution. We set #plans=4, #debug-round=5 and generation budget=20; and **Resample** follows a similar principle as Li et al. (2022b): resample solutions repeatedly and filter them with visible test cases.[4]

We studied our method on three models with different model sizes and capacities. We experimented on large language models from the GPT and Llama 3.1 family. Specifically we use **GPT-4o-mini**, **GPT-4o**[5], and **Llama-3.1-8B** [6].

Main Results We compared CodeTree with other baselines in Table 3.4. We noticed that Reflexion and Resampling serve as strong baselines for HumanEval and MBPP datasets given the same solution generation budget, comparable to CodeTree-BFS/DFS. CodeTree with Critic Agent outperforms all other baselines in 4 out

[4] We set sampling temperature=1 for Resample, and report the best results over 2 runs. For other methods, we report the single run's results with deterministic inference.

[5] https://openai.com/index/hello-gpt-4o/

[6] https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct. Note that we reported our replicated results which might be different from the original reported ones.

Model	Method	HumanEval	HumanE+	MBPP	MBPP+	Codecontests	Avg.
GPT-4o-mini	Direct	86.6%	78.7%	87.8%	73.3%	10.3%	67.3%
	CoT	84.8%	78.0%	89.2%	74.3%	12.7%	67.8%
	Reflexion	92.1%	83.5%	96.6%	**78.6%**	21.8%	74.5%
	MapCoder	91.5%	78.0%	90.0%	-	-	-
	Resample	89.0%	80.5%	94.3%	76.8%	18.2%	71.8%
	CodeTree-BFS	93.3%	82.1%	91.5%	72.3%	20.6%	72.0%
	CodeTree-DFS	92.7%	81.1%	87.6%	71.4%	20.6%	70.7%
	Strategy List	90.2%	80.5%	90.5%	69.6%	22.4%	70.6%
	CodeTree	**94.5%**	**84.8%**	**96.8%**	77.0%	**26.4%**	**75.9%**
GPT-4o	Direct	88.4%	81.7%	92.3%	75.9%	20.6%	71.8%
	CoT	92.1%	84.1%	93.7%	77.2%	24.8%	74.4%
	Reflexion	94.5%	84.8%	97.9%	79.6%	41.8%	79.7%
	MapCoder	92.7%	81.7%	90.9%	-	-	-
	Resample	93.9%	84.8%	96.2%	77.0%	32.7%	76.9%
	CodeTree-BFS	94.5%	84.1%	93.9%	70.7%	35.8%	75.8%
	CodeTree-DFS	**95.1%**	83.5%	91.5%	76.2%	36.4%	76.5%
	Strategy List	**95.1%**	82.3%	92.6%	73.3%	36.4%	75.9%
	CodeTree	94.5%	**86.0%**	**98.7%**	**80.7%**	**43.0%**	**80.6%**
Llama-3.1-8B	Direct	63.4%	54.3%	73.4%	63.8%	6.1%	52.2%
	CoT	65.9%	56.1%	74.6%	65.3%	4.2%	53.2%
	Reflexion	79.9%	69.5%	90.2%	72.0%	13.5%	65.0%
	Resample	**82.3%**	71.3%	**91.0%**	**73.8%**	**15.2%**	**66.7%**
	CodeTree-BFS	80.5%	68.3%	91.0%	69.3%	15.8%	65.0%
	CodeTree-DFS	80.5%	68.9%	89.7%	70.4%	15.2%	64.9%
	Strategy List	82.3%	70.1%	**91.0%**	72.5%	13.9%	66.0%
	CodeTree	82.3%	**72.0%**	90.5%	73.3%	12.1%	66.0%

Table 3.4: Experimental results by pass@1 on HumanEval, MBPP, EvalPlus, and CodeContests: methods are baseline methods that generate program solution only once, are methods with solution generation budget of 20 samples like our methods. are CodeTree variants with or without Critic Agent to guide the tree search. Note that MapCoder does not work with Llama-3.1-8B as noted by Islam et al. (2024).

Model	**GPT-4o-mini**	
Benchmark	HumanEval	HumanEval+
CodeTree	94.5%	84.8%
w/o verification	91.5%	81.7%
w/o node abort	91.5%	81.1%
w/o scoring	92.7%	82.1%

Table 3.5: Ablation study for different tasks of Critic Agent. We used GPT-4o-mini to evaluate corresponding settings and reported the pass@1 on the HumanEval and HumanEval+ benchmarks.

of 5 benchmarks for GPT-4o-mini and GPT-4o. For instance, CodeTree achieves pass@1=43.0% on competition-level coding tasks in the Codecontests benchmark (i.e. 22.4% performance gain over the Resampling baseline), showing its advantage in solving hard problems.

We found that CodeTree-BFS almost always performs better than CodeTree-DFS, suggesting that exploring diverse strategies is more effective than iteratively refining from one solution. Interestingly, on Llama-3.1-8B model, Resampling achieves the best results on 4 benchmarks. This observation indicates that small language models may not be suitable for multi-agent frameworks like CodeTree, where models are required to follow task-specific roles and instructions and perform distinct tasks with reasonable accuracy.

Ablation Study Results in Table 3.4 indicate, Critic Agent plays a crucial role in CodeTree over naive search methods like BFS/DFS. We further analyzed which task is the most crucial for Critic Agent. Specifically, we conducted the following ablation experiments: (1) w/o Solution Verification, where we excluded the verification task for any solution passing visible tests; (2) w/o Node Abort Evaluation, where we let the agents keep exploring till we reach the max depth or whenever a solution is accepted; (3) w/o Node Scoring, where the environmental feedback is solely execution outputs, without Critic Agent's evaluation.

The results in Table 3.5 show that all proposed tasks are crucial for Critic Agent to guide the tree expanding and solution search. Among these tasks, node abort and solution verification tasks are the most effective and have significant impacts on final performances.

3.3.4 Conclusion

We introduce CodeTree, a new framework of agent-guided tree search for code generation tasks. Introducing a tree-based structure as a unified search space, CodeTree includes a Critic agent to guide the tree search and make critical decisions such as termination, expanding and scoring of tree nodes. CodeTree facilitates multi-agent collaboration (among Thinker, Solver, and Debugger agents) to find the correct solution within a limited solution generation budget. In this paper, we leverage LLMs' natural language reasoning capabilities in a zero-shot manner. This is largely reflected in Thinker Agent who gives distinct problem-solving or code fixing strategies, and Critic agent, who scores, verifies and acts on a node of a candidate solution.

3.4 AlgoSimBench: Identifying Algorithmically Similar Problems for Competitive Programming

3.4.1 Overview

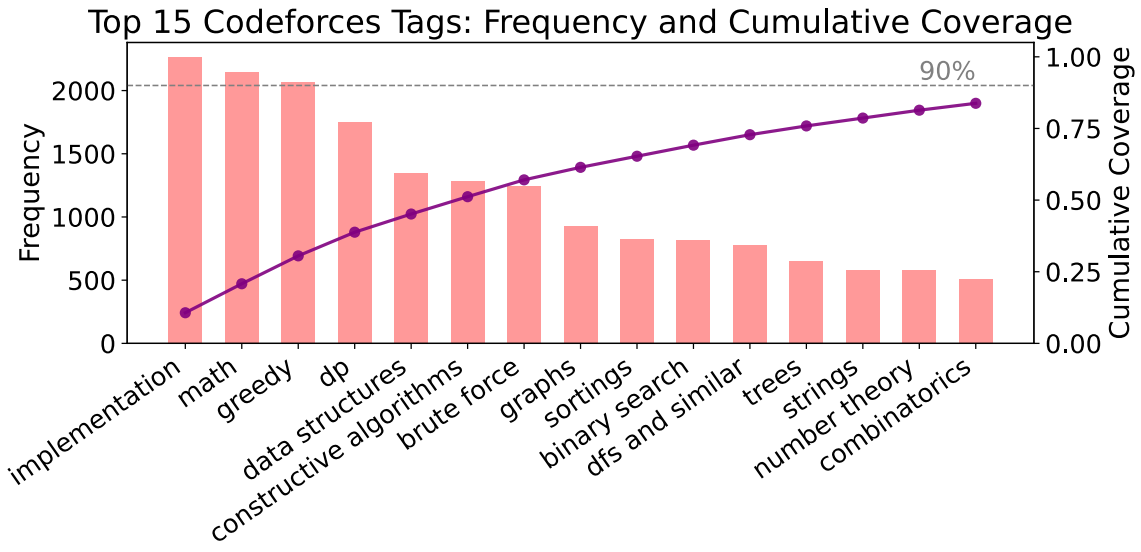
Recent progress in LLMs, such as reasoning models, has demonstrated strong abilities to solve complex competitive programming problems, often rivaling top human competitors. However, it remains underexplored whether these abilities generalize to relevant domains that are less seen during training. To address this, we introduce AlgoSimBench(Li and Mooney, 2025), a new benchmark designed to assess LLMs' ability to identify algorithmically similar problems (ASPs)—problems that can be solved using similar algorithmic approaches. AlgoSimBench consists of 1317 problems, labeled with 231 distinct fine-grained algorithm tags, from which we curate 402 multiple-choice questions (MCQs), where each question presents one algorithmi-

cally similar problem alongside three textually similar but algorithmically dissimilar distractors. Our evaluation reveals that LLMs struggle to identify ASPs, with the best-performing model (o3-mini) achieving only 65.9% accuracy on the MCQ task. To address this challenge, we propose *attempted solution matching* (ASM), a novel method for improving problem similarity detection. On our MCQ task, ASM yields an average of 8.6% absolute accuracy improvement across different models. AlgoSimBench can also serve as a probing task for code retrieval methods. We also evaluate code embedding models and retrieval methods on similar problem identification. Combining ASM with a keyword-prioritized method, BM25, can yield up to 52.2% accuracy.

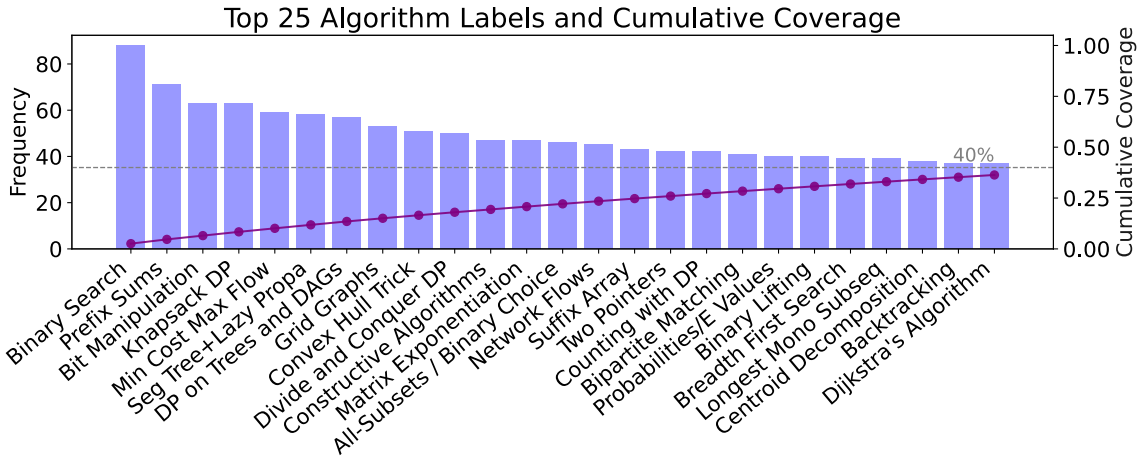
3.4.2 Dataset Curation

Many competitive programming platforms (e.g., Codeforces, CodeChef) provide algorithm tags, but these are usually too general (e.g., dynamic programming) to capture the solution strategy. To identify ASPs, we need more specific tags such as Longest Increasing Subsequence or 0/1 Knapsack, characterizing a specific technique needed to solve the problem. A comparison between the distributions of AlgoSimBench's and Codeforces' tags are given in Figure 3.7. To obtain problems annotated by expert programmers with fine-grained labels, we found four communities that have collected problems with detailed algorithm and topic tags. We retain labeled problems originally from Codeforces, CodeChef, and AtCoder, where formatted problem statements and corresponding human solutions are available. We manually unified labels from these different sources and removed duplicates and overly broad or ambiguous tags to assemble 319 distinct fine-grained labels. We collected 1522 tagged problems, filtered to 1,317 clean ones, 903 of which appear as the given problem or multiple-choice options in AlgoSimBench.

As Jain et al. (2021); Wei et al. (2025) found, language and code models are sensitive to functionality-preserving attacks and struggle to disentangle program semantics from natural language semantics. Suresh et al. (2025) also describes hard



(a) Top 15 Codeforces broad algorithm tags.



(b) Top 25 fine-grained labels in our dataset.

Figure 3.7: Comparison between Codeforces tag distribution and our fine-grained tag distribution.

negatives as those semantically similar to positive cases but which do not directly address a retrieval query.

Competitive programming problems usually contain a background story, connecting the algorithms to a real-life situation. We intentionally invert the correlation between algorithmic similarity and textual similarity. That is, algorithmically similar problems should differ as much as possible in wording and surface descriptions, while dissimilar problems should appear similar in text but differ in algorithmic structure. Specifically: The similar option shares the closest match in problem topics and algorithmic approach but differs significantly in text and overall linguistic meaning. The distractors belong to different algorithmic categories but are chosen to be textually close to the original problem. The idea underlying this setting is that *models should be able to ignore features irrelevant to problem solving, and base its judgment of algorithmic similarity solely on the problem structure.* We obtain a set of problems, each with a problem statement, a corresponding solution program, and a set of algorithmic labels. We create multiple-choice questions as follows: Given a reference problem, we find an algorithmically similar problem with the exact same set of labels, and three algorithmically dissimilar problems. With the Contriever model(Izacard et al., 2022) as the scorer for similarity, we ensure textually-similar distractors and textually-dissimilar ASP following the steps: a). Select the ASP with the lowest similarity score, b). Select the distractors with the highest similarity scores that are higher than the score in the previous step. c) Human-verify the validity of options. d) If not valid, go back to a) or b) to pick substitute(s). For problems where we cannot form a valid 4-option, we exclude it from AlgoSimBench MCQ.

3.4.3 Method

Episodic Retrieval (Shi et al., 2024) uses similarity among solution programs to identify similar problems, but it compares LLM-generated code for the query problem with oracle-solution code for the potential retrieval targets. This assumes that the initial solution contains the correct algorithmic signal. However, this breaks down when

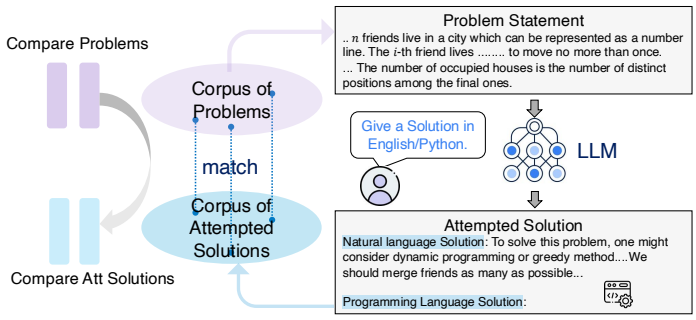


Figure 3.8: Illustration of Attempted Solution Matching. Attempted solutions are generated by LLMs, forming a matching corpus for better ASPs identification.

the LLM selects the wrong approach, e.g., misclassifying a dynamic programming problem as greedy-search. Such errors can accumulate, decreasing LLMs' capacity to identify algorithmically similar problems.

This motivates our approach: rather than comparing problem statements directly, we propose Attempted Solution Matching (ASM)—a method that prompts an LLM to generate a solution attempt for each problem, and then compares these attempted solutions to identify ASPs. We illustrate this method in Figure 3.8.

In ASM, each problem statement (i.e., problem description) is first mapped to a first-attempt solution using an LLM, and ASPs are identified by evaluating the similarities between attempted solutions.

The attempted solution can take two forms: A natural language explanation of the solution strategy (NL-solution), denoted as ASM-NL; or a program written in code (PL-solution) ASM-PL. While both should contain algorithmic information, ASM-NL can include content on problem analysis, strategy exploration, algorithm selection, and problem solving; on the other hand, ASM-PL focuses on the complete implementation, whose content is rich in details but insufficient in analysis.

3.4.4 Experiments

Experimental Settings For End-to-End Selection(i.e., model generates the option with the full MCQ in the instruction), we evaluate 7 open and closed SoTA LLMs: GPT-4o-mini, GPT-4o[7], o3-mini-medium[8], Deepseek-R1 (DeepSeek-AI et al., 2025a), Deepseek-V3 (DeepSeek-AI et al., 2025b), Claude-3.5-Sonnet[9] and Gemini 2.0 Flash.[10] For Retrieval-Based Selection(i.e., with the given problem as query and four options as corpus, a model retrieve the most similar one from the corpus), we tested various metrics for measuring the similarity between the query and candidate answers. We used cosine similarity of dense embeddings from a text embedding model BART (Lewis et al., 2020a) and a code embedding model GraphCodeBert (Guo et al., 2021), and a sparse retriever, BM25 (Robertson et al., 1995). We compare ASM to using the following baseline problem representations. **Statement**: Original natural-language descriptions are used to represent problems, **Summary**: To mitigate aspects unrelated to problem-solving, an LLM is first prompted to summarize and abstract the problem, minimizing narrative elements and formatting details irrelevant to the structure of the problem. **Solution:** Each problem is replaced with a correct solution written by an expert programmer, which serves as an "oracle" upper baseline.

End-to-End Selection MCQ accuracies of all seven models using End-to-End selection are given in Table 3.6. We found that none of the models perform particularly well when only given the original problem Statements. Reasoning LLMs like Deepseek-R1 and o3-mini perform better than other models, achieving accuracies around 60%. Given oracle solutions (**Solution**) instead of problem statements, LLMs can identify algorithmic similarity much better, yielding improvements as high as 26.3%.

[7] https://openai.com/index/hello-gpt-4o/
[8] https://openai.com/index/openai-o3-mini/
[9] https://www.anthropic.com/news/claude-3-5-sonnet
[10] https://deepmind.google/technologies/gemini/flash/

Model	Statement	Summary	ASM-NL	ASM-PL	Solution*
GPT-4o-mini	35.5	35.8	**43.8 (↑ 8.3)**	42.5 (↑ 7.0)	54.4
GPT-4o	41.5	38.1	**53.2 (↑ 11.7)**	53.0 (↑ 11.5)	63.4
o3-mini-medium	65.9	63.4	74.4 (↑ 8.5)	**75.1 (↑ 9.2)**	72.6
Deepseek-R1	63.7	57.7	69.2 (↑ 5.5)	**70.4 (↑ 6.7)**	70.6
Deepseek-V3	55.2	53.2	**64.9 (↑ 9.7)**	62.7 (↑ 7.5)	66.7
Claude-3.5-Sonnet	44.2	45.0	**54.7 (↑ 10.5)**	53.0 (↑ 8.8)	70.5
Gemini 2.0 Flash	51.2	48.5	**57.9 (↑ 6.7)**	55.5 (↑ 4.3)	69.7
Avg	50.4	48.8	59.5 (↑ 9.1)	58.5 (↑ 8.1)	66.5

Table 3.6: End-to-End Selection Performances(Accuracy%) on AlgoSimBench-MCQ over different models and methods. Results in bold indicate the best performing non-oracle method for each model. Absolute performance gain over the **Statement** baseline is marked with ↑. *Solution* is an oracle-input setting where oracle solutions were directly provided to the model.

Summarizing problems actually hurts the performance of most models except for GPT-mini-4o and Claude-3.5-Sonnet. We hypothesize that SoTA LLMs can already ignore superficial similarities when comparing programming problems. Our methods, **ASM-NL** and **ASM-PL**improve absolute performances from 4.3% to 11.7% across all models. **ASM-NL** outperforms **ASM-PL** on 5 out of 7 models, indicating that most models can generate and compare solutions described in natural language better than actual code solutions. For o3-mini-medium, both ASMs even outperform the oracle-solution baseline and give the best performance overall. The performance gaps between **Statement** and **ASM** also suggest that LLMs cannot generalize their code reasoning ability to similar tasks when they are not formatted as the training tasks.

Retrieval-Based Selection In this setting, we treat AlgoSimBench as a similarity ranking problem: given a reference problem, rank options by their algorithmic similarity, which is scored by a retrieval method. LLMs map statements in to **Summary**, **ASM-NL** and **ASM-PL**. We evaluate how well different retrieval strategies—including ASM—perform at identifying the correct ASP. Table 3.7 shows results

for three similarity metrics, BM25, BART, and GraphCodeBert (GCB). For dense embedding models, we calculate the cosine similarity of the embeddings of two text sequences to rank options.

	Summary			**ASM-NL**			**ASM-PL**		
	BM25	BART	GCB	BM25	BART	GCB	BM25	BART	GCB
GPT-4o-mini	25.6	23.4	20.6	**35.3**	29.1	22.4	34.8	26.1	32.8
GPT-4o	25.8	24.6	26.1	**42.5**	30.1	32.1	35.6	28.8	29.1
o3-mini-medium	39.3	32.6	31.3	**49.0**	35.6	38.3	48.8	28.4	39.3
Deepseek-V3	29.9	26.1	22.4	**46.0**	33.1	32.1	45.5	32.3	35.1
Deepseek-R1	30.1	24.1	25.9	**52.2**	32.8	31.1	45.0	32.8	35.9
Gemini-2.0-Flash	27.1	24.4	18.6	**41.0**	34.3	26.6	38.0	36.8	35.6
Claude-Sonnet-3.5	29.4	25.1	25.4	**39.6**	28.1	28.6	35.0	29.1	29.9
Avg	29.6	25.8	24.3	**43.7**	31.9	30.2	40.4	30.6	34.0

Table 3.7: Accuracy(%) comparison across different LLMs and retrieval methods in the Retrieval-Based Selection setting.

ASM-NL again outperforms other methods. Using LLM problem summaries at least performs better than random guessing. However, retrieval based directly on problem statements performs worse than random since AlgoSimBench is constructed to make textual problem-statement similarity a misleading indicator of ASPs. An interesting finding is that simple term-matching (BM25) actually performs better than dense embedding models. A possible reason for this is that features indicating algorithmic similarity are often keywords like algorithmic terms or key descriptions of the core idea, rather than a complete solution with implementation details.

ICL Exemplar Selection w/ASM As ASM can help find problems that are algorithmically more similar, we apply ASM to enhance code generation. Shi et al. (2024) showed that including in-context exemplars of solved problems can enhance LLMs' abilities to solve competitive programming problems. They proposed Episodic Retrieval, using an LLM's solution along with the problem statement to retrieve the most similar human solutions from a corpus using BM25. Then, the retrieved problem is

Exemplar Selection Method	GPT-4o-mini	GPT-4o
w/o Exemplar	9.4%	17.3%
Retrieve w/Random	10.4%	17.9%
Retrieve w/Statement	11.4%	16.9%
Episodic Retrieve	12.4%	18.6%
Retrieve w/ASM-NL	**13.7%**	19.2%
Retrieve w/ASM-PL	13.0%	**19.9%**

Table 3.8: ICL enhanced by different exemplar selection methods. Results are Pass@1 on the USACO Benchmark with one exemplar in the context.

inserted as an ICL examplar, hopefully improving the ability to solve the given problem. ASM, on the other hand, directly matches LLM attempted solutions for both the given problem and problems in the corpus. Utilizing their methodology, we explored how ASM could be used for better selection of ICL exemplars. We also applied two baselines that select either a *Random* exemplar or retrieve the most similar exemplar using problem statements. Table 3.8 shows that with the same Retriever (BM25, which outperformed various dense-embedding methods), ASM methods achieve the best pass@1 performance. While the absolute improvement is modest, this might be limited by how much performance gain ICL with a single example can bring to competitive programming (Tang et al., 2023a; Patel et al., 2024).

3.4.5 Conclusion

This section introduces AlgoSimBench, a new benchmark designed to evaluate models' ability to reason about algorithmic similarity between competitive programming problems. This dataset serves two key purposes. First, it provides a focused evaluation of the algorithmic reasoning abilities of LLMs, decoupled from the full generation of solutions. Second, it enables the study of retrieval methods that go beyond surface-level textual similarity, instead capturing deeper structural and problem-solving-related semantics. We also investigate **RQ5: Does a model's ability to solve specific problems reflect a deeper, generalizable understanding of the**

underlying algorithms? While problem-solving is often viewed as a joint capability involving problem understanding, algorithm identification, reasoning, strategy development, and code implementation, our results suggest that these components do not always generalize or decompose cleanly in new contexts.

Chapter 4: Proposed Works

4.1 Instruction-aware Code Embedding

Following the findings from AlgoSimBench, we aim to improve the language models' performance as dense retrievers for specified tasks like identifying algorithmically similar pairs.

4.1.1 Motivation

The Attempt Solution Matching (ASM) component of AlgoSimBench currently relies on a large language model (LLM) to rewrite user queries in order to improve retrieval quality. However, this approach is computationally expensive and does not scale well to large corpora. Interestingly, we observe that simple term-matching methods such as BM25 often outperform dense embedding models in retrieving algorithmically similar problems or code snippets. This counterintuitive result raises the question of why dense retrieval methods underperform in this setting compared to traditional sparse retrieval.

One explanation lies in the fundamental differences between sparse and dense methods. BM25 is keyword-focused: it ranks documents by term frequency–inverse document frequency (TF–IDF) weighting (Robertson and Zaragoza, 2009), giving high importance to rare but discriminative tokens such as algorithm names ("Dijkstra," "FFT"), mathematical operators, or domain-specific jargon. In contrast, dense retrieval methods map entire sentences into continuous vector representations using neural encoders, typically by mean pooling over token embeddings or using the final hidden state of a special token (e.g., [CLS]) (Wang et al., 2024; Izacard et al., 2022). These embeddings are designed to capture holistic semantic meaning rather than emphasize localized, task-specific signals. Furthermore, current embedding models are trained on mixed-domain data without explicit task conditioning; robustness

across tasks depends largely on how positive and negative pairs are constructed during training (Jain et al., 2021; Gao et al., 2022; Suresh et al., 2025), not on architectural mechanisms for disambiguating retrieval intents.

A plausible hypothesis is that features critical for identifying algorithmic similarity are concentrated in a small set of discriminative tokens—such as algorithm names, mathematical terms, or concise descriptions of the core idea—rather than in the broader contextual or implementation details of a solution. In contrast, dense retrieval methods based on LLM-derived embeddings assume that a sentence (or document) can be represented by a single global embedding vector, typically obtained via mean pooling or last-token representations. These embeddings are designed to capture holistic semantic meaning, not task-specific signals. As a result, current dense models may dilute or obscure the fine-grained, keyword-level distinctions that are crucial for algorithmic retrieval tasks. Moreover, in domains such as code retrieval, the target notion of similarity is inherently multi-faceted: queries may seek functionality similarity, algorithmic similarity, useful references (e.g., function documentation), or stylistic similarity (e.g., comments). Without task-specific conditioning, embedding models cannot easily disambiguate these different intents.

4.1.2 Related Works

Recent progress in LM-based text embeddings has focused on several complementary dimensions. First, pooling strategies such as mean pooling and last-token pooling remain the most widely used methods for aggregating token-level representations into fixed-length embeddings, though more recent work explores hybrid or trainable pooling layers(Lee et al., 2025). Second, contrastive learning(Gao et al., 2022) has become the standard training objective, where models are optimized to bring positive pairs closer while pushing apart negatives, with effectiveness highly sensitive to the choice of negatives. This connects to a third line of research on dataset design, where methods for hard negative mining—such as sampling semantically close but non-relevant examples or filtering with teacher models—have been

shown to significantly improve retrieval performance. SoTA code embedding models are still developed on the assumption that "high textual similarity indicates more relevant code" (Guo et al., 2021; Suresh et al., 2025).

Many recent text embedding methods took advantage of LLMs and their ability to follow instructions to extract text embeddings from causal LLMs by turning off causal mode and fine-tuning (Lee et al., 2025), a secondary task(Feng et al., 2025), appending instruction prompts (Su et al., 2023a; Zhang et al., 2025; Kryvosheieva et al., 2025), including few-shot examples (Li et al., 2024) and jointly training representation tasks with generation tasks (Muennighoff et al., 2025). These works are a big leap by distinguishing different tasks. However, 1). they still bind any text piece to a specific task, which might not be the real-world scenario. 2). the methods raise limitations to generalize to unseen tasks like AlgoSimBench without in-domain training.

Instead, we want to propose an efficient embedding method that utilizes 1) the fact that instructions are natural language and 2) an explicit model structure to learn the mutual information between the instruction and the text to represent.

4.1.3 Method

While exploring different methods to train an LLM, the choice of training data and LLMs is also important. Qwen3-Embedding(Zhang et al., 2025) is the open, SoTA, instruction-tuned embedding model, which could be our first option. We use SimCSE(Gao et al., 2022) as the training objects.

4.1.3.1 Cross-Attention Pooling between text and instruction

Chen et al. (2018) proposed attention pooling for transformer-based models. Conceptually, attention weights assign relevance or importance to tokens. Such a design could be useful to highlight "keywords" for different tasks. Thus, we focus on cross-attention pooling, a mechanism that allows the embedding to be calculated

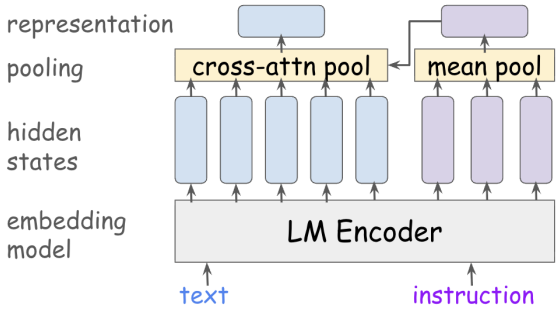


Figure 4.1: Overview of Cross-Attention Pooling for Instruction-aware Code Embedding with a joint LM Encoder to embed text and instruction hidden states.

based on *how much attention the instruction assigns to different text tokens*. The final representation of text should be a weighted sum of hidden states of all tokens in the text. The overview of the method is given in 4.1.

Let the input text be a sequence of tokens $T = \{t_1, t_2, \ldots, t_n\}$ and the instruction be a sequence of tokens $I = \{i_1, i_2, \ldots, i_m\}$. Our goal is to compute an instruction-aware embedding for the text, $e_{T|I}$.

First, we feed both the text and the instruction through a shared pre-trained transformer-based encoder (e.g., BERT or GPT) to obtain their token-level hidden state representations. $H_T = \{h_1, h_2, \ldots, h_n\}$, where $h_j \in \mathbb{R}^d$ is the hidden vector for token t_j. The hidden states for the instruction are denoted as $H_I = \{h'_1, h'_2, \ldots, h'_m\}$, where $h'_k \in \mathbb{R}^d$ is the hidden vector for token i_k. Here, d is the dimensionality of the hidden states. To guide the pooling of text representations, we first create a single vector representation for the instruction. This is achieved by mean-pooling its token hidden states. This vector will serve as the Query $Q = \frac{1}{m}\sum_{k=1}^{m} h'_k$, $Q \in \mathbb{R}^d$ in our attention mechanism. We use the standard scaled dot-product attention mechanism, where the instruction's query vector attends to the text's hidden states. The text's hidden states, H_T, serve as both the Keys $(K)$and Values(V).

The final text embedding, $e_{T|I}$, is computed as the weighted sum of the text's hidden states (the values), using the attention weights:

$$e_{T|I} = \sum_{j=1}^{n} \text{Softmax}(\frac{Q \cdot h_j^T}{\sqrt{d}}) h_j$$

This resulting vector $e_{T|I} \in \mathbb{R}^d$ is a representation of the text that is dynamically conditioned on the instruction. By weighting tokens based on their relevance to the instructional query, the model can effectively highlight "keywords" or crucial features for the specific task at hand, thus explicitly modeling the mutual information between the instruction and the text.

4.1.3.2 Learn Transformations from General to Intruction-Aware

Another alternative is to learn general embeddings and instruction-specified embeddings simultaneously. We propose to learn transformations of the general text embedding conditioned on the prompt. The embedding model is learning a generic, universal embedding e_u and a list of tasks $\{t_1, t_2, \cdots, t_n\}$(i.e., task instruction) with corresponding embeddings$\{e_{t1}, e_{t2}, \cdots, e_{tn}\}$, so that the task-specific embedding can be calculated by $\tilde{e_u} = e_u + \Delta(e_u, e_{ti})$ where $e \in \mathbb{R}^d$. The actual method or instruction-transformation is still to be decided with many other options:

- Additive Shift: the general embedding of a text piece is applied a small, instruction-conditioned update by: $\tilde{e_u} = e_u + \Delta(e_u, e_{ti})$ where $e \in \mathbb{R}^d$.
- Bilinear Interaction: We learn not only the parameters of LLMs, but also of this transformation itself, $\Delta(e_u, e_{ti}) = W\big((Be_u) \odot (Ce_{ti})\big)$, through a second-order bilinear interaction.
- Latent Attention Shift: We can also use the Latent attention layer from (Lee et al., 2025), instead of a universal latent array to produce the values of hidden states V, we might make it conditioned on an instruction.

4.1.4 Evaluation

For code embedding, we can use the public training data and benchmarks to evaluate models as used in Kryvosheieva et al. (2025). Our methods can be potentially applied to text embedding tasks. We can train it on public retrieval datasets utilized by Zhang et al. (2025); Lee et al. (2025) and to evaluate it on the massive text embedding benchmark (MTEB) (Muennighoff et al., 2023) leaderboard [1]. We will also evaluate it on our AlgoSimBench for identifying algorithmically similar pairs of problem descriptions or programs.

4.2 Training Embeddings from Downstream Code Generation Feedback

In this proposed work, we approach enhanced code embedding with a specific purpose: to improve the performance of Retrieval-Augmented Code Generation (RAG-CodeGen). Instead of code-NL or code-code parallel training data, we utilize the performance of code generation as the signal to train an embedding model.

4.2.1 Motivation

Wang et al. (2025) has found that retrievers struggle to fetch relevant and helpful context for code generation. A central challenge in retrieval-augmented code generation lies in defining what constitutes "helpful" retrieved content. Helpfulness is inherently multifaceted: depending on the query and downstream task, different forms of evidence may be useful, ranging from direct code snippets to conceptual explanations or related APIs. Manually annotating such judgments is both expensive and ambiguous, since the same piece of content may vary in utility across contexts.

Traditional approaches to training embeddings rely on supervised pairs—either code-to-code or code-to-NL—which assume a fixed notion of relevance. However,

[1] https://huggingface.co/spaces/mteb/leaderboard

these signals fail to capture the dynamic, task-specific notion of usefulness in code generation. A snippet that is semantically similar to the query may still be unhelpful for generation if it does not align with the model's decoding process, while a seemingly distant piece of content could enable the model to complete a solution more effectively. Thus, conventional supervised training leaves a mismatch between embedding similarity and generation success.

By contrast, downstream generation feedback provides a natural, task-grounded measure of helpfulness. The ultimate test of a retriever is whether the retrieved evidence improves code generation quality, not whether it satisfies a static similarity metric. Using generation outcomes as feedback transforms embedding training into a process that is both adaptive and context-sensitive: the model learns to prioritize content that actually advances problem-solving, even if that deviates from superficial similarity.

This perspective motivates our work: to move beyond proxy similarity objectives and instead let the code generator itself define the training signal for retrieval. Doing so not only addresses the limitations of manual annotation but also aligns embedding learning directly with the downstream task of effective retrieval-augmented generation.

4.2.2 Related Works

Recent progress in applying Reinforcement Learning (RL) has transformed RAG from static pipelines into dynamic, self-optimizing frameworks. Milestones include training agents to decide when to retrieve for cost-efficiency (Kulkarni et al., 2024), optimizing what to retrieve by rewriting queries(Gao et al., 2025), and training generators to handle noisy contexts(Huang et al., 2025). The current state-of-the-art has moved towards holistic optimization, either by jointly training the entire pipeline or by incorporating it as a team of collaborative agents(Chen et al., 2025b). These advancements, however, have been predominantly validated on natural language tasks

like open-domain question answering, where reward signals are based on textual similarity and factual correctness.

Despite these advances, the direct application of existing RAG+RL methodologies to the code domain is challenging. Beyond general RL hurdles like sample inefficiency and complex reward engineering, the core limitation is a mismatch in the definition of "helpfulness." Current reward functions are tailored for natural language, using metrics like F1 scores or semantic similarity. In contrast, the utility of a retrieved code snippet is determined by its functional correctness, syntactic validity, and logical compatibility, for which textual similarity is a poor proxy. Code generation offers a powerful and objective reward signal through execution feedback—whether the code compiles and passes unit tests.

4.2.3 Method

To address this, we plan to explore leveraging the downstream performance of a code generation model as a reward signal for retrieval training. The method itself is similar to (Lu and Liu, 2024; Yan and Ling, 2025). We adapt a different setting, instead of relying solely on static annotations, 1. Instead of hand-crafted preference, our reward signal is given by executing the code against test cases, which is the gold evaluation for correctness, 2. The retriever is encouraged to explore candidate contexts (e.g., by occasionally selecting from beyond the top-ranked results), appending them to the generator's prompt, and observing whether the augmented input improves code generation quality—such as increased test pass rate or reduced generation loss. Instances that yield measurable improvements can then be treated as implicit positive supervision, while less effective alternatives provide contrastive signals.

This approach naturally transforms retrieval training into a feedback-driven process where "helpfulness" is defined operationally by its impact on the generator's success. In effect, the embedding model learns to capture similarity not merely in

surface-level semantics, but in terms of downstream utility for code generation tasks.

An even more ambitious direction is to move beyond retriever optimization alone and consider a co-training framework in which both the retriever and the generator are updated in tandem(Lu and Liu, 2024; Le et al., 2025). While the retriever can be shaped to prefer content that empirically improves task outcomes, the generator itself may also learn to better interpret and leverage the retrieved evidence. By allowing both components to adapt iteratively, the system has the potential to converge toward a cooperative equilibrium: the retriever surfaces content tailored to the generator's evolving strengths, and the generator becomes increasingly skilled at grounding its outputs in the retrieved material. Such a joint optimization paradigm remains largely underexplored, but it opens a promising avenue for making retrieval-augmented generation more robust and context-sensitive across diverse problem settings.

4.2.3.1 Evaluation

The proposed method is designed for situations where the content of gold "helpfulness" is infeasible. Existing benchmarks that are built for RAG-CodeGen can be a great source to train and evaluate our method (Wang et al., 2025; Chen et al., 2025a). In addition, Gupta et al. (2025) curated code generation tasks in a retrieval-augmented setting. Alternatively, benchmarks like Jimenez et al. (2024) rely their performance heavily on the quality of repo-level retrieval, thus we might test if our method can improve the performance by retrieving more related information, combined with existing agentic(Antoniades et al., 2025) or agentless(Xia et al., 2024) frameworks. Potentially, if trained on ICL-enhanced CP programming problems(Li et al., 2022b; Shi et al., 2024), embeddings models are expected to perform better on AlgoSimBench.

Chapter 5: Conclusion

5.1 Summary of Works

The core focus of this proposal is to enhance competitive-level code generation by utilizing natural language reasoning. The first three works aim at identifying the reasoning gap between natural language and programming language, utilizing natural language reasoning as a bridge towards better CP code generation. The last completed work and future works will focus more on generalizing capacities across different forms of reasoning.

Initially, we evaluated the abilities of Large Language Models (LLMs) in competitive programming. We found that while LLMs struggle to solve problems directly, they are proficient at explaining human-written solutions and translating those verbal explanations back into correct code. Our experiments further revealed that the more detailed the explanation, the more accurately LLMs can implement the corresponding solution.

Based on this observation, we proposed synthesizing problem-solving Chain-of-Thought (CoT) by having LLMs explain human-written code. This method distills a model's explanatory capabilities into the generation of editorial-style CoTs, which outperforms training directly on code and improves problem solve rates. Our experiments also show that training only on the most crucial steps of the CoT yields better performance than training on the entire chain. These findings highlight that natural language can serve as an effective bridge to code generation when used as a preceding reasoning step.

Following the core idea of decomposing symbolic reasoning into natural language reasoning and code implementation, we investigated whether this could be achieved in a training-free framework. We proposed CodeTree, an agentic workflow for code generation. This system separates high-level strategy from implementation,

assigning problem reasoning to a "Thinker Agent" that uses natural language to devise distinct strategies. "Coder" and "Debugger" agents then implement the program and fix bugs based on the Thinker's guidance. A "Critic Agent" oversees the process by scoring, expanding, and terminating a tree search for the final solution. Our method, using GPT-4o, yielded better performance than the then-state-of-the-art models while using fewer tokens.

With the emergence of more advanced LLMs and post-training methods, major improvements have occurred in code generation for competitive programming. However, it is less explored whether reasoning learned from problem-solving can generalize to highly relevant tasks. Therefore, we introduced AlgoSimBench, a curated and verified dataset to test a model's ability to identify algorithmically similar problems from a set of semantically similar distractors. The adversarial nature of the benchmark makes this task extremely challenging. To address this, we proposed Attempted Solution Matching (ASM) and found that its application brings performance gains to both end-to-end selection via LLM prompting and to retrieval-based selection methods.

Two future works and potential methods are proposed in the previous chapter. The first work proposes to learn general embeddings and embedding transformations to shift general embeddings towards any specified task. The second works aim at exploring co-training for the retriever and generator for retrieval-augmented code generation(RAGCode).

5.2 Proposed Timeline

- Sep 2025, Thesis Proposal
- June 2025, Finish Proposed Works
- Aug 2025, Thesis Defense

Works Cited

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=G7sIFXugTX.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback, 2022. URL https://arxiv.org/abs/2212.08073.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry,

Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Rina Diane Caballar and Cole Stryker. What is reasoning in ai?, 2025.

Pierre Chambon, Baptiste Roziere, Benoit Sagot, and Gabriel Synnaeve. Bigo(bench) – can llms generate code with controlled time and space complexity?, 2025. URL https://arxiv.org/abs/2503.15242.

Angelica Chen, Jérémy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez. Improving code generation by training with natural language feedback. *arXiv preprint arXiv:2303.16749*, 2023a.

Jingyi Chen, Songqiang Chen, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. When llms meet api documentation: Can retrieval augmentation aid code generation just as it helps developers?, 2025a. URL https://arxiv.org/abs/2503.15231.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Qian Chen, Zhen-Hua Ling, and Xiaodan Zhu. Enhancing sentence embedding with generalized pooling. In Emily M. Bender, Leon Derczynski, and Pierre Isabelle, editors, *Proceedings of the 27th International Conference on Computational Linguistics*, pages 1815–1826, Santa Fe, New Mexico, USA, August 2018. Association for Computational Linguistics. URL https://aclanthology.org/C18-1154/.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks, 2022.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023b.

Yiqun Chen, Lingyong Yan, Weiwei Sun, Xinyu Ma, Yi Zhang, Shuaiqiang Wang, Dawei Yin, Yiming Yang, and Jiaxin Mao. Improving retrieval-augmented generation through multi-agent reinforcement learning, 2025b. URL https://arxiv.org/abs/2501.15228.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2022. URL https://arxiv.org/abs/2009.14794.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Yu, Vincent Zhao, Yanping Huang, Andrew Dai, Hongkun Yu, Slav Petrov, Ed H. Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling

instruction-finetuned language models, 2022. URL https://arxiv.org/abs/2210.11416.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020. URL https://arxiv.org/abs/2003.10555.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL https://arxiv.org/abs/1901.02860.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang,

Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025a. URL https://arxiv.org/abs/2501.12948.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jiawei Wang, Jin Chen, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, Junxiao Song, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Litong Wang, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang,

Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qiancheng Wang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, Runxin Xu, Ruoyu Zhang, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Shuting Pan, T. Wang, Tao Yun, Tian Pei, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wanjia Zhao, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaokang Zhang, Xiaosha Chen, Xiaotao Nie, Xiaowen Sun, Xiaoxiang Wang, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xingkai Yu, Xinnan Song, Xinxia Shan, Xinyi Zhou, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, Y. K. Li, Y. Q. Wang, Y. X. Wei, Y. X. Zhu, Yang Zhang, Yanhong Xu, Yanhong Xu, Yanping Huang, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Li, Yaohui Wang, Yi Yu, Yi Zheng, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Ying Tang, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yu Wu, Yuan Ou, Yuchen Zhu, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yukun Zha, Yunfan Xiong, Yunxian Ma, Yuting Yan, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Z. F. Wu, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhen Huang, Zhen Zhang, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhibin Gou, Zhicheng Ma, Zhigang Yan, Zhihong Shao, Zhipeng Xu, Zhiyu Wu, Zhongyu Zhang, Zhuoshu Li, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Ziyi Gao, and Zizheng Pan. Deepseek-v3 technical report, 2025b. URL https://arxiv.org/abs/2412.19437.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2019.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. Kto: Model alignment as prospect theoretic optimization, 2024. URL https://arxiv.org/abs/2402.01306.

Yingchaojie Feng, Yiqun Sun, Yandong Sun, Minfeng Zhu, Qiang Huang, Anthony K. H. Tung, and Wei Chen. Don't reinvent the wheel: Efficient instruction-following text embedding based on guided space transformation, 2025. URL https://arxiv.org/abs/2505.24754.

Jingsheng Gao, Linxu Li, Ke Ji, Weiyuan Li, Yixin Lian, yuzhuo fu, and Bin Dai. SmartRAG: Jointly learn RAG-related tasks from the environment feedback. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OCd3cffulp.

Tianyu Gao, Adam Fisch, and Danqi Chen. Making pre-trained language models better few-shot learners, 2021. URL https://arxiv.org/abs/2012.15723.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings, 2022. URL https://arxiv.org/abs/2104.08821.

Gaël Gendron, Qiming Bao, Michael Witbrock, and Gillian Dobbie. Large language models are not strong abstract reasoners. In Kate Larson, editor, *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 6270–6278. International Joint Conferences on Artificial Intelligence Organization, 8 2024. doi: 10.24963/ijcai.2024/693. URL https://doi.org/10.24963/ijcai.2024/693. Main Track.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

Daya Guo, Shuo Ren, Suyu Lu, Junjie Feng, Duyu Tang, Nan Duan, Ming Zhou, and Sining Liu. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2021.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Dhruv Gupta, Gayathri Ganesh Lakshmy, and Yiqing Xie. Sacl: Understanding and combating textual bias in code retrieval with semantic-augmented reranking and localization, 2025. URL https://arxiv.org/abs/2506.20081.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: Retrieval-augmented language model pre-training, 2020. URL https://arxiv.org/abs/2002.08909.

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, 2021a.

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*, 2021b.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George

van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack W. Rae, and Laurent Sifre. Training compute-optimal large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes, 2023.

Wenyue Hua and Yongfeng Zhang. System 1 + system 2 = better world: Neural-symbolic chain of logic reasoning. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 601–612, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. URL https://aclanthology.org/2022.findings-emnlp.42.

Jerry Huang, Siddarth Madala, Risham Sidhu, Cheng Niu, Hao Peng, Julia Hockenmaier, and Tong Zhang. Rag-rl: Advancing retrieval-augmented generation via rl and curriculum learning, 2025. URL https://arxiv.org/abs/2503.12759.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning, 2022. URL https://arxiv.org/abs/2112.09118.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5954–5971, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.482. URL https://aclanthology.org/2021.emnlp-main.482/.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. URL https://arxiv.org/abs/2001.08361.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

Donald K. Knuth. *The TEXbook*. Addison-Wesley, 1984.

Daria Kryvosheieva, Saba Sturua, Michael Günther, Scott Martens, and Han Xiao. Efficient code embeddings from code generation models, 2025. URL https://arxiv.org/abs/2508.21290.

Mandar Kulkarni, Praveen Tangarajan, Kyung Kim, and Anusua Trivedi. Reinforcement learning for optimizing rag for domain chatbots, 2024. URL https://arxiv.org/abs/2401.06800.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Tien P. T. Le, Anh M. T. Bui, Huy N. D. Pham, Alessio Bucaioni, and Phuong T. Nguyen. When retriever meets generator: A joint model for code comment generation, 2025. URL https://arxiv.org/abs/2507.12558.

Chankyu Lee, Rajarshi Roy, Mengyao Xu, Jonathan Raiman, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. NV-embed: Improved techniques for training LLMs as generalist embedding models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=lgsyLSsDRe.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2020a.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc., 2020b. URL https://

proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

Chaofan Li, MingHao Qin, Shitao Xiao, Jianlyu Chen, Kun Luo, Yingxia Shao, Defu Lian, and Zheng Liu. Making text embedders few-shot learners, 2024. URL https://arxiv.org/abs/2409.15700.

Jierui Li and Raymond Mooney. Distilling algorithmic reasoning from llms via explaining solution programs, 2024. URL https://arxiv.org/abs/2404.08148.

Jierui Li and Raymond Mooney. Algosimbench: Identifying algorithmically similar problems for competitive programming, 2025. URL https://arxiv.org/abs/2507.15378.

Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. Explaining competitive-level programming solutions using llms, 2023.

Jierui Li, Hung Le, Yingbo Zhou, Caiming Xiong, Silvio Savarese, and Doyen Sahoo. CodeTree: Agent-guided tree search for code generation with large language models. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3711–3726, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.189. URL https://aclanthology.org/2025.naacl-long.189/.

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, December 2022a. doi: 10.1126/science.abq1158. URL https://doi.org/10.1126/science.abq1158.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022b.

Changshu Liu, Shizhuo Dylan Zhang, Ali Reza Ibrahimzada, and Reyhaneh Jabbarvand. Codemind: A framework to challenge large language models for code reasoning, 2024. URL https://arxiv.org/abs/2402.09664.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

Hanzhen Lu and Zhongxin Liu. Improving retrieval-augmented code comment generation by retrieving for generation, 2024. URL https://arxiv.org/abs/2408.03623.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning, 2023.

YINGWEI MA, Yue Liu, Yue Yu, Yuanliang Zhang, Yu Jiang, Changjian Wang, and Shanshan Li. At which training stage does code data help LLMs reasoning? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KIPJKST4gw.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Niklas Muennighoff, Nouamane Tazi, Loic Magne, and Nils Reimers. MTEB: Massive text embedding benchmark. In Andreas Vlachos and Isabelle Augenstein, editors, *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 2014–2037, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.eacl-main.148. URL https://aclanthology.org/2023.eacl-main.148/.

Niklas Muennighoff, Hongjin Su, Liang Wang, Nan Yang, Furu Wei, Tao Yu, Amanpreet Singh, and Douwe Kiela. Generative representational instruction tuning, 2025. URL https://arxiv.org/abs/2402.09906.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. https://openai.com/blog/chatgpt, 2023a.

OpenAI. Gpt-4 technical report, 2023b.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In

S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 27730–27744. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

Arkil Patel, Siva Reddy, Dzmitry Bahdanau, and Pradeep Dasigi. Evaluating in-context learning of libraries for code generation, 2024. URL https://arxiv.org/abs/2311.09635.

Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 27496–27520. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/pei23a.html.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. Technical report, OpenAI, 2018. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf. Preprint.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

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 *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=HPuSIXJaa9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Stephen E. Robertson, Susan Walker, S. Jones, Micheline M. Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3: Retrieving via probabilistic retrieval. *NIST Special Publication*, 500-225:109–123, 1995.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020. URL https://arxiv.org/abs/1910.01108.

Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=9Vrb9D0WI4.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Tool-

former: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=Yacmpz84TH.

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

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations, 2018. URL https://arxiv.org/abs/1803.02155.

Ben Shi, Michael Tang, Karthik R Narasimhan, and Shunyu Yao. Can language models solve olympiad programming? In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kGa4fMtP9l.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Alexander G Shypula, Aman Madaan, Yimeng Zeng, Uri Alon, Jacob R. Gardner, Yiming Yang, Milad Hashemi, Graham Neubig, Parthasarathy Ranganathan, Osbert Bastani, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ix7rLVHXyY.

Manav Singhal, Tushar Aggarwal, Abhijeet Awasthi, Nagarajan Natarajan, and Aditya Kanade. Nofuneval: Funny how code LMs falter on requirements

beyond functional correctness. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=h5umhm6mzj.

Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 3008–3021. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 1102–1121, Toronto, Canada, July 2023a. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.71. URL https://aclanthology.org/2023.findings-acl.71/.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023b. URL https://arxiv.org/abs/2104.09864.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. Patient knowledge distillation for BERT model compression. In Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan, editors, *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4323–4332, Hong Kong, China, November 2019. Association for Computational Linguistics. doi: 10.18653/v1/D19-1441. URL https://aclanthology.org/D19-1441/.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025. URL https://arxiv.org/abs/2412.01007.

Ruixiang Tang, Dehan Kong, Longtao Huang, and Hui Xue. Large language models can be lazy learners: Analyze shortcuts in in-context learning. In *Findings of the Association for Computational Linguistics: ACL 2023*. Association for Computational Linguistics, 2023a. doi: 10.18653/v1/2023.findings-acl.284. URL http://dx.doi.org/10.18653/v1/2023.findings-acl.284.

Xiaojuan Tang, Zilong Zheng, Jiaqi Li, Fanxu Meng, Song-Chun Zhu, Yitao Liang, and Muhan Zhang. Large language models are in-context semantic reasoners rather than symbolic reasoners, 2023b. URL https://arxiv.org/abs/2305.14825.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Siddhant Waghjale, Vishruth Veerendranath, Zhiruo Wang, and Daniel Fried. ECCO: Can we improve model-generated code efficiency without sacrificing functional correctness? In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 15362–15376, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.859. URL https://aclanthology.org/2024.emnlp-main.859/.

Boshi Wang, Xiang Deng, and Huan Sun. Iteratively prompt pre-trained language models for chain of thought. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 2714–2730, Abu

Dhabi, United Arab Emirates, December 2022a. Association for Computational Linguistics. URL https://aclanthology.org/2022.emnlp-main.174.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training, 2024. URL https://arxiv.org/abs/2212.03533.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022b.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=1PL1NIMMrw.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL https://aclanthology.org/2023.acl-long.754/.

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?, 2025. URL https://arxiv.org/abs/2406.14497.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yaofeng Sun, Yuan Liu, Thiago S. F. X. Teixeira, Diyi Yang, Ke Wang, and Alex Aiken. Equibench: Benchmarking code reasoning capabilities of large language models via equivalence checking, 2025. URL https://arxiv.org/abs/2502.12466.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022a. URL https://openreview.net/forum?id=gEZrGCozdqR.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022b.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. A survey on knowledge distillation of large language models. URL https://arxiv.org/abs/2402.13116.

Shi-Qi Yan and Zhen-Hua Ling. Rpo: Retrieval preference optimization for robust retrieval-augmented generation, 2025. URL https://arxiv.org/abs/2501.13726.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with

large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Xiang Yue, Xingwei Qu, Ge Zhang, Yao Fu, Wenhao Huang, Huan Sun, Yu Su, and Wenhu Chen. Mammoth: Building math generalist models through hybrid instruction tuning, 2023.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL https://arxiv.org/abs/2506.05176.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WZH7099tgfM.

Xuekai Zhu, Biqing Qi, Kaiyan Zhang, Xingwei Long, and Bowen Zhou. Pad: Program-aided distillation specializes large models in reasoning, 2023.