Algorithmic Inversion: A Learnable Algorithm Representation for Code Generation

Zhongyi Shi
Chinese Academy of Sciences
Institute of Software
Beijing, China
joeyhere729@gmail.com

Fuzhang Wu*
Chinese Academy of Sciences
Institute of Software
Beijing, China
fuzhang@iscas.ac.cn

Weibin Zeng
Chinese Academy of Sciences
Institute of Software
Beijing, China
zengweibin@iscas.ac.cn

Yan Kong
Chinese Academy of Sciences
Institute of Software
Beijing, China
kongyan@iscas.ac.cn

Sicheng Shen
Chinese Academy of Sciences
Institute of Software
Beijing, China
sicheng@iscas.ac.cn

Yanjun Wu
Chinese Academy of Sciences
Institute of Software
Beijing, China
yanjun@iscas.ac.cn

***Abstract*—The prevalent fine-tuning paradigm for large language models (LLMs) has demonstrated strong performance on various code generation tasks. However, these models still fall short when confronted with algorithmic programming problems, where precise algorithmic reasoning is required. Humans typically adopt diverse algorithmic techniques to tackle complex programming problems, enabling general analysis and accurate implementation. Building on this observation, we propose a method that learns compact, LLM-friendly representations of algorithmic knowledge, termed Algorithmic Inversion (AI), which aims to aid LLMs in understanding programming problems. Specifically, we apply a lightweight fine-tuning process on code-oriented models to automatically learn algorithm embeddings. When concatenated with the inputs, the algorithm embeddings act as instructive signals, guiding LLMs in generating correct code solutions by providing contextual algorithmic hints. We apply our approach to models of three different parameter sizes and evaluate them on three algorithmic programming benchmarks. Our extensive experiments show that applying AI to small (1.5B parameters) models results in absolute improvements of up to 1.8 on Pass@1, while large models (15B parameters) achieve improvements of up to 1.4, compared to Prompt-Tuning. Additionally, our method outperforms traditional full fine-tuning approaches by a significant margin across all tested benchmarks. Furthermore, our analysis of the generated code reveals that AI effectively enhances the model's problem-solving process by providing clear algorithmic guidance. Codes and datasets are available[1].**

***Index Terms*—Code Generation, Algorithmic Knowledge Representation, Large Language Models**

I. Introduction

Code generation refers to the automatic synthesis of program code from natural language descriptions, a technique that has seen significant advances with the advent of large language models (LLMs). While these models have achieved impressive results on many code generation tasks, complex problems like algorithmic programming remain a significant challenge [1]–[3]. Algorithmic programming problems typically include a natural language description, test cases, and code snippets. For straightforward tasks like implementing a bubble sort, current state-of-the-art models [4]–[7] can effortlessly map them to the corresponding code. However, as problem complexity escalates, the direct generation of accurate codes becomes increasingly difficult for both humans and models.

To improve the performance of LLMs on tackling algorithmic programming problems, one common approach is to fine-tune it on copious amounts of natural language-code pairs ($< Problem\ Description,\ Code >$). While straightforward, it overlooks the process of algorithmic analysis, leading the model to rely more on "memorization" rather than "understanding" when generating code. To bridge the semantic gap between problem descriptions and codes, a series of studies [3], [8]–[10] introduce intermediate steps and attempt to decompose complex problems into easy-to-solve sub-problems and then generate a step-by-step directive plan for code generation. While plan-based prompting provides strong interpretability, accurately decomposing a problem into a concrete plan is as complicated as solving the problem itself. Some researchers find that integrating high-level algorithmic knowledge into prompts tends to improve the model's capabilities in problem solving and planning [11]. However, the effectiveness of prompting techniques is highly dependent on the distribution of the model's training data, which is usually unknown. One line of work called P*-tuning [12], [13] involves prepending a sequence of continuous task-specific *prefix* vectors into the inputs and intermediate layers, which are then trained on problem-code pairs to learn an effective prompt. Nevertheless, the direct application of P*-tuning results in a one-for-all *prefix* (effective learned prompt), which is insufficient for algorithmic programming challenges that require flexible application of various programming techniques. These observations motivate us to explore a method

*Corresponding Author
[1]https://github.com/joeysbase/Algorithmic-Inversion

TABLE I
THE TOPIC TAGS CURATED FOR THE ALGORITHMIC PROGRAMMING PROBLEMS. COMMON DATA STRUCTURES LIKE ARRAY ARE GROUPED INTO THE "DATA STRUCTURE" TAG

Algorithms	brute force	dynamic programming
	math	greedy
	prefix sum	randomized
	divide and conquer	constructive algorithm
Techniques	search	sorting
	sliding window	backtracking
	recursion	counting
	memoization	two pointers
	enumeration	
Others	interactive	2-sat
	implementation	simulation
	shortest path	fft
	data stream	design
	special	
Operation	bit manipulation	string suffix structure
	bitmask	data structure

that flexibly exploits programming techniques and introduces effective and easy-to-implement intermediate steps.

In practice, professional programmers typically begin by analyzing complex problems and determining essential programming techniques (e.g., algorithms and data structures). With these techniques as guidance, they can write codes efficiently and accurately. There are two critical steps in this process: **(1) identifying the core programming techniques involved in the input problems**, and **(2) implementing codes guided by these techniques**. Inspired by this fact, this paper proposes a novel code generation method, named **A**lgorithm **I**nversion and **AI** for short (as shown in Fig. 1d). Our approach involves analyzing the problem description to identify key programming techniques and constructing learnable algorithm embeddings as instructive signals for the LLMs. For each type of programming technique, we construct a series of learnable algorithm embeddings, which serve as instructive signals to LLMs by concatenating them with the problem description. Specifically, we develop an independent transformer-based tag generator to produce programming topic tags for each problem. The algorithmic embeddings are treated as learnable parameters in the LLM's input layer, optimized through standard language modeling objective on a frozen LLM. Similar to Huang et al. [11], we categorize algorithmic programming problems into 30 *topic tags* covering various algorithmic techniques based on the Competitive Programmer's Handbook [14] (see Table I). To train our model, we collect data triplets of the form ($< Problem,\ Tags,\ Code >$) from the internet. In summary, our contributions are as follows:

- We propose a novel code generation algorithm capable of analyzing algorithmic programming problems and identifying the essential programming techniques, while explicitly encoding these as learnable embeddings to guide the model in generating accurate code.
- We design a model-friendly representation learning method for algorithmic knowledge, which can be optimized through standard gradient back-propagation. During the training process, we dynamically activate the corresponding algorithmic embeddings using the topic tags of programming problems to compute their gradients and facilitate updates.
- We conducted comprehensive validation as to the efficacy of **AI** based on various benchmarks. Our **AI**-based coder significantly surpasses other competitors on APPS [2] and CodeContest [4], demonstrating the effectiveness of the proposed algorithm embeddings.

II. RELATED WORK

A. Code Fine-tuned LLMs

In recent years, the emergence of large language models (LLMs), particularly those based on Transformer architecture, have made significant progress in the field of code generation. These models have demonstrated remarkable capabilities for understanding and generating natural language. The introduction of models such as CodeX [4], CodeGen [15], Incoder [16], Code LLama [17], CodeGeeX [18], StarCoder [5], and DeepSeek-Coder [6] has propelled LLMs to unprecedented levels in the domain of code generation. Furthermore, Luo et al. [19] introduced WizardCoder, which utilizes the Evol-Instruct method to fine-tune pre-trained models through detailed instructions.

B. Complex Problems Decomposition

Aiming to facilitate interpretability and generalizable problem solving process, researchers take inspiration from human complex problems solving strategies. Self-Planning [9] divides the process of code generation into two phases, planning and implementation; the former step utilizes LLMs' comprehension ability and few-shot prompting to decompose problems into a directive plan, then the plan is appended to the input, allowing models to follow during the implementation in the latter step. KareCoder [11] optimizes code generation by creating prompts that integrates algorithmic knowledge with newly retrieved information. MapCoder [3] utilizes a Planning Agent that formulates step-by-step coding plans to facilitate the code generation process. To address increasingly complex code generation tasks and improve model performance, Li et al. [20] proposed the MoT framework, which aimed to break complex tasks into simple sub-tasks. According to Jain et al. [21], enhancing code readability by inserting detailed plans can improve the accuracy of code generation. Self-Edit [22] proposes a method for producing higher-quality code by leveraging supplementary comments as guidance. The study by [10] demonstrates that fine-tuning models using <problem, reasoning processes> pairs can assist models in addressing competitive-level programming challenges. These studies illustrate that problem decomposition strategies can significantly enhance LLMs' ability to solve complex problems.

C. P*-tuning

Although originally proposed as a lightweight alternative to full fine-tuning, P*-tuning has succeed beyond this intention. P*-tuning generally involves learning several continuous embeddings inserted into the inputs and intermediate layers

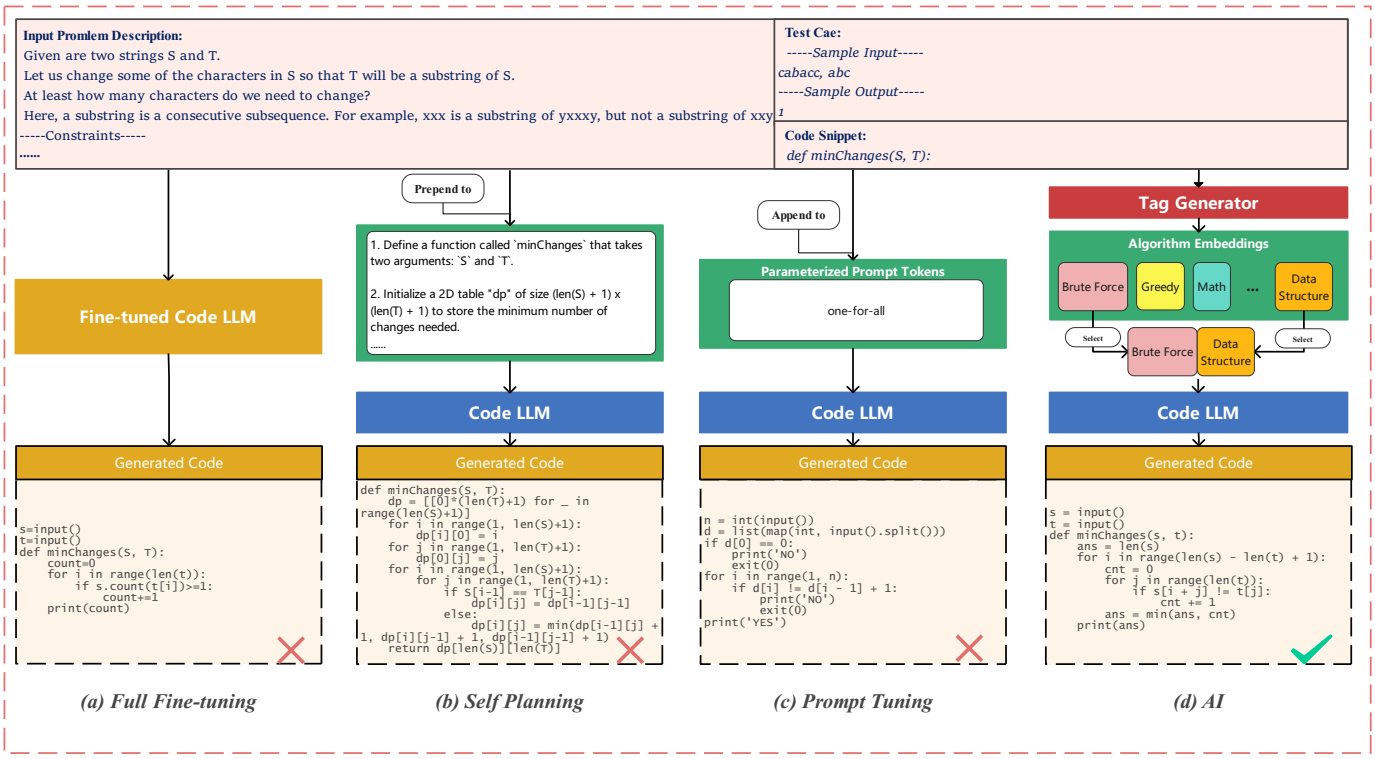

Fig. 1. Existing LLM-based code generation methods face challenges with algorithmic programming problems. Fine-tuning approaches often rely on "memorization" rather than "understanding", plan-based methods depend heavily on the model's problem decomposition ability, and prompt tuning lacks algorithm knowledge. To enhance LLMs' performance in solving these problems, we introduce a novel approach called **A**lgorithm **I**nversion. AI utilizes a tag generator to analyze problems and follow the instructions of algorithm embeddings that store algorithm knowledge.

while freezing pre-trained weights. Prefix Tuning [13] and Prompt Tuning [12] are two similar lightweight fine-tuning methods; Prefix Tuning adds a trainable prefix to each layer of the language model, while Prompt Tuning freezes the pretrained model and adds virtual prompt tokens solely at the input layer. P-Tuning [23] converts prompts into a learnable embeddings and employs a combination of MLP & LSTM to process this embedding layer. Building on the idea of learnable embeddings, Textual-Inversions, introduced by Gal et al. [24], represents image concepts by means of 'pseudo-words (embeddings within the text encoder)' learned from few concept examples, which are capable to guide the model in generating new images that reflect the same concepts. CodeClassPrompt [25] aims to learn distinct features from different layers of a pretrained model and combines them to enhance classification accuracy.

III. Method

Inspired by the "analyze first, then solve" strategy that professional programmers commonly use when solving algorithmic problems, our method includes two main steps when solving algorithmic programming problems:

1) Analyzing the problem to determine relevant programming techniques needed to solve the problem.
2) Implementing the code solutions according to the standard procedure of using the techniques along with details related to the problem.

As shown in Fig. 2, our method first examines whether the problem to be solved has available topic tags to determine the programming techniques involved. If there are no available topic tags, a tag generator is called to analyze the problem and generate the corresponding topic tags. Then, the obtained tags are used to dynamically select and enable the parameters relevant to specific algorithms within the algorithm embedding layer. Specifically, the irrelevant algorithm embeddings (AE) are dismissed, and only those corresponding to the topic tags are selected and concatenated, maintaining attention on the relevant AEs during gradient computation and code generation. Finally, the selected AEs are prepended to the problem embeddings, aiding the generation process by providing algorithmic guidance.

In the following subsections, we detail the core steps in our proposed AI method.

A. Problem Analysis

Accurate analysis of problems and identification of suitable algorithmic techniques are crucial for solving algorithmic programming tasks. Buggy codes are usually repercussions of misunderstandings and a lack of proper guidance. To address

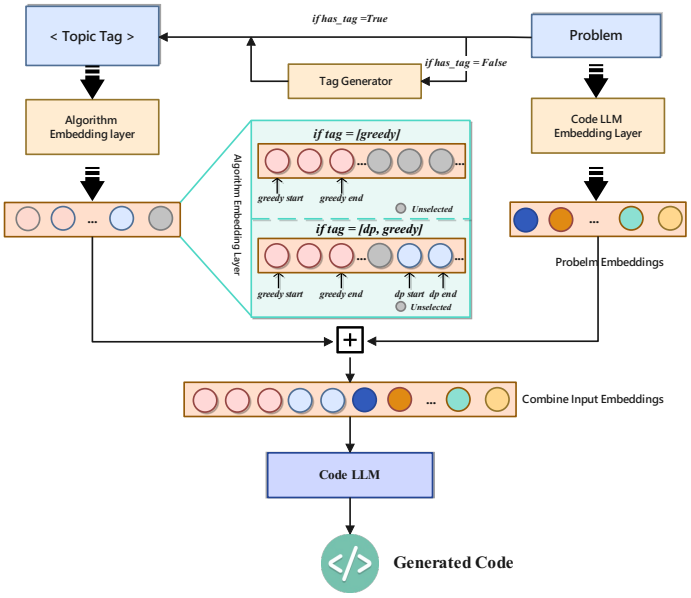

Fig. 2. The procedure of code generation using Algorithm Inversion (AI) method. It first calls a tag generator to analyze the problem, producing relevant topic tags, then the algorithm embeddings corresponding to these topic tags are selected and prepended to the input, steering the generation process.

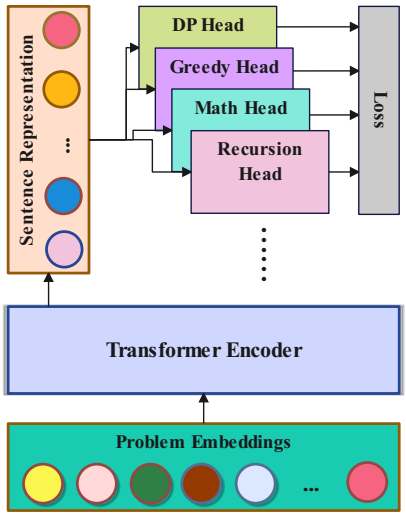

Fig. 3. The outline of the tag generator. It adopts a transformer encoder structure and utilizes multiple specialized classification heads to analyze problems and generate tags.

these issues, a robust tag system is needed to categorize problems, along with an algorithm capable of effective text analysis and classification. Drawing inspirations from the Competitive Programmer's Handbook [14] and based on the available topic tags on many coding platforms like Leetcode[2], we designed a tag system that classifies algorithmic programming problems into 30 topic tags (see Table I). These topic tags can be further grouped into four categories, namely, algorithm, techniques, operations on data structures, and other programming topics. Encoder models are preferred over decoder models as the *tag generator* to fulfill the task of analyzing and classifying texts. The encoder models excel in capturing text semantics and provide a sentence representation vector appropriate to classification [26], [27], while the decoder models specialize in natural language generation (NLG) tasks, which do not align with our goal. Specifically, the language modeling objective leads to a discrepancy between our goal and that of NLG, where multiple token prediction steps are needed to generate a sentence, resulting in cumulative errors. Moreover, the limitations of auto-regressive models lead to disadvantages in computation efficiency and resources. Therefore, we opt for the encoder models, training a tag generator dedicated to analyzing the question and generating tags on its basis. The resulting tag generator is more efficient in terms of computing resources, contributing to the success of our proposed AI method, as shown in Sec. V-A. Additional evaluation of the tag generator can be found in Appendix A.

Our implementation of the *Tag Generator* is based on the Transformer encoder architecture [28]. The topic tags are chosen from $\mathbb{T} = \{t_1, t_2, \ldots, t_K\}$. Here, $K = 30$, representing the total number of topic tag in our tag system. Each tag is internally represented by an integer ranging from 0 to 29 during computation. The N problems in the training data are denoted as $\mathbb{P} = \{p_1, p_2, \ldots, p_N\}$, with their corresponding embeddings represented as $\mathbf{P}^e = Embed^{enc}(\mathbb{P}) = \{\boldsymbol{P}_1^e, \boldsymbol{P}_2^e, \ldots, \boldsymbol{P}_N^e\}$, where $Embed^{enc}(.)$ is the embedding layer of the encoder. The sentence representation vectors $\boldsymbol{P}^{sr}$ are obtained by passing the problem embeddings $\boldsymbol{P}^e$ through the encoder model, expressed as $\boldsymbol{P}^{sr} = Encoder(\mathbf{P}^e) = \{\boldsymbol{p}_1^{sr}, \boldsymbol{p}_2^{sr}, \ldots, \boldsymbol{p}_N^{sr}\}$. The labels are stored in $\boldsymbol{L} = \{\boldsymbol{l}_1, \boldsymbol{l}_2, \ldots, \boldsymbol{l}_n\}^T$, in which $\boldsymbol{l} \in \mathbb{R}^{k \times 1}$ and each element $L_{n,k} \in \{0, 1\}$, with k representing the number of topic tags. A value of 1 indicates the n^{th} problem belongs to the k^{th} topic tag. As a problem may be relevant to several topics, we use the one-vs-all binary classification approach to train multiple specialized classification heads, each dedicated to a specific topic (see Fig. 3). The loss function for training the tag generator is formulated as follows:

$$\begin{aligned} \text{Loss} = -\frac{1}{NK} \sum_{n=1}^{N} \sum_{k=1}^{K} [L_{n,k} \log \delta(MLP_k(\boldsymbol{P}_{:,n}^{sr})) \\ + (1 - L_{n,k}) \log(\delta(1 - MLP_k(\boldsymbol{P}_{:,n}^{sr})))] \end{aligned} \tag{1}$$

where $\delta(.)$ is the sigmoid function, and $MLP_k(.)$ represents the k^{th} classification head.

B. Algorithm Embeddings Learning

Our goal in this section is to find an LLM-friendly representation of algorithmic knowledge (or techniques), i.e., LLMs can fully understand and exploit, resembling human-friendly natural language descriptions. Representing and incorporating this knowledge in terms of natural language as prompt is an intuitive approach [11]. However, the effectiveness of this method heavily depends on the distribution of pre-training

[2]https://leetcode.com/

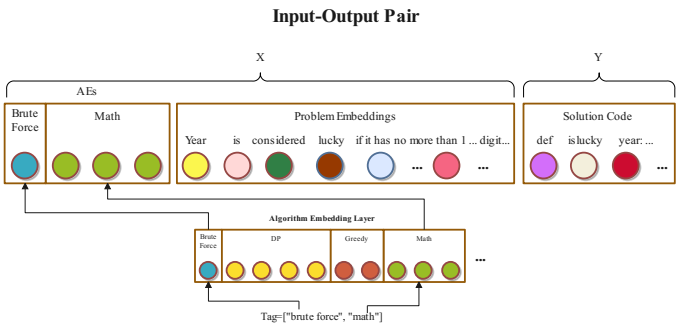

Fig. 4. An example of the input-output pair at training time. The ground-truth tags are used to retrieve relevant AEs and concatenate them. In this way, the relative position of each AE is maintained.

data, which is often unknown, making prompt crafting challenging. Additionally, some models may not be trained to utilize this type of knowledge, namely, they are trained directly on problem-to-code generation data. Thus, appropriate prompts may be difficult for humans to interpret.

To address the challenge and achieve our goal, we propose using learnable algorithm embeddings to represent each algorithmic technique, which is LLM-friendly and obviates the need for manual prompting. Concretely, each algorithmic technique k in our tag system is represented as a matrix (i.e., an algorithm embedding, or AE) $A_k \in \mathbb{R}^{C_k \times D}$, where D is the embedding dimension and C_k indicates the length, which reflects the complexity of the k^{th} topic. C_k is set according to the proportion of average code length of the k^{th} topic to the sum of that of all topics, denoted by $C_k = round(base \frac{AvgCodeLen(t_k)}{\sum_{i=1}^{k} AvgCodeLen(t_i)})$, where $base$ is a hyperparameter of AI, representing the total length of each AE. The length C_k of each AE is computed from the training data (Sec. IV-A) before initialization and remains constant during both training and inference stages. All AEs together comprise the algorithm embedding layer (AEL, Fig. 2). The ground-truth topic tags $\mathbb{P}_n^t \in \mathbb{T}$ of each problem are utilized to select and activate relevant AEs from the AEL at training time. If a problem has multiple tags, the corresponding AEs are concatenated following a predefined order, maintaining the relative order of each AE. Now that the algorithmic knowledge is represented, we only need to incorporate the embeddings into the input. The relevant AEs are prepended to the problem embeddings, forming the combined input $[AEs; Problem\ Embeddings]$ denoted as "X" in Fig. 4. In this way, AEs can be readily learned by maximizing the probability of generating ground-truth solutions ("Y" in Fig. 4), capturing features specific to each individual topic, and creating tailored instructive signals.

C. Code Generation

With the tag generator and algorithmic knowledge representation in place, the next step is to integrate them for implementing our proposed AI method. At inference time, each problem is analyzed by the tag generator, which produces a set of topic tags, denoted as $\mathbb{P}_n^{\bar{t}} \in \mathbb{T}$. The threshold of the tag generator is set high to prevent misleading guidance. As a result, some problems may result in an empty set of topic tags. In such cases, no AEs are selected or prepended to the input, and only the problem is passed to the code LLM which is equivalent to using the pre-trained model alone. We denote the token at the i^{th} decoding step of the n^{th} problem as T_i^n and those before the i^{th} step as $T_{<i}^n$. The generation process can be formulated as follows:

$$\mathrm{T}_i^n = \begin{cases} LM([AEL(\mathbb{P}_n^{\bar{t}}); \mathbf{P}^e_{:,:,n}; Embed^{dec}(T_{<i}^n)]), & \text{if } \mathbb{P}_n^t \neq \emptyset, \\ LM([\mathbf{P}^e_{:,:,n}; Embed^{dec}(T_{<i}^n)]), & \text{otherwise.} \end{cases} \tag{2}$$

where $LM(.)$ represents code LLMs, $AEL(.)$ is the algorithm embedding layer that selects and concatenates the relevant AEs based on the generated tag set $\mathbb{P}_n^{\bar{t}}$, and $Embed^{dec}$ is the embedding layer of code LLMs.

IV. Experiment Setup

We evaluate AI's efficacy and analyze various factors that affect its performance by answering the following research questions (RQs):

RQ1: How does AI perform in the algorithmic programming settings compared to baseline methods? This RQ aims to pinpoint if the additional instructive signals of AE truly improve code accuracy. We conduct comparisons on two well-known datasets, `APPS` and `CodeContest`, as well as on `CodeF`, where AI is contrasted with the KareCoder[3] method using StarCoder2-15B. This helps reveal AI's role in bridging the gap between large and small models.

RQ2: Do AEs converge as the amount of training data increases? Since AEs are learned from specific problem descriptions and code implementations (i.e., co-occurence patterns of tokens), which differ from human perspectives, we want to investigate if AE parameters converge as the amount of training data increases, similar to how humans summarize a general procedure for each algorithmic technique.

RQ3: How does varying the length of AE affect code generation performance? A greater length allows more capacity to capture features and handle complexity within specific topics. This RQ explores whether increased length corresponds to improved performance and if the gains align proportionally.

RQ4: How does the position of AE affect code generation accuracy? Natural language plans typically follow problem descriptions, while general instructions such as task specifications are placed before them. The latter should be broadly effective, as many models have undergone instruction fine-tuning [29], whereas the effectiveness of the former depends on the distribution. This RQ investigates whether placing the AE after problem descriptions yields better performance.

[3]Since KareCoder is not fully open-source, we could only compare it with AI on the CodeF dataset.

RQ5: Do LLMs follow the guidance of AE during the generation process? For this RQ, we analyze codes generated by Qwen2.5-Coder 1.5B with AE integration and using other methods. Both correct and erroneous codes are examined to determine if LLMs adhere to the guidance.

A. datasets and metrics

CodeContest [2]. The CodeContest dataset contains 13,610 problems sourced from the Codeforces platform. It is divided into training, validation, and test sets with a strict temporal split to ensure no data overlap between the sets. Each problem includes metadata such as topic tags and difficulty ratings. In our study, we select the test set and a subset of the training set for evaluation and training, respectively.

APPS [1]. The APPS dataset is sourced from various open-access coding platforms, such as CodeForces and Kattis, containing 10,000 problems halved into training and test sets. In both sets, problems are further split into three subsets based on difficulty levels, namely, Introductory, Interview, and Competition, for finer evaluations. In our experiments, we discovered data overlap between the CodeContest training set and the APPS test set. Therefore, we performed de-duplication based on the problem descriptions by means of MinHash [30] and string matching, resulting in 2,137 non-overlapping problems in the final test set. The de-duplication process created imbalance across difficulty levels. Thus, we sampled 500 problems using stratified sampling to restore balance and ensure effective evaluation.

CodeF [11]. The CodeF dataset, sourced from the CodeForces platform, comprises 1,523 problems split into two sets: pre-September 2021 and post-September 2021. These problems are further categorized into three difficulty levels: simple, medium, and hard.

Training Data. The training data for our experiments includes a subset of the CodeContest training set and a supplementary dataset collected from the LeetCode website. Following Coignion et al. [31], 3,283 problems, along with metadata such as topic tags and similar questions, were scraped from LeetCode's publicly available problem set[4]. We filtered out problems that require Premium membership and fetched solutions for each problem from WalkCC[5], a repository that collects human-written multilingual solutions from various sources. Problems without Python solutions were excluded. Ultimately, 2,302 valid problems with ground-truth tags and solutions were gathered as part of the training set. For CodeContest, we filtered out problems without tags and Python solutions from its training set, resulting in a subset of 4,703 problems. This subset was combined with the data from LeetCode, resulting in a final training dataset of 7,005 problems. The statistics of the training data are shown in Fig. 5.

Metrics. We use Pass@k [4] to assess the accuracy of the generated code. This metric measures functional correctness by evaluating against hidden test cases. For each problem, k code solutions are submitted. If any of the k solutions pass all hidden test cases, the problem is considered solved. Pass@k is then computed as the percentage of problems solved. We set $k = \{1,\ 5,\ 10\}$ for all datasets and sample 10 solutions using a temperature of 0.3 and a top_k of 35 to ensure stable results [31].

[4]https://leetcode.com/problemset/

[5]https://github.com/walkccc/LeetCode

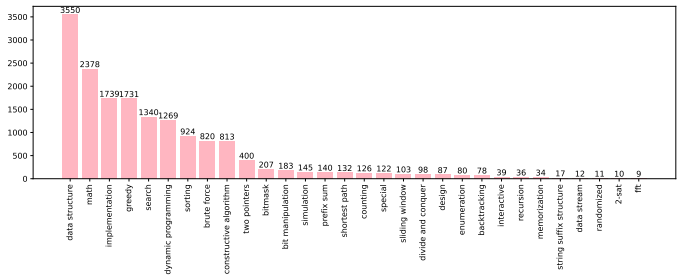

Fig. 5. The statistics of each topic tag in our training data. Some problems have more than one topic tag.

B. Baseline Methods and Models

To demonstrate the efficacy and performance of our approach, we compare it with four effective methods using models of varying parameter sizes.

Full Fine-Tuning (FFT). Full fine-tuning is a widely used method for adapting pre-trained models to downstream tasks. It activates all parameters to compute gradients and update the model.

Self-planning (SP) [9]. A prompting method that leverages LLMs' planning capabilities to decompose intents (problem descriptions in algorithmic programming tasks) into several easy-to-solve sub-problems, which takes the form of step-by-step plans, and solve them accordingly. Empirical studies show that providing ground-truth plans significantly improves performance on several benchmarks.

Prompt-tuning (PT) [12]. A lightweight alternative for fine-tune pre-trained models for downstream tasks. It replaces the discrete prompt tokens in the model's embedding table with prompts parameterized by new dedicated parameters, which can be updated by optimizing the language modeling objective.

KareCoder (KC) [11]. Aiming to improve the inferential capabilities of LLMs by enhancing input information, KareCoder utilizes a manually crafted knowledge library that stores data on algorithms and data structures. At inference time, KareCoder exploits LLM's understanding to identify suitable knowledge and retrieve relevant information from the knowledge library, which is then added to the model's input as part of the prompt. The retrieved information prompts the model to generate a step-by-step plan, which is then followed to create the solution code.

Models. We select four open-source models that are widely used in the research community and have competitive performances on common coding benchmarks, such as HumanEval [4] and MBPP [32], to serve as foundation for our experiments.

Specifically, they include StarCoder2-15B [5], DeepSeek-Coder-1.3B, DeepSeek-Coder-7B [6], and Qwen2.5-Coder-1.5B [7]. Using these models, our proposed AI method is shown to be effective across models of varying sizes.

C. post-processing

To ensure the effective evaluation of test results, we use a post-processing algorithm to accurately extract generated codes. Among the LLMs chosen, the generated codes are often surrounded by natural language descriptions and marked by special symbols ' ` ` `python {code} ` ` ` '; therefore, we apply string matching to locate these symbols and subscript out the desired codes. Those that failed to match, either being special or irregular cases, are deemed compilation errors (which is usually true) and fed directly to the evaluator without further processing.

V. Main Results

A. RQ1: Performance Comparison

Table II presents the main results on the APPS and CodeContest datasets. Ground-truth tags are provided for the evaluation on CodeContest and CodeF, whereas the tag generator is used to create tags for the APPS dataset. The default training configuration for fine-tuning is 10 epochs with an effective batch size of 128 and learning rate of 5e-5. For AI and PT, we adjusted the learning rate to 1e-3 and used a Cosine Annealing scheduler. The Pass@1 metric is the primary focus of our experiments, as it closely aligns with practical scenarios. We report the best result found when varying hyperparameters such as the length of AE and the epochs for the FFT setting.

Evaluation results on CodeContest. Our proposed AI method demonstrated significant improvements over the PT baseline, with relative gains of up to 180% on Pass@1, highlighting the instructive power of AI. Enhancement on Pass@5 and Pass@10 metrics is also pronounced, with the highest relative improvement reaching 108%. Moreover, smaller models (1.3B and 1.5B) show greater improvement compared to larger models (7B and 15B) when using our method, which is advantageous in low-resource contexts. Meanwhile, there are some interesting findings when compared to other baselines. Our analyses of these findings are as follows:

- The FFT method applied to DeepSeek-Coder-1.3B and StarCoder2-15B resulted in decreased performance compared to the Vanilla baseline. The degeneration is likely due to the over-modification, which distorts the pre-trained parameters and negatively impacts code generation accuracy.
- While the PT baseline shows improvement over the Vanilla baseline, the AI method achieves significantly greater enhancement. This result demonstrates the effectiveness of incorporating algorithmic knowledge, suggesting that AE parameters inherently capture such knowledge.
- Unexpectedly, the SP approach provides minimal assistance in solving problems in the CodeContest test set and even leads to performance deterioration. A possible explanation is that the models followed buggy plans generated by state-of-the-art LLMs, which misleads the generation process, as analyzed in Sec. V-E.

Evaluation results on APPS. The effectiveness of AI is further demonstrated by the evaluation results on the APPS dataset. The AI method achieves a relative improvement of up to 32.7% on the Pass@1 metric compared to the PT baseline, with even larger absolute gains observed on Pass@5 and Pass@10. Moreover, AI demonstrates superior performance over the FFT baseline, suggesting AI is a powerful alternative to the FFT paradigm. Also, there are some interesting findings, which we analyze as follows:

- On the DeepSeek-Coder-1.3B base model, AI slightly underperforms the PT approach. This discrepancy likely arises because the tag generator threshold is set high to avoid misleading signals, which results in 214 problems without available topic tags during code generation. In such cases, AI defaults to direct code generation (Vanilla), relying primarily on the base model's inherent capabilities.
- The SP and FFT methods perform differently on the StarCoder2-15B base model compared to other base models. As noted by Lozhkov et al. [5], the StarCoder2-15B base model is specifically trained on direct problem-to-code generation data tailored for algorithmic programming tasks. The SP method deviates from the direct problem-to-code generation pattern observed during pre-training, as it introduces a natural language plan between the problem and the code. This divergence leads to a drop in performance. In contrast, the FFT method involves over-modifications to the pre-trained parameters, which results in a more severe decline in performance.

Evaluation results on CodeF. The comparison between our proposed AI method and the KC baseline is conducted on the CodeF post2021-9 split. As shown in Table III, our proposed AI method demonstrates substantial improvement on the StarCoder2-15B base model compared to the PT baseline, and it surpasses direct generation using ChatGPT. These results indicate that AI can effectively reduce performance gaps between large and small models.

B. RQ2: Convergence of AE

Humans are capable of summarizing general procedures and capturing topic-specific features after observing sufficient examples. Therefore, we investigate whether AEs will converge to a specific point, similar to how humans summarize information. To set up the experiment, we split the training data into multiple subsets of size N at intervals of 1000, where $N = \{1000, 2000, ..., 6000, 7005\}$. Random seeds were fixed to ensure consistent initialization of AE parameters, and the length of AE is set to 67 and 716 to reveal different convergence patterns. We then trained models on each subset and AE length mentioned above using the default training configuration for AI. To reveal convergence patterns, we evaluate the models on the APPS dataset using the Pass@1 metric. The

TABLE II

EVALUATION RESULTS OF AI AND OTHER BASELINE METHODS ON THE APPS AND CODECONTEST DATASETS. VANILLA REFERS TO ZERO-SHOT DIRECT GENERATION USING PRE-TRAINED CODE LLMS.

Code LLMs	MethodsImprovement	CodeContest			APPS		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
Deepseek-Coder-1.3B	Vanilla	0.1	0.5	0.6	1	2.3	3.2
	PT	0.4	0.9	1.2	3.8	8.4	11.2
	SP	0	0	0	0.9	4	7
	FFT	0.06	0.3	0.6	2.4	5.9	8.6
	AI	0.7	1.8	2.5	3.8	7.8	9.8
	Relative Improvement	+75%	+100%	+108%	+0%	-12.8%	-12.5%
Qwen2.5-Coder-1.5B	Vanilla	0.2	0.8	1.2	3.7	6.3	7.4
	PT	0.5	1.6	1.8	5	9.2	11.2
	SP	0.3	0.8	1.2	4.2	10.2	12.4
	FFT	0.4	1.8	2.4	5.5	10.2	12.8
	AI	1.4	3.6	4.2	7.3	12.8	16
	Relative Improvement	+180%	+100%	+75%	+32.7%	+25.5%	+25%
Deepseek-Coder-7B-Base	Vanilla	0.3	1.3	1.8	2	4.8	6
	PT	1	2.6	3.7	6.6	12	14.6
	SP	0.1	0.4	0.6	2.5	8.1	11.4
	FFT	0.9	1.5	1.8	4.5	8.9	11.4
	AI	1.4	3.5	4.3	7.3	12.3	14.8
	Relative Improvement	+40%	+34.6%	+16.2%	+10.6%	+2.5%	+1.4%
StarCoder2-15B	Vanilla	0.9	2.9	4.3	10.3	22.1	27.4
	PT	1.8	4.3	5.5	15.2	25	30
	SP	1.1	2.8	3.6	13.3	23.4	26.6
	FFT	0.7	2.4	3.1	9	16.4	20.4
	AI	3.2	6.2	6.7	15.7	25.7	30
	Relative Improvement	+77.8%	+44.2%	22.1%	+3.3%	+2.8%	+0%

TABLE III

EVALUATION RESULTS ON THE CODEF POST2021-9 SPLIT. RESULTS OF ORANGE COLORED CELLS ARE TAKEN FROM [11].

Method	Pass@1
ChatGPT	12.9
ChatGPT+KC	15.9
ChatGPT+SP	14.2
StarCoder2-15B+PT	10.7
StarCoder2-15B+AI	13.2

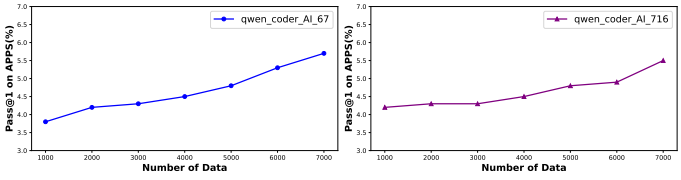

Fig. 6. Variation of the size of training data shows continuing improvement on Pass@1.

rationale is that if AEs are exposed to a sufficient number of examples, the Pass@1 metric should plateau, indicating no further improvement in performance. This plateau would suggest that AEs have captured all topic-relevant features needed to address all problems within a specific topic, much like human-summarized general procedures.

As presented in Fig. 6, performance measured by Pass@1 indicates an upward trend, which suggests that additional training data is required and highlights the presence of under-utilized parameters in AEs.

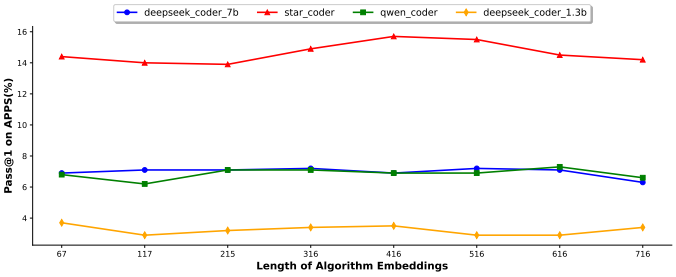

Fig. 7. Variation of AE length does not achieve significant gains on Pass@1.

C. RQ3: Length of AE

Longer AEs mean more parameters for each topic, allowing more features to be captured and stored. We conducted the experiment by varying the length of AEs across $\{67, 117, 215, 316, 416, 516, 616, 716\}$, while keeping all other settings in the default training configuration. Models were trained on each length variation using the AI approach. The performance index, measured by Pass@1 for each length variant, is shown in Fig. 7. We found that the performance of the medium and small-sized models (1.3B, 1.5B, and 7B) does not improve significantly as the length increases; instead, it remains relatively steady. In contrast, the large (15B) model showed a peak performance of 15.6 at the length of 416. This result aligns with the conclusion presented in Sec. V-B, which suggests that parameters may be under-utilized.

TABLE IV
COMPARATIVE EVALUATION OF THE "PREFIX" AND "INFIX" POSITION ON THE APPS DATASET

		APPS		CodeContest	
		Prefix	Infix	Prefix	Infix
Qwen2.5-Coder-1.5B	Pass@1	7.3	2.0	1.4	0.1
StarCoder2-15B	Pass@1	15.6	10.0	3.1	1.4

D. RQ4: Position of AE

In this RQ, we investigate the influence of the position of AE (i.e., prefix and infix) on code accuracy as measured by the Pass@1 metric. The prefix setting ($[AEs; Problem; Code]$) resembles natural language instructions typically placed before a task description, while the infix setting ($[Problem; AEs; Code]$) is similar to self-planning prompting, which is less common and depends on the pre-training distribution. Thus, we hypothesize that the prefix setting should be more effective than the infix setting. The evaluation results for the position variants are shown in Table IV. The infix variant performs significantly worse than the prefix variant, which aligns with our hypothesis.

E. RQ5: Case Analysis

To illustrate the influence of AEs on the generation process, we manually analyzed a problem and its solution codes generated by Qwen2.5-Coder-1.5B using different methods. Three authors conducted the manual analysis, and solution correctness was validated using both hidden test cases and cross-verification by the authors. As shown in Figure 8a, the model successfully generated a solution code that passed all hidden test cases using our proposed AI method. The tag generator produced the topic tags "brute force" and "data structure" for this problem, which was confirmed to be correct through empirical analysis. The model followed the guidance of the "brute force" and "data structure" AEs to iterate through each character in S and T and perform string matching. This approach reflects the "brute force" algorithm and involves operations on the "array" data structure. In contrast, using the self-planning method, the model followed an incorrect plan that misclassified the problem as a dynamic programming setting, resulting in erroneous code. Additionally, both the prompt-tuning and full fine-tuning method produced logically flawed codes that were incomprehensible.

VI. LIMITATIONS

Limited data. As illustrated in Fig. 5, due to objective constraints, the training data we used displays a significant imbalance in tag distribution. Some tags, such as 'randomized' and '2-sat', have only a limited number of instances. This imbalance will negatively impact the accuracy and generalizability of both the tag generator and the algorithm embeddings. Moreover, our experiments in RQ2 (Sec. V-B) suggests insufficient training data, which also limits gains on performance. In future work, we intend to expand both the training data and the tag set. Specifically, we will focus on increasing the sample size for long-tail tags and diversifying the types of programming technique tags to enhance the applicability of the algorithmic embeddings.

Accuracy on tag generator. In our study, we utilize a tag generator to acquire the necessary topic tags at inference time in case there are no available tags in the problems, e.g., APPS. Although the generator we used has achieved an accuracy rate of 85% in our test set, there still exists a discrepancy between the most fitted and the generated tags. Inappropriate tags could mislead models, generating false codes. The issue of generating correct and appropriate tags remains an ongoing research challenge.

VII. CONCLUSION

In this paper, we have explored the LLMs-based methods for solving algorithmic programming problems and proposed a new pipeline for problem analysis and code generation based on algorithmic embeddings. We first summarize common algorithmic techniques in the programming domain and use these programming tags to analysis programming problems. Explicitly encoding these tags as learnable semantic feature vectors significantly enhances the model's ability to find correct code solutions. Additionally, the input programming tag embedding ensures that the generated code exhibits high interpretability and readability. We conduct comprehensive experiments across multiple datasets, and the results convincingly validate the advancement of our proposed Algorithm Inversion method.

VIII. ACKNOWLEDGMENTS

This work is supported by Major Project of ISCAS (ISCAS-ZD-202402).

APPENDIX

A. Tag Generator Evaluation

1) Datasets: We created the test set for the tag generator by combining the CodeF pre-2021-9 split with a test set reserved from the training data. The statistics of the final test set are presented in Figure 9.

2) Metrics: We compute the overall accuracy as the percentage of problems without wrong tags. For instance, if the predicted tags are ["greedy"] and the true tags are ["greedy", "math"], the prediction is considered correct because it does not generate misleading tags. Furthermore, we use two metrics, referred to as Empty-Rate and Error-Rate. Empty-Rate quantifies the proportion of predictions that are empty, while Error-Rate measures the percentage of predictions containing incorrect tags. Additionally, we compute the recall for each individual topic tag to evaluate the model's performance across categories.

3) Performance Evaluation: The evaluation results are summarized in Table V and Figure 10. For this evaluation, the classification head threshold was set to 0.95 and 0.97. Despite variations in recall across topic tags, the tag generator achieved a notable overall accuracy of 87.7%. The Error-Rate could be further reduced by increasing the threshold value, while this could reduce recall and increase Empty-Rate.

<Problem description>
Given are two strings S and T.
Let us change some of the characters in S so that T will be a substring of S.
At least how many characters do we need to change?
Here, a substring is a consecutive subsequence. For example, xxx is a substring of yxxxy, but not a substring of xxyxx.

-----Constraints-----
- The lengths of S and T are each at least 1 and at most 1000.
- The length of T is at most that of S.
- S and T consist of lowercase English letters.

<Test cases>
-----Input-----
Input is given from Standard Input in the following format:
S
T
-----Output-----
Print the minimum number of characters in S that need to be changed.
-----Sample Input-----
cabacc
abc
-----Sample Output-----
1
For example, changing the fourth character a in S to c will match the second through fourth characters in S to T.
Since S itself does not have T as its substring, this number of changes - one - is the minimum needed.

(a) AI

```
s = input()
t = input()
def minChanges(s, t):
    ans = len(s)
    for i in range(len(s) - len(t) + 1):
        cnt = 0
        for j in range(len(t)):
            if s[i + j] != t[j]:
                cnt += 1
        ans = min(ans, cnt)
    print(ans)
```

(b) Self Planning

```
def minChanges(S, T):
    dp = [[0]*(len(T)+1) for _ in range(len(S)+1)]
    for i in range(1, len(S)+1):
        dp[i][0] = i
    for j in range(1, len(T)+1):
        dp[0][j] = j
    for i in range(1, len(S)+1):
        for j in range(1, len(T)+1):
            if S[i-1] == T[j-1]:
                dp[i][j] = dp[i-1][j-1]
            else:
                dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1]
+ 1, dp[i-1][j-1] + 1)
    return dp[len(S)][len(T)]
```

<Plan>
1. Define a function called `minChanges` that takes two arguments: `S` and `T`.

2. Initialize a 2D table "dp" of size (len(S) + 1) x (len(T) + 1) to store the minimum number of changes needed.

3. Initialize the first row and first column of dp with values from 0 to len(S) and 0 to len(T) respectively.

4. Loop over each character in S (indexed by i) and each character in T (indexed by j).

5. In the loop, if S[i-1] == T[j-1], then dp[i][j] = dp[i-1][j-1].

6. In the loop, if S[i-1] != T[j-1], then dp[i][j] = min(dp[i-1][j] + 1, dp[i][j-1] + 1, dp[i-1][j-1] + 1).

7. Return dp[len(S)][len(T)] as the minimum number of changes needed.

(c) Prompt tuning

```
n = int(input())
d = list(map(int, input().split()))
if d[0] == 0:
    print('NO')
    exit(0)
for i in range(1, n):
    if d[i] != d[i - 1] + 1:
        print('NO')
        exit(0)
print('YES')
```

(d) Full Fine-tuning

```
s=input()
t=input()
def minChanges(S, T):
    count=0
    for i in range(len(t)):
        if s.count(t[i])>=1:
            count+=1
    print(count)
```

<Genertaed topic tags>
"brute force" and "data structure"

Fig. 8. Codes generated by Qwen2.5-Coder-1.5B using different methods.

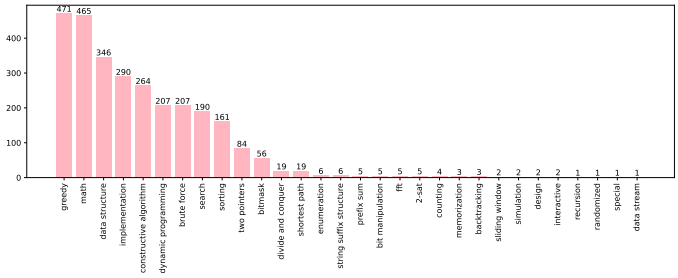

Fig. 9. Statistics of the test set

TABLE V
OVERALL PERFORMANCE INDEX

Threshold	0.95	0.97
Overall Acc	87.7%	90%
Empty-Rate	11.1%	13%
Error-Rate	12.3%	9%

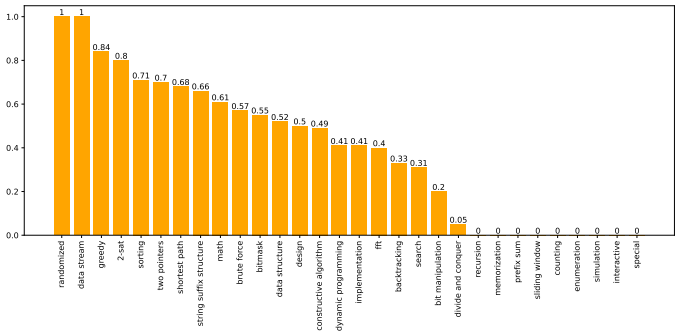

Fig. 10. Recall index of each topic tags

REFERENCES

[1] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt, "Measuring coding challenge competence with APPS," Aug. 2021. [Online]. Available: https://openreview.net/forum?id=sD93GOzH3i5

[2] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. D. Lago, T. Hubert, P. Choy, C. d. M. d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. S. Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022, arXiv:2203.07814 [cs]. [Online]. Available: http://arxiv.org/abs/2203.07814

[3] M. A. Islam, M. E. Ali, and M. R. Parvez, "MapCoder: multi-agent code generation for competitive problem solving," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 4912–4944. [Online]. Available: https://aclanthology.org/2024.acl-long.269

[4] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

[5] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei *et al.*, "Starcoder 2 and the stack v2: The next generation," *arXiv preprint arXiv:2402.19173*, 2024.

[6] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-coder: when the large language model meets programming – the rise of code intelligence," Jan. 2024, arXiv:2401.14196 [cs]. [Online]. Available: http://arxiv.org/abs/2401.14196

[7] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang,

X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," Sep. 2024, arXiv:2409.12186 [cs]. [Online]. Available: http://arxiv.org/abs/2409.12186

[8] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2024, pp. 24 824–24 837.

[9] X. Jiang, Y. Dong, L. Wang, F. Zheng, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Trans. Softw. Eng. Methodol.*, 2024, just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3672456

[10] J. Li and R. Mooney, "Distilling algorithmic reasoning from LLMs via explaining solution programs," Apr. 2024, arXiv:2404.08148 [cs]. [Online]. Available: http://arxiv.org/abs/2404.08148

[11] T. Huang, Z. Sun, Z. Jin, G. Li, and C. Lyu, "Knowledge-aware code generation with large language models," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, ser. ICPC '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 52–63. [Online]. Available: https://dl.acm.org/doi/10.1145/3643916.3644418

[12] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 3045–3059. [Online]. Available: https://aclanthology.org/2021.emnlp-main.243

[13] X. L. Li and P. Liang, "Prefix-tuning: optimizing continuous prompts for generation," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, C. Zong, F. Xia, W. Li, and R. Navigli, Eds. Online: Association for Computational Linguistics, Aug. 2021, pp. 4582–4597. [Online]. Available: https://aclanthology.org/2021.acl-long.353

[14] A. Laaksonen, "Competitive programmer's handbook," *Preprint*, vol. 5, 2017.

[15] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: an open large language model for code with multi-turn program synthesis," Feb. 2023, arXiv:2203.13474. [Online]. Available: http://arxiv.org/abs/2203.13474

[16] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: a generative model for code infilling and synthesis," Apr. 2023, arXiv:2204.05999. [Online]. Available: http://arxiv.org/abs/2204.05999

[17] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: open foundation models for code," Jan. 2024, arXiv:2308.12950. [Online]. Available: http://arxiv.org/abs/2308.12950

[18] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "CodeGeeX: a pre-trained model for code generation with multilingual benchmarking on HumanEval-X," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 5673–5684. [Online]. Available: https://dl.acm.org/doi/10.1145/3580305.3599790

[19] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "WizardCoder: empowering code large language models with evol-instruct," Jun. 2023, arXiv:2306.08568. [Online]. Available: http://arxiv.org/abs/2306.08568

[20] J. Li, P. Chen, B. Xia, H. Xu, and J. Jia, "Motcoder: Elevating large language models with modular of thought for challenging programming tasks," *arXiv preprint arXiv:2312.15960*, 2023.

[21] N. Jain, T. Zhang, W.-L. Chiang, J. E. Gonzalez, K. Sen, and I. Stoica, "LLM-assisted code cleaning for training accurate code generators," Oct. 2023. [Online]. Available: https://openreview.net/forum?id=maRYffiUpI

[22] K. Zhang, Z. Li, J. Li, G. Li, and Z. Jin, "Self-edit: fault-aware code editor for code generation," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 769–787. [Online]. Available: https://aclanthology.org/2023.acl-long.45

[23] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, "GPT understands, too," Oct. 2023, arXiv:2103.10385. [Online]. Available: http://arxiv.org/abs/2103.10385

[24] R. Gal, Y. Alaluf, Y. Atzmon, O. Patashnik, A. H. Bermano, G. Chechik, and D. Cohen-or, "An image is worth one word: personalizing text-to-image generation using textual inversion," Sep. 2022. [Online]. Available: https://openreview.net/forum?id=NAQvF08TcyG

[25] Y. Ma, S. Luo, Y.-M. Shang, Y. Zhang, and Z. Li, "Codeprompt: Improving source code-related classification with knowledge features through prompt learning," *arXiv preprint arXiv:2401.05544*, 2024.

[26] J. Devlin, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[27] Y. Liu, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, vol. 364, 2019.

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 6000–6010.

[29] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[30] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[31] T. Coignion, C. Quinton, and R. Rouvoy, "A performance study of llm-generated code on leetcode," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 79–89. [Online]. Available: https://doi.org/10.1145/3661167.3661221

[32] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.