Knowledge-Aware Code Generation with Large Language Models

Tao Huang
School of Information Science and Engineering, Shandong Normal University
Jinan, China
2022317095@stu.sdnu.edu.cn

Zhihong Sun
School of Information Science and Engineering, Shandong Normal University
Jinan, China
2022021002@stu.sdnu.edu.cn

Zhi Jin*
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
zhijin@pku.edu.cn

Ge Li
Key Lab of HCST (PKU), MOE; SCS, Peking University
Beijing, China
lige@pku.edu.cn

Chen Lyu*†
School of Information Science and Engineering, Shandong Normal University
Jinan, China
lvchen@sdnu.edu.cn

ABSTRACT

Large Language Models (LLMs) perform well on basic programming problems. However, they encounter challenges when dealing with complex tasks involving the use of diverse algorithmic and data structure skills, particularly programming competition-level problems. Notably, ChatGPT exhibits proficient performance on problems it has encountered during its pre-training phase, but this performance deteriorates when faced with novel problems. Consequently, enhancing the ability of LLMs to address unfamiliar problems has emerged as a pivotal research focus. The problem-solving process of LLMs mirrors human programmers' approach to a certain extent. When confronted with new programming tasks, human programmers engage in task planning and code writing with the previously acquired knowledge about algorithms and data structures. Despite having learned such knowledge, LLMs struggle to effectively apply it when faced with specific new problems. To address this issue, we constructed a novel dataset, CodeF, which contains a portion of programming problems that ChatGPT has not previously encountered. Furthermore, we developed a Knowledge Library tailored for Python programming contest problems and introduced the concept of **K**nowledge-Aw**are Code** Gene**r**ation (KareCoder). KareCoder bolsters the models' understanding and problem-solving capabilities by integrating prompt and knowledge from the library into the LLMs' code generation reasoning process, especially on Pass@1 metrics. Upon testing on the CodeF and APPS datasets, KareCoder demonstrated outstanding performance in handling novel problems previously unencountered by LLMs. In contrast with the code directly generated by ChatGPT, KareCoder achieved a relative improvement of 23.3% on the Pass@1 metric on the CodeF post2021-9 dataset. Additionally, it performs well compared to other methods when dealing with problems that LLMs have previously encountered. Our dataset and experiment data are open-sourced and can be accessed at https://github.com/CodeGeneration3/KareCoder.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**.

KEYWORDS

Code Generation, Large Language Models, Knowledge Library

ACM Reference Format:
Tao Huang, Zhihong Sun, Zhi Jin, Ge Li, and Chen Lyu. 2024. Knowledge-Aware Code Generation with Large Language Models. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3643916.3644418

*Zhi Jin and Chen Lyu are the corresponding authors.
†This work was done when Chen Lyu was a visiting scholar at Peking University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '24, April 15–16, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0586-1/24/04…$15.00
https://doi.org/10.1145/3643916.3644418

1 INTRODUCTION

Code generation tasks aim to automatically generate executable programs based on natural language descriptions. In recent years, code generation tasks have garnered substantial attention and undergone extensive development in both the academic and industrial realms. Some prominent applications encompass [1–5]. Specifically, the recent proliferation of Large Language Models (LLMs) such as CodeGen [6], CodeX [7] and ChatGPT [8], has not only induced a profound impact on the domain of code generation but also significantly fostered the progress of associated fields, including Natural Language Processing (NLP) and intelligent software engineering.

Code generation can be conceptualized as a complex reasoning task. The objective of this task is to transfigure Natural Language (NL) descriptions into Programming Language (PL) forms. Typically, this process encompasses two phases: training and inference, requiring a copious number of natural language-code pairs (<Text, Code>) as a foundation. High-quality datasets play a pivotal role in the code generation task. Several related studies [9, 10] have demonstrated that with the enlargement of the model size, augmentation in the quantity of data and escalation in computational prowess, the performance of the model accordingly experiences

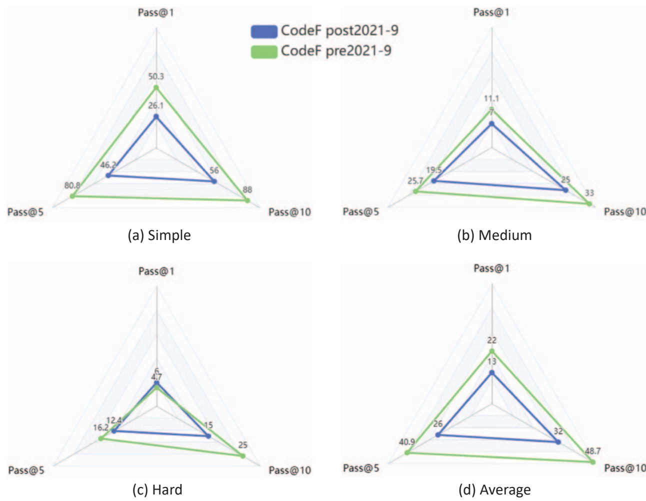


Figure 1: Experimental validation of split-difficulties on CodeF post2021-9 and CodeF pre2021-9 datasets.

enhancement. Nevertheless, empirical evidence suggests that utilizing the model's training set as a test set for inference may lead to over-optimized results. Given that the training data for LLMs are not publicly accessible, datasets frequently employed in code generation tasks, such as MBPP [11], Humaneval [7] and APPS [12], may be encompassed in the training set of LLMs. Consequently, should we use these datasets as test data, whether the genuine test outcomes would be compromised due to this potential data overlap constitutes an issue warranting our thorough exploration.

Based on the above thinking, we construct CodeF to investigate the potential overlap of test data included in the training data of ChatGPT. This research aims to decipher whether this overlap could result in an inflation of actual results. We employ the "gpt-3.5-turbo-0613" model of ChatGPT and use the cutoff time (September 2021) of ChatGPT's training data [13] as disclosed by OpenAI. This date serves as the limit to divide our dataset CodeF[1]. We used a direct generation approach with ChatGPT to produce codes for both parts of the dataset and evaluated the results using Pass@1, Pass@5 and Pass@10. The results are shown in Figure 1. As Figure 1 demonstrates, the experimental results on CodeF pre2021-9 outperform those on CodeF post2021-9, especially regarding problems of easy difficulty, where the relative difference of Pass@1 between the two reaches an astonishing 92.7%. The somewhat lower Pass@1 score for hard difficulty problems in CodeF pre2021-9 might be due to the excessive complexity of these problems, ChatGPT may not have successfully captured the features of these problems.

Drawing on the aforementioned results, LLMs have indeed acquired knowledge of problems they encountered during the training phase. Nonetheless, they appear to lack a strategy for addressing unencountered problems. This results in the disparity in outcomes between the two parts of data depicted in Figure 1. The situation is akin to a student sitting for an examination: they may effortlessly solve problems they have previously reviewed, yet struggle with new and unreviewed problems. With the tutors provide guidance, the student has a significantly enhanced likelihood of successfully completing the problems. We know that algorithms and structures serve as important features of code, so can code generation be enhanced by incorporating algorithms and structures? Hence, we consider infusing some knowledge into the problems in the assistance of in-context learning. This approach can serve to augment ChatGPT's relevant knowledge, thereby ameliorating its proficiency in solving previously unencountered problems.

In previous research, Jiang et al. [14] put forth a code generation method termed "Self-planning". This technique harnesses the innate capability of LLMs to strategize the programming problem, thereby guiding the code generation process. Concurrently, Li et al. [15] proposed a method called "Brainstorming Boost", aimed at stimulating LLMs for deeper introspection. Dong et al. [16] employed a multi-role interaction model to facilitate cooperation and interaction amongst LLMs during code generation tasks. Drawing inspiration from these methods, we opted to enhance the input information fed into LLMs to improve their inferential capabilities when addressing previously unencountered problems. This in turn improves their generalization capacities for handling new programming problems. Consequently, we are introducing a new approach named **K**nowledge-Aw**are** **Code** Gene**r**ation (KareCoder).

Specifically, we first organized and built a Knowledge Library involving algorithms and data structures information based on the tags of algorithms and data structures that may be used in complex programming problems. Subsequently, KareCoder exploits the planning capabilities of LLMs in conjunction with the external Knowledge Library to direct code generation. The operational process of KareCoder unfolds over two distinct stages:

- **Prompt Engineering Stage:** In this stage, the LLMs learn the one-shot example prompt that we furnish and subsequently generate knowledge-aware prompt pertinent to the problem based on the algorithms and data structures information in the Knowledge Library.
- **Coding Stage:** In this stage, we guide the LLMs to incorporate the problem and the prompt generated in the preceding step. Then, under the guidance of the prompt, the model systematically generates code which addresses the corresponding programming problems.

In summary, the main contributions of this paper are as follows:

(1) We constructed a novel code generation dataset, CodeF, consisting of problems manually crawled, cleaned and inspected to mitigate the issue of data overlap with the training data of LLMs. For each programming problem, we extracted pertinent information relating to the algorithms and data structures tags, difficulty and date (time of release).
(2) We assembled a Knowledge Library for Python programming problems, detailing the algorithms and data structures potentially employed in resolving programming problems. We propose a approach grounded in LLMs, named KareCoder, which amalgamates algorithms and data structures knowledge into the code generation process.

[1] CodeF is a novel dataset we have independently designed to meet task requirements, resolving the issue of potential data obsolescence that may exist in extant datasets (specifically, the data may be incorporated in the training data of LLMs like ChatGPT). Detailed information of the dataset can be found in the "Dataset Collection" subsection in Section 3.1.

(3) We conducted a comprehensive validation of the efficacy of KareCoder on both CodeF and APPS benchmarks. KareCoder surpasses other competitors on CodeF (e.g., ChatGPT [8], Self-planning [14], SCOT [17], SCOT&KareCoder), while also demonstrating commendable performance on APPS in comparison to other methods outside of ChatGPT.

2 RELATED WORK

2.1 Code Generation

Model size divides code generation tasks into those using small to medium models and LLMs. Ling et al. [18] first introduced a method to translate natural language into code fragments utilizing a sequence-to-sequence model. Sun et al. [19] adopted the Transformer architecture to resolve the issue of long-term dependencies amongst code elements, proposing TreeGen to enhance the model and incorporate information regarding the code structure. Wang et al. [20] proposed CodeT5, which integrates the code's features during the pre-training phase, emphasizing the model's reasoning about tokens with practical significance during the generation phase. Dong et al. [4] devised a PDA-based approach to guarantee the syntactic correctness of code generation.

As the volume of training data for language models proliferates, more LLMs are being utilized for code generation. The emergence of CodeBERT [21] has marked a new epoch in the use of pre-trained LLMs. Pre-trained LLMs such as CodeGen [6], InCoder [22], CodeX [7], AlphaCode [23], Code Llama [24] and StarCoder [25] have yielded significant enhancements in code generation performance. Beyond these models tailored for code-related tasks, ChatGPT [8], a model designed for problem-and-answer applications, has also demonstrated remarkable code generation capabilities.

2.2 Code Generation Dataset

MBPP [11] and HumanEval [7] are datasets commonly utilized in code generation related research [3, 14, 16]. They comprise handwritten programming problems, which are intrinsically simple. They include only rudimentary task descriptions and provide relatively short solutions, thereby not meeting the complexity required for our daily tasks. Conversely, datasets like APPS [12], derived from several open-source programming competition websites, offer considerably more challenging and lengthier problems, thus serving as a more objective measure. CodeContest [23], designed to fine-tune and evaluate the AlphaCode model, has observed a significant enhancement in the quality of test cases compared to previous datasets. Despite the satisfactory performance of these datasets, there exists a need for a new dataset that will not overlap with the training data of ChatGPT. The cutoff time for the training data of ChatGPT, which OpenAI has made public [13], is September 2021. In order to circumvent a substantial count of false positives in the training data when working with LLMs, there is an immediate requirement for a new dataset encompassing data from September 2021 onwards. Furthermore, the dataset also needs to include the algorithms and data structures tags recommended for the problems to enable more effective integration of algorithms and data structures information.

2.3 Prompt Techniques

As LLMs grow in size and parameters, fine-tuning them needs more resource and time. Numerous recent studies[26–28] have explored enhancing the performance of LLMs by integrating prompts. For instance, Zhang et al. [29] introduced an automated COT prompting method, AutoCoT, which samples diverse problems and generates inference chains to construct examples. Concurrently, Zhou et al. [30] proposed a least-to-most prompting strategy, where complex problems are segmented into a series of sub-problems that are then sequentially addressed. Similarly, Liu et al. [31] delved into the feasibility of guiding ChatGPT in code generation tasks through manually crafted prompts. By incorporating some external information, we could offer prompts to LLMs, thereby enabling them to generate superior prompts for task planning.

3 SUPPORT DATA DESIGN

In this section, we delineate the dataset CodeF and Knowledge Library that we assembled. Specifically, we expound on the reasons behind constructing a new dataset for a programming competition problem, the sources of CodeF and the three distinct types of Knowledge Libraries that we formulated.

3.1 Dataset Collection

Models are conventionally divided into two crucial phases: training and testing. Our research aims to explore the intricate interplay between algorithms, data structures and the overlap between training and testing sets, and their influence on the code generation capability of large models. We undertook the creation of a specialized dataset for an algorithm-centered programming competition, which we named CodeF. This dataset is informed by the methodologies and insights gained from creating the TACO dataset [32]. Taking September 2021 (the cutoff date for ChatGPT's training data [13]) as the node, we divided CodeF into two parts: CodeF pre2021-9 whose problems ChatGPT might have encountered during its training, CodeF post2021-9 which consists of problems that ChatGPT would not have previously seen.

In this study, we have constructed a new dataset of Python programming problems, CodeF, which incorporates 1523 problems posted from January 2020 to April 2023 on the CodeForces[2] programming website. As depicted in Figure 2, the creation of CodeF primarily involves two phases: data acquisition and processing.

During the data acquisition phase, we evaluated several programming contest platforms, including Codeforces, HackerRank and Geeksforgeeks. Factors such as legal restrictions, interface complexities of the platforms and our specific requirement for algorithms and data structures tag for the problems, as well as the release time information, led us to select CodeForces (a programming competition website) as our data source. In order to ensure dataset quality, we designed an HTML parser specifically tailored for the CodeForces site. We crawled the site for problem description, solution, input and output examples and a set of associated labels, which include difficulty, date and tag (indicating algorithms and data structures suitable for solving the problem), etc. We consolidated all the problems into a standardized format.

[2]https://codeforces.com

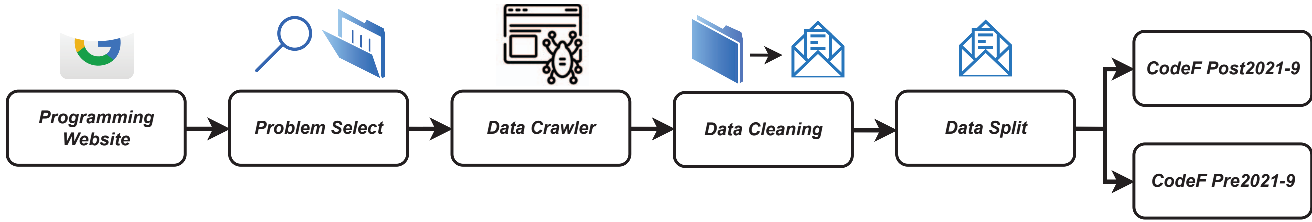


Figure 2: Schematic diagram of CodeF acquisition and processing. The processing part includes data cleaning and splitting.

Table 1: Subsets of Simple, Medium and Hard difficulties in CodeF Post2021-9 and CodeF Pre2021-9 datasets.

Dataset	*Difficulty*	*Difficulty Range*	*Problem Number*
CodeF Post2021-9	*Simple*	*800	000-099
	Medium	*1300-*1600	400-499
	Hard	*1900-*2500	600-699
CodeF Pre2021-9	*Simple*	*800	000-099
	Medium	*1300-*1600	400-499
	Hard	*1900-*2200	640-739

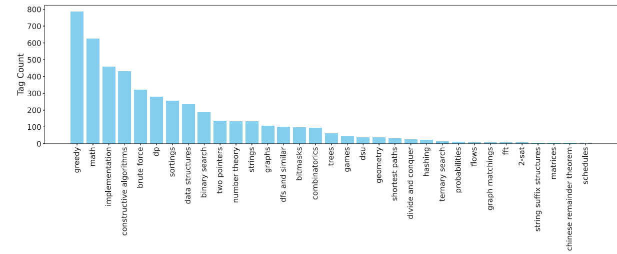
Table 2: Descriptive statistics of CodeF.

Dataset	*Average question token*	*Average algorithm tags number*
CodeF	2047.1	3.1
CodeF pre2021-9	2018.7	3.2
CodeF post2021-9	2079.0	3.0

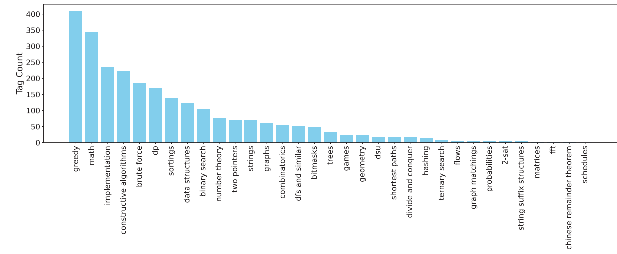
During the data acquisition process, we endeavoured to ensure the consistency of our data with that of the CodeForces website. However, since the code was sourced from user submissions, often embedded with comments and potential error codes, we implemented the following measures during the data processing phase:

- **Code De-commenting:** For a cleaner CodeF and to remove comments that are considered invalid information, we employed Abstract Syntax Tree (AST) parsing to remove comment nodes from code that was heavily annotated.
- **Code De-duplication:** We collected codes from various users, leading to potential duplicates. Using the MinHash algorithm and Jaccard index, with a threshold of 0.85, we minimized these duplicates to unique files.
- **Unit Test Validation:** Although we extracted code that was verified as correct by the website, we conducted unit tests on all codes to prevent errors during parsing. This step assured the accuracy of the test cases and codes.

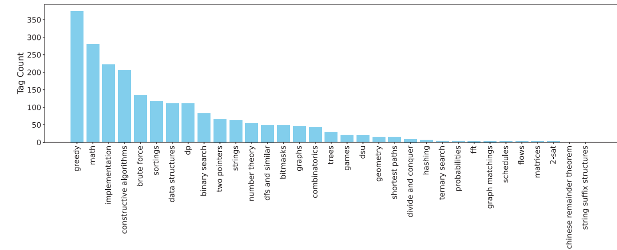
According to the cutoff date of ChatGPT's training data - September 2021, we partitioned the dataset into CodeF pre2021-9 (comprising 805 problems) and CodeF post2021-9 (consisting of 718 problems). We got the raw data on the difficulty of the problems from the CodeForces website. The difficulty level of all problems is



(a) Distribution of CodeF.



(b) Distribution of CodeF pre2021-9.



(c) Distribution of CodeF post2021-9.

Figure 3: Distribution of algorithms and data structure tags.

denoted in the format "*xxx" (ranging from *800-*3100[3]). In order to explore the rules of different difficulty problems, we classified the datasets according to the level of difficulty and extracted one hundred problems each from the Simple, Medium and Hard difficulty categories from both datasets, as depicted in Table 1.

To verify the reasonableness of the two parts of the data after we divided CodeF with the time of September 2021 as the node, we analyzed the average problem length and the average number of algorithm tags of the CodeF and the two parts of CodeF, as shown in Table 2. We invited eight students with proficient knowledge of algorithms and data structures to examine the algorithm and data

[3]The difficulty level of which were sourced from the CodeForces programming website.

```
1  // Knowledge Description:
2  // This is the default Knowledge Library of KareCoder.
3  Greedy:The greedy algorithm is a heuristic approach that selects the best
         available option at each step without considering the long-term impact.
         It is useful for optimization problems where finding an exact
         solution is intractable, but may not always lead to the optimal
         solution. Greedy algorithms are easy to implement and efficient.
4  ---------------------------------------------------------------------------
5  // Knowledge pseudo-code:
6  Greedy:function Greedy(problem):
7             solution = empty_solution()
8             while not problem_solved(problem, solution):
9                 candidate = find_best_candidate(problem, solution)
10             add_candidate_to_solution(candidate, solution)
11        return solution
12 ---------------------------------------------------------------------------
13 // Knowledge Step of pseudo-code:
14 Greedy:1.Initializes an empty solution.
15        2.Enters a loop until the problem is solved, based on the solution.
16        3.Finds the best candidate for the next step based on the problem and
         the current solution.
17        4.Adds the best candidate to the solution.
18        5.Continues to the next iteration of the loop.
19        6.Once the problem is solved, it returns the final solution.
```

Figure 4: Examples of different Knowledge Libraries. The corresponding Greedy algorithm is shown here.

structures tags in our CodeF dataset, and eventually summarized 33 categories of tags. Meanwhile, in order to show the distribution of various types of algorithms and data structures on the problems of our dataset more intuitively, the distribution of tags was also summarized using statistical methods. The classification of all 33 algorithms and data structures tags in different parts of the CodeF dataset is shown in Figure 3.

Although our original intention of creating CodeF was to test the effectiveness of LLMs on programming problems of different vintages. After a series of rigorous processing during the construction of our dataset, in the end, CodeF dataset can be used as training data for code generation models, and the algorithms and data structures information in it is also can be applied to the fields of code understanding and code algorithm recommendation.

3.2 Knowledge Library

To enhance ChatGPT's proficiency in tackling novel programming problems, it is necessary to equip it with knowledge about algorithms and data structures. This knowledge serves as contextual reference information during the generation of prompts to address these problems. Specifically, we have conceptualized three formats of the Knowledge Library and investigated the effectiveness of these varying formats for prompt generation in RQ2 (see Section 6.2).

We used resources such as ChatGPT and Google search engines to gather initial information and refer to the book Competitive Programmer's Handbook [33] for organization. Concurrently, we invited a group of individuals proficient in competitive programming knowledge. We manually refined and summarized the descriptions of each algorithm and data structure tag (these tags include all tags of CodeF problems). Our Knowledge Library of algorithms and data structures covers as much as possible the knowledge used in common programming problems. We invited seven employees and interns from companies in the programming industry to manually evaluate the correctness of the knowledge in our Knowledge Library. They evaluated all 33 pieces of knowledge and revised the final Knowledge Library to correctly explain each algorithm or data structure.

Our Knowledge Library is stored in the form of a dictionary, with each tag and its knowledge forming a one-to-one correspondence, i.e., {"tag": "knowledge"}. Ultimately, we integrated the information from the manually evaluated knowledge to build our Knowledge Library. Figure 4 provides a few examples from the Knowledge Library. The Knowledge Library we devised principally assumes the following three formats:

- **Knowledge Description:** This format provides a definitive overview of algorithms and data structures knowledge, encompassing descriptions of their properties and definitions. With this type of Knowledge Library, ChatGPT can comprehend the basic characteristics and applications of the corresponding algorithms and data structures, thereby generating appropriate problem-solving prompts.
- **Knowledge Pseudo-Code:** This Knowledge Library offers pseudo-code explanations of algorithms and data structures, emulating programming syntax to describe the steps and procedures involved in algorithms and data structures. By exploring this library, ChatGPT can understand the specific implementation of the corresponding programming knowledge, resulting in prompt skewed towards solution steps.
- **Knowledge Step of Pseudo-Code:** This form of Knowledge Library provides a textual step-by-step explanation of the pseudo-code associated with algorithms and data structures knowledge. With its assistance, ChatGPT can understand the steps and flow of programming knowledge from a natural language perspective.

In our Knowledge Library, we offered descriptions, pseudo-code examples and steps of pseudo-code for 33 algorithms and data structure categories, such as greedy and dynamic programming. Essential to the KareCoder method, this content complements CodeF. Additionally, it can be integrated with the CodeF to train models for applications in code generation and code evaluation domains.

4 APPROACH

In this section, we present a new method: KareCoder. We begin with an overview of KareCoder and present it in two stages.

4.1 Overview

Code generation aims to help solve programming problems. We postulate that the thought process of LLMs parallels that of human programmers, which can be bifurcated into two stages when tackling programming problems: initially, comprehending the problem and learning or recalling knowledge that may be applied to its solution, followed by formulating a preliminary plan; secondly, executing code writing based on this plan.

While a majority of problems on programming platforms come with algorithmic or data structure tags, there exists a subset of problems from non-programming platforms or datasets, such as APPS, lacking these algorithmic tags. To enhance the adaptability of KareCoder, we developed a tag generator, drawing from ChatGPT and Prompt technologies and based on our Knowledge Library. This generator can formulate or rectify tags, especially for problems that either lack them or have tags misaligned with our Knowledge

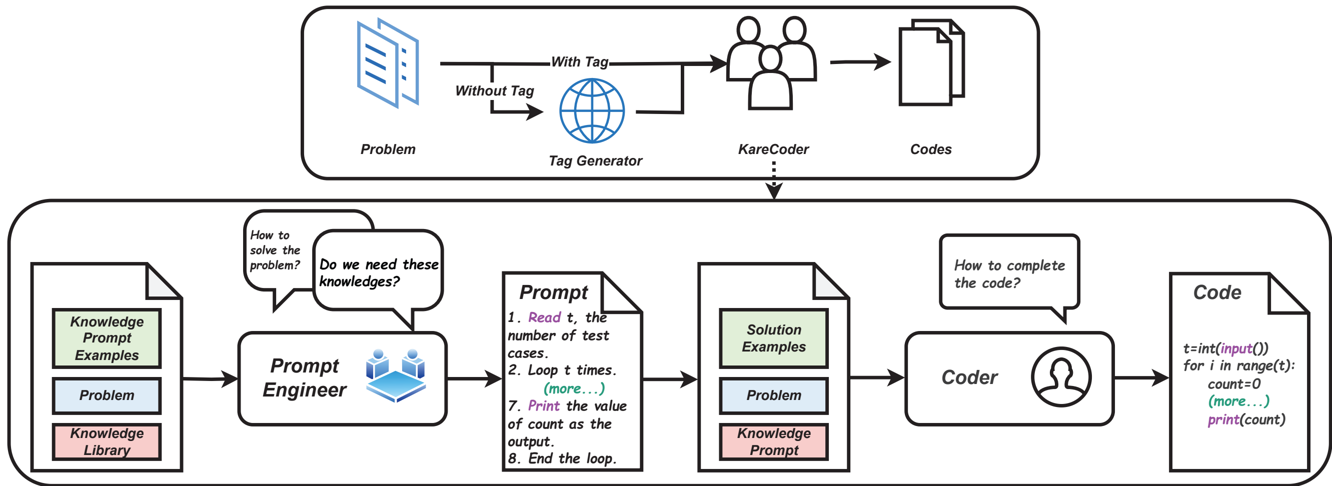


Figure 5: An overview of KareCoder. Given a programming problem, KareCoder matches the Knowledge Library appropriate to the problem and then generates a prompt for solving the problem. Finally, the prompt is used to guide the generation of code.

Library. We evaluated the tag generator's accuracy with Cohen's Kappa and manual evaluation methods. Leveraging this tag generator, we ensure a seamless integration of each programming problem into KareCoder's processing framework. Consequently, as depicted in Figure 5, we connected a tag generator in front of KareCoder and partitioned the workflow of KareCoder into two stages, Prompt Engineering and Coding, adhering to a sequential approach to the task:

- In **Prompt Engineering Stage**, we designate ChatGPT as a prompt engineer, accountable for comprehending the problem and learning as well as reviewing the algorithms and data structures knowledge pertinent to resolving the problem, subsequently generating a problem-solving prompt.
- In **Coding Stage**, we assign ChatGPT the role of a coder, tasked with perusing the problem and implementing this solution in a programming language, guided by the prompt provided by the prompt engineer.

4.2 Prompt Engineering Stage

In Prompt Engineering Stage, we regard ChatGPT as a prompt engineer. We anticipate that ChatGPT will acquire the relevant knowledge needed for resolving programming problems, subsequently, based on its pre-existing knowledge complemented by newly recalled information, it will develop a prompt for addressing the problem. Given ChatGPT's input window constraints, inputting the entire Knowledge Library isn't viable. Therefore, we update and customize the knowledge for each problem by performing absolute value matching via dictionaries to associate the tags of the problem with the tags in the Knowledge Library. The working principle of dictionary matching is as follows: The corresponding relationships between the dataset and the Knowledge Library are <problem, tag> and <tag, knowledge>, respectively. We can achieve matching through the tag in both of them.

As depicted in Figure 6, we manually constructed a one-shot example (X) that generates a knowledge-aware prompt (P) based on the problem (Q) and the knowledge description (K') in our Knowledge Library (K). The one-shot example (X) we developed was

```
1  // Problem (Q):
2  There is a string s of length 3, consisting of uppercase and lowercase
       English letters. Check if it is equal to "YES" (without quotes), where
       each letter can be in any case. For example, "yES", "Yes", "yes" are
       all allowable. (more ...)
3  // Knowledge Description (K'):
4  Brute force: Brute force is a straightforward approach to problem-solving
       that exhaustively searches all possible solutions. (more ...)
5  Strings: Strings are sequences of characters used to represent text or other
       data in a computer program. (more ...)
6  ------------------------------------------------Example (X)----------------
7  //Knowledge-aware Prompt (P):
8  (more...)
9  2. Loop t times to read each test case string s.
10 3. Convert the string s to lowercase using the built-in function lower().
11 4. Check if the string s is equal to "yes".
12 (more...)
```

```
1  // Problem (Q):
2  You are given a grid with n rows and m columns. We denote the square on the i
       -th (1 ≤ i ≤ n) row and j-th (1 ≤ j ≤ m) column by (i, j) and the
       number there by a_ij. All numbers are equal to 1 or to -1. You start
       from the square (1, 1) and can move one square down or one square to
       the right at a time. You want to end up at the square (n, m). Is it
       possible to move in such a way so that the sum of the values written
       in all the visited cells (including a_11 and a_nm) is 0? (more...)
3  // Knowledge Description (K'):
4  DP: Dynamic programming, is a technique used to solve complex problems by
       breaking them down into smaller, simpler sub-problems. It involves
       solving each sub-problem only once and storing the solutions in a
       table for future use. (more...)
5  ------------------------------------------------Prompt Engineer-------------
6  // Knowledge-aware Prompt (P):
7  (more...)
8  6.Initialize an array dp of size m+1 with zeros.     # DP initialization.
9  7.Loop n times, set dp[0] to 1 if i=0, else 0.       # DP boundary conditions.
10 8.Loop m times, compute the value of dp[j + 1] based on dp[j], dp[j + 1] and
       the element at a[i][j]. The bit shift operation is used here to handle
       the state transition.   # DP status transfer.
11 9.Check the (n+m-1)//2 bit of dp[-1]. If it is 1, print "YES"; otherwise,
       print "NO".    # DP result check.
12 (more...)
```

Figure 6: Illustration of the input and output of Prompt Engineering Stage. The input includes a example, a new problem and its corresponding knowledge description, the output is the knowledge-aware prompt for the new problem.

```
// Problem (Q):
There is a string s of length 3, consisting of uppercase and lowercase
      English letters. Check if it is equal to "YES" (without quotes), where
      each letter can be in any case. For example, "yES", "Yes", "yes" are
      all allowable. (more ...)
// Knowledge-aware Prompt (P):
(more...)
2. Loop t times to read each test case string s.
3. Convert the string s to lowercase using the built-in function lower().
4. Check if the string s is equal to "yes".
(more...)
-----------------------------------------------Example (Y)------------------
// Solution Code (C):
(more...)
for i in range(t):
    s = input()
    s = s.lower()
    print('YES' if s == 'yes' else 'NO')
(more...)
```

```
// Problem (Q):
You are given a grid with n rows and m columns. We denote the square on the i
      -th (1 ≤ i ≤ n) row and j-th (1 ≤ j ≤ m) column by (i, j) and the
      number there by a_ij. All numbers are equal to 1 or to -1. You start
      from the square (1, 1) and can move one square down or one square to
      the right at a time. You want to end up at the square (n, m). Is it
      possible to move in such a way so that the sum of the values written
      in all the visited cells (including a_11 and a_nm) is 0? (more...)
// Knowledge-aware Prompt (P):
(more...)
6.Initialize an array dp of size m+1 with zeros.    # DP initialization.
7.Loop n times, set dp[0] to 1 if i=0, else 0.      # DP boundary conditions.
8.Loop m times, compute the value of dp[j + 1] based on dp[j], dp[j + 1] and
      the element at a[i][j]. The bit shift operation is used here to handle
      the state transition.   # DP status transfer.
9.Check the (n+m-1)//2 bit of dp[-1]. If it is 1, print "YES"; otherwise,
      print "NO".   # DP result check.
(more...)
----------------------------------------------Coder---------------------
// Solution Code (C):
(more...)
dp = [0] * (m + 1)          # DP initialization.
for i in range(n):          # DP body.
    dp[0] = int(not i)      # DP boundary conditions.
    for j in range(m):      # DP status transfer.
        dp[j + 1] = (dp[j] | dp[j + 1]) << (a[i][j] == -1)
if dp[-1] & (1 << ((n + m - 1) // 2)):      # DP result check.
    print("YES")
else:
    print("NO")
(more...)
```

Figure 7: Illustration of the input and output of Coding Stage. The input includes a example, a new problem and its corresponding prompt for generating code, the output is the solution code for the new problem.

curated by eight students involved in the review of CodeF's algorithm and data structure tags. Following the review, we identified four representative examples for experimentation, ultimately selecting one as the one-shot example. Subsequently, we concatenated the new problem with a DP (dynamic programming) knowledge description derived from the Knowledge Library. Relying on the one-shot example we provided and the DP algorithms pertinent to the new problem, we crafted a step-by-step problem-solving, knowledge-aware prompt. Full details are available on our GitHub[4].

Let Q represent the problem, P represent the knowledge-aware prompt, K signify the Knowledge Library, K' signify the knowledge description, X denote a set of examples utilized to generate prompts based on the problem and the knowledge description, i.e., $X = \left\{\left\langle Q_x, K'_x, P_x\right\rangle\right\}_{x=1}^{n}$. Consequently, the knowledge-aware prompt generation problem can be modelled as:

$$f\left(P \mid Q, K, X\right) \triangleq f\left(P \mid Q, K', X\right) \cdot m\left(K' \mid Q, K\right) \tag{1}$$

[4]https://github.com/CodeGeneration3/KareCoder

Herein, f represents the generative tasks completed using ChatGPT, m signifies the dictionary-matching tasks of knowledge.

4.3 Coding Stage

During Coding Stage, we conceptualize ChatGPT as a Coder, with the expectation that it can integrate the problem and the knowledge-aware prompt to generate code that resolves the problem. The objective of this stage is to transform a step-by-step prompt, represented in natural language, into an executable program. As illustrated in Figure 7, this diagram schematically portrays the Coding Stage. Full details are available on our GitHub.

Let Q represent the problem, P stand for the knowledge-aware prompt, C symbolize the generated code, and Y correspond to a set of examples pertaining to code generation, which are predicated on the problem and the associated prompt, i.e., $Y = \left\{\left\langle Q_y, P_y, C_y\right\rangle\right\}_{y=1}^{n}$. The one-shot example ($Y$) was summarized using the same approach as one-shot example (X). Therefore, the task of code generation based on the prompt can be mathematically formulated as :

$$f\left(C \mid Q, P, Y\right) \tag{2}$$

Subsequently, the comprehensive KareCoder approach can be mathematically represented as:

$$f(C \mid Q, K) \triangleq \underbrace{f(C \mid Q, P, Y) \cdot}_{\text{Generate-Code}} \underbrace{f(P \mid Q, K', X) \cdot}_{\text{Generate-Prompt}} \underbrace{m(K' \mid Q, K)}_{\text{Match-Knowledge}} \tag{3}$$

5 ENVIRONMENT SETUP

5.1 Research Questions

In this research, we introduce KareCoder that supplements ChatGPT with a Knowledge Library to guide its operation. To evaluate the efficacy of KareCoder and to analyze the associated influencing factors, we explore the following research questions (RQs):

RQ1: How does the KareCoder perform compared to other baselines? The objective of this RQ is to ascertain whether KareCoder can produce programs of greater accuracy than other methods. We conducted comparisons using the CodeF post2021-9 dataset as well as subsets of the dataset delineated by levels of difficulty - Simple, Medium and Hard.

RQ2: What is the better choice for the Knowledge Library? Determining how to enable ChatGPT to better utilize programming knowledge without constraining its innate reasoning abilities is a challenge we need to address. Consequently, we have designed three distinctive forms of the Knowledge Library for evaluation.

RQ3: Will different shot times and ChatGPT Settings affect the KareCoder performance? Within this RQ, we examined the impact of employing k-shot examples on the effectiveness of KareCoder and whether variations in parameter settings affect KareCoder's performance under a 1-shot example condition.

RQ4: Is KareCoder only valid because of the datasets we used? This can be a confusing issue for many people. Hence, we also tested our methods on the first 500 problems from the APPS test set to demonstrate that the efficacy of KareCoder is not contingent upon a specific dataset.

5.2 Benchmarks

CodeF Dataset: CodeF is a Python task dataset that we developed. These problems are sourced from the CodeForces website and range from January 2020 to April 2023, encompassing a total of 1523 problems. Of these, 805 problems are from the period prior to September 2021 and 718 problems are from the subsequent period. Our experiments are primarily focused on the problems posed after September 2021. From these problems, we extracted three subsets, namely Simple, Medium and Hard, based on the level of difficulty.

APPS Dataset: We used the APPS dataset as an auxiliary validation source in our experiments. The APPS dataset is derived from various programming competition websites, including CodeForces, LeetCode, CodeChef, etc. The training set comprises 5,000 problems, while the test set also contains 5,000 problems. In our experiments, we selected the first 500 problems of the test set for our tests.

5.3 Comparison Baselines

For the purposes of comparison, we selected ChatGPT, two recently proposed methods for code generation, specifically, Self-planning [14] and SCOT [17], and SCOT&KareCoder which combines the Structured COT and Knowledge-aware method as baselines. Simultaneously, to further investigate the capabilities of each method, we crafted a system prompt tailored for the four baselines methods with the aim of enhancing the quality of the generation results.

ChatGPT [8] is an LLM proposed by OpenAI, it exhibits characteristics similar to a question-and-answer system, with a powerful code-writing ability. It exhibits exceptional performance in few-shot cases. For our experiments, we utilized OpenAI's API to call ChatGPT with the specific model "gpt-3.5-turbo-0613".

Plan is our implementation inspired by Self-planning [14]. The Self-planning strategy encompasses two phases: planning phase and implementation phase. Drawing from its idea, we fashioned a two-phase process: initially, we allow ChatGPT to generate a step-by-step prompt for problem-solving, following which, the code of the problem is generated under the guidance of the prompt.

SCOT [17] aimed at investigating how to access the coding mindset of LLMs in the context of code generation. SCOT first generates a Structured COT, delineating the solution process in programming logic, then transmutes this Structured COT into a program using a specific programming language. SCOT specifies a temperature setting of 0.8 and a Top_P value of 0.95, we adhered to these parameters in our experiments.

SCOT&KareCoder is a way to combine KareCoder's knowledge-aware prompt with SCOT's Structured COT. This baseline differs from KareCoder in that instead of generating a step-by-step prompt similar to the Plan approach in the Prompt Engineering Stage, we generate a Knowledge-aware Structured COT to guide the code generation.

5.4 Metrics

To evaluate the accuracy of the generated code, we used Pass@k as an evaluation metric, as shown in the formula below. For each problem, we generate n ≥ k copies of code and calculate the number of correct samples that pass the test sample, c (where c ≥ n). In our experiments, we generated 5 copies of code for each problem and evaluated our approach using Pass@1, Pass@3 and Pass@5. Notably, Pass@1 is especially important as it aligns more with practical application needs.

$$Pass@k = \mathop{\mathbb{E}}_{Problems}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \tag{4}$$

6 RESULT AND ANALYSIS

In conducting the experiments for these research questions, we adhere to a default configuration of a one-shot setting, with the temperature parameter set to 1 and the Top_p parameter also set to 1. Should there be alterations to this configuration, we will explicitly specify them. The comparative rule utilized in our study involves contrasting KareCoder (marked in blue in the tables) with the other best methods (marked in yellow in the tables).

6.1 RQ1: Performance Comparison

1) Evaluation on Simple, Medium and Hard difficulties of CodeF Post2021-9. Within RQ1, we examined the effectiveness of the knowledge introduced by KareCoder on three difficulty levels (Simple, Medium and Hard) within the CodeF post2021-9 dataset through a series of comparative experiments, which are depicted in Table 3. We assessed the outcomes of direct code generation by ChatGPT with 1-shot, as well as Plan, SCOT, SCOT&KareCoder and KareCoder under 1-shot and 3-shot conditions, respectively. The analyses of these results are as follows:

- KareCoder approach markedly excelled beyond other approaches in terms of the Pass@1 performance metric. This enhancement is largely ascribed to the creation of a knowledge-aware prompt, individually tailored to each programming problem. Such prompts not only directed the code generation process but also markedly bolstered the code's correctness, aligning with our objective of augmenting the Pass@1 performance index.
- Nevertheless, in Pass@3 and Pass@5 metrics, KareCoder still outperformed other methods, but it didn't achieve the significant gains that direct code generation with ChatGPT did. This results likely arises from the inherent risks associated with the intermediate step of prompt or Structured COT generation. Though designed to facilitate code production, inaccuracies in these generated elements have the potential to compromise the precision of all subsequent code generation pertaining to the specific problem at hand.
- In our study, KareCoder outshone in 1-shot Pass@k metrics, except Hard level, with over 20% advantages in Pass@3 and Pass@5 in simple and medium level. In addition, KareCoder's in the 3-shot experiment did not result in an absolute advantage. However, multiple shots may lead to waste of resources caused by long input length. Thus, we favor fewer-shot techniques, a preference supported by our RQ3 experiment assessing KareCoder's efficiency.

2) Evaluation on the CodeF Post2021-9. Further tests on the CodeF Post2021-9 dataset confirm KareCoder's efficacy. As Table 4 illustrates, on the Pass@1 metric, KareCoder and the Plan method achieve 15.9% and 14.2% respectively, ranking first and second, far

Table 3: Comparative evaluation of KareCoder and other methods on Simple, Medium and Hard difficulties of CodeF Post2021-9.

Method	*Simple*			*Medium*			*Hard*		
	Pass@1	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*
ChatGPT	26.1	39.8	46.2	7.0	15.0	19.5	6.0	10.4	12.4
ChatGPT+Plan	28.5	31.8	33.6	5.1	7.6	8.4	7.3	8.6	9.0
ChatGPT+SCOT	19.1	24.3	25.2	6.4	11.3	13.1	**7.7**	**11.6**	12.4
ChatGPT+SCOT&KareCoder	24.7	25.0	25.0	7.7	9.2	9.6	7.3	8.0	8.0
ChatGPT+KareCoder	**30.5**	**38.8**	**41.8**	**10.5**	**14.4**	**16.4**	7.4	11.1	**12.5**
Relative Improvement	7.0% ↑	22.0% ↑	24.4% ↑	36.4% ↑	27.4% ↑	25.2% ↑	-3.9% ↓	-4.3% ↓	0.8% ↑
ChatGPT+Plan(3-shot)	29.7	31.9	32.4	7.1	**12.5**	**15.8**	6.5	7.9	8.5
ChatGPT+SCOT(3-shot)	29.9	30.0	30.0	**10.8**	12.2	12.5	7.7	9.4	9.8
ChatGPT+SCOT&KareCoder(3-shot)	29.7	30.0	30.0	9.1	10.5	10.9	8.8	9.8	9.9
ChatGPT+KareCoder(3-shot)	**31.5**	**34.1**	**34.9**	9.3	10.9	11.5	**9.6**	**11.7**	**12.5**
Relative Improvement	5.4% ↑	6.9% ↑	7.7% ↑	-13.9% ↓	-12.8% ↓	-27.2% ↓	9.1% ↑	19.4% ↑	26.3% ↑

Table 4: Comparative evaluation of KareCoder with other methods on the CodeF post2021-9.

Method	*Pass@1*	*Pass@3*	*Pass@5*
ChatGPT	12.9	22.4	26.9
ChatGPT+Plan	14.2	17.3	18.3
ChatGPT+SCOT	11.2	14.2	15.0
ChatGPT+SCOT&KareCoder	14.0	15.2	15.5
ChatGPT+KareCoder	**15.9**	**19.8**	**21.3**
Relative Improvement	12.0% ↑	14.5% ↑	16.4% ↑

more than the results generated directly by ChatGPT; conversely, in terms of the Pass@3 and Pass@5 metrics, ChatGPT's direct generation remains the most effective. The rationale behind this phenomenon has been comprehensively discussed in the previous part of this RQ. Compared with the Plan, SCOT and SCOT&KareCoder methods, KareCoder attains a relative improvement of 12.0%, 14.5% and 16.4% on the Pass@1, Pass@3 and Pass@5 metrics, respectively.

> ***Answer to RQ1:*** *Experiments on CodeF post2021-9 and its subsets—Simple, Medium and Hard show that KareCoder considerably enhances ChatGPT's performance on the Pass@1 metric. Nevertheless, due to constraints arising from the intermediate process, KareCoder does not outperform the direct usage of ChatGPT on the Pass@3 and Pass@5 metrics but maintains advantage over other methods when evaluated on the Pass@3 and Pass@5 metrics.*

6.2 RQ2: Effectiveness of Knowledge Type

1) Library Variants. The Knowledge Library is a critical component of KareCoder. This research proposes a Knowledge Library dedicated to algorithms and data structures, specifically designed for programming problems. This innovation aims to addresses ChatGPT's "knowledge forgetting" during solution planning, specifically the loss of necessary programming knowledge. For this research question, we investigated various Knowledge Library alternatives and contrasted different design possibilities. We built three distinct Knowledge Libraries, namely: Knowledge Pseudo-code, Knowledge Steps of Pseudo-code and the Knowledge Description employed by KareCoder. Further details regarding these diverse Knowledge Libraries can be found in Section 3.2.

2) Results. As demonstrated in Table 5, among the three Knowledge Librarys, the Knowledge Description yielded the most superior performance, achieving the highest scores across all difficulty levels on the Pass@1, Pass@3 and Pass@5 metrics. Notably, its relative improvement reached 15.1%, 16.7% and 13.8% on the Pass@1 metric. Moreover, KareCoder utilizing the Knowledge Description consistently outperforms the Plan method without a Knowledge Library. These results suggest that LLMs exhibit improved performance in handling novel problems when suitable external information is integrated. Conversely, the integration of unsuitable external information could potentially yield adverse effects.

> ***Answer to RQ2:*** *Based on the current analysis, the Knowledge Description of the Knowledge Library has exhibited the most promising results. Nonetheless, we conjecture that the performance of KareCoder could be further enhanced by constructing more refined Knowledge Libraries and implementing superior knowledge-importing techniques in the future.*

6.3 RQ3: Influence of Shot Times and Settings

In this RQ, we scrutinize the performance of KareCoder in relation to varying shot times and parameter settings for ChatGPT, thereby performing a stability analysis of the KareCoder method.

1) Impact of Shot Times. As indicated in Table 6, the results derived from 1-shot, 2-shot and 3-shot experiments show that, KareCoder's effectiveness does not significantly fluctuate in response to variations in shot times. When implementing ChatGPT, we must also account for constraints related to maximum input length. While the "gpt-3.5-turbo-16k-0613" model can accommodate greater input length, it is crucial to strike a balance between shot times and performance, taking into account considerations related to computational resources and associated costs. Taking these factors into account, it is reasonable to conclude that the 1-shot setting adequately caters to the vast majority of the method's requirements.

Table 5: Performance of KareCoder utilizing different Knowledge Libraries.

Knowledge Library	*Simple*			*Medium*			*Hard*		
	Pass@1	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*
Knowledge Description	**30.5**	**38.8**	**41.8**	**10.5**	**14.4**	**16.4**	**7.4**	**11.1**	**12.5**
Knowledge Pseudo-code	23.5	27.3	28.7	9.0	12.7	14.2	6.5	8.0	8.6
Knowledge Step of Pseudo-code	26.5	29.3	29.8	7.4	10.7	12.0	5.9	7.9	8.8
Relative Improvement	15.1% ↑	32.4% ↑	40.3% ↑	16.7% ↑	13.4% ↑	15.5% ↑	13.8% ↑	38.8% ↑	42.0% ↑

Table 6: Comparative evaluation of KareCoder with different shot times on various difficulties of CodeF post2021-9.

Shot Times	*Simple*			*Medium*			*Hard*		
	Pass@1	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*
1-shot	30.5	**38.8**	**41.8**	10.5	14.4	16.4	7.4	11.1	**12.5**
2-shot	30.7	34.2	36.1	**12.6**	**15.5**	**16.8**	6.0	8.0	9.0
3-shot	**31.5**	34.1	34.9	9.3	10.9	11.5	**9.6**	**11.7**	**12.5**
Relative Improvement	-3.2% ↓	13.5% ↑	15.8% ↑	-16.7% ↓	-7.1% ↓	-2.4% ↓	-22.9% ↓	-5.1% ↓	0%

Table 7: Comparative evaluation of KareCoder with different ChatGPT Settings on various difficulties of CodeF post2021-9. For convenience of representation, we use "Tem" to represent "Temperature".

ChatGPT Setings	*Simple*			*Medium*			*Hard*		
	Pass@1	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*	*Pass@1*	*Pass@3*	*Pass@5*
Tem=0.8 Top_P=0.95	29.4	33.7	35.4	8.7	10.6	11.8	**8.6**	10.2	10.7
Tem=0.9 Top_P=0.95	29.3	31.5	32.2	8.8	10.5	11.7	6.2	7.6	8.0
Tem=1 Top_P=1	**30.5**	**38.8**	**41.8**	**10.5**	**14.4**	**16.4**	7.4	**11.1**	**12.5**
Relative Improvement	3.7% ↑	15.1% ↑	18.1% ↑	19.3% ↑	35.8% ↑	39.0% ↑	-14.0% ↓	8.8% ↑	16.8% ↑

2) Impact of ChatGPT Settings. Within the RQ, we investigate the degree to which various parameter settings, such as Temperature and Top_P, impact the efficacy of KareCoder. As illustrated in Table 7, KareCoder performs optimally when both Temperature and Top_P are designated as 1. This may be attributed to the fact that we furnished ChatGPT with an external Knowledge Library and we did not impose stringent restrictions on the implementation of algorithms and data structures within the Knowledge Library. Instead, we intended the Knowledge Library to serve as a guide, while KareCoder necessitated a significant degree of randomness. Establishing Temperature=1 and Top_P=1 endows ChatGPT with greater randomness, thereby circumventing the generation of chained error code owing to errors encountered during the production of knowledge-aware prompt.

> ***Answer to RQ3:*** *Varying Shot times, 1-shot, 2-shot and 3-shot, each setting exhibits unique benefits and limitations. Nonetheless, when considering computational resource constraints, the 1-shot setting is already sufficient to meet the accuracy demands of code generation. As for the parameter configurations of ChatGPT, we select the default settings, specifically, Temperature=1 and Top_P=1, to preserve the randomness of the outputs generated by ChatGPT to enhance the effectiveness of KareCoder.*

6.4 RQ4: Impact of Different Datasets

In an effort to comprehensively assess the efficacy of KareCoder, we selected the first 500 problems from the open-source dataset, the APPS test set and contrasted KareCoder's performance with other methods on these problems. As indicated in Table 8, five methods were employed: direct code generation using ChatGPT, Plan, SCOT, SCOT&KareCoder and KareCoder, the performance of the other four methods has not been able to surpass ChatGPT, the phenomenon that has been verified in previous research [15]. In light of the experimental results, we hypothesize that this may be attributed to overlapping data within APPS that is present in the ChatGPT training set. During the training process of ChatGPT, the massive number of natural language-code pairs (<Text, Code>) serve as training data. Consequently, ChatGPT can resolve these problems without the prerequisite for problem planning.

In a comprehensive evaluation, KareCoder's performance is approximately on par with the best of other methods and does not manifest a marked superiority. This condition potentially be ascribed to ChatGPT's prior exposure to open-source problems in APPS, resulting in the additional knowledge failing to deliver anticipated guidance. Nonetheless, in the experiments contrasting these methods, KareCoder exhibits superior performance across the Pass@k metric compared to the Plan method, which employs

Table 8: Comparative evaluation of KareCoder with other methods on the top 500 problems in the APPS test set.

Method	*Pass@1*	*Pass@3*	*Pass@5*
ChatGPT	19.5	30.0	34.4
ChatGPT+Plan	9.2	11.1	11.9
ChatGPT+SCOT	**13.3**	14.2	14.4
ChatGPT+SCOT&KareCoder	13.0	15.0	**16.1**
ChatGPT+KareCoder	13.1	**15.2**	15.8
Relative Improvement	-1.5% ↓	1.3% ↑	-1.9% ↓

the same step-by-step prompt. Furthermore, the SCOT&KareCoder, which incorporates external knowledge, exhibits a slightly lower performance than SCOT alone on the Pass@1 metric but significantly surpasses SCOT on the Pass@3 and Pass@5 metrics. From this phenomenon, we can still discover the role of incorporating knowledge into code generation through a two-by-two comparison of Plan and KareCoder, SCOT and SCOT&KareCoder.

Answer to RQ4: *Through experimentation on the first 500 problems drawn from the APPS test set, the outcomes directly generated by ChatGPT markedly exceed those of the other four methods. Despite this, KareCoder retained advantages over the other three methods, KareCoder's performance is approximately on par with the best of other methods.*

7 THREATS TO VALIDITY

7.1 Internal Validity

Limited algorithms and data structures tags. Our experiments were conducted mainly on the CodeF dataset, the algorithms and data structures tags of CodeF are sourced from the CodeForces programming website. When testing on dataset with no tags, e.g., APPS, matching programing problems with the Knowledge Library is not feasible. To mitigate this issue, we devised prompts and 3-shot examples to aid ChatGPT in predicting tags, recommending suitable algorithms and data structures for problems lacking these tags. Although we have checked the tags generated by tag generator manually, the algorithms and data structures recommended may not always be the most fitting. The issue of recommending more appropriate algorithms and data structures for new problems remains an ongoing research challenge.

Limited Knowledge Library and methods of integrating knowledge. KareCoder has yet to achieve optimality in terms of the comprehensiveness of its Knowledge Library and the methods of knowledge integration. The current Knowledge Library equipped with KareCoder comprises information on algorithms and data structures. We use contextual prompts to integrate knowledge and problems and generate prompts that help solve programming problems. However, certain programming problems might necessitate using internet resources or extensions from Python's library. In the future, we intend to expand our Knowledge Library and enhance its quality, enabling KareCoder to access broader knowledge and constantly updating the ways in which KareCoder acquires knowledge to solve more complex programming problems.

7.2 External Validity

Limited dataset. At present, the specific training data of ChatGPT is not publicly available, leaving us uncertain if there is an overlap between the existing datasets and the training data of ChatGPT. If we use the training data as the test set for inference, the value of the research done would significantly diminish. Currently, we can only test using the CodeF dataset, and we are lacking other entirely new datasets to verify KareCoder's effectiveness in code generation. As the training data of ChatGPT gets updated, we may need to continually explore new data for research.

8 DISSCUCCION

In this study, we assess the effectiveness of KareCoder by multiple RQs. Initially, we conducted preliminary exploration on "gpt-3.5-turbo-0301", which was followed by expanded experiments on "gpt-3.5-turbo-0613". However, our investigation was constrained by limited computational resources, preventing tests on a broader range of LLMs. We aim to extend our testing to additional LLMs to ascertain the scalability and practicality of KareCoder. The code generation dataset CodeF, proposed in this research, is based on the ChatGPT3.5 training data until September 2021. It is important to note that the existing dataset partitioning might lose relevance due to updates in the ChatGPT training data. To address this, we have included release timestamps for each problem in the dataset. Users are encouraged to download CodeF from our GitHub repository and can re-split the dataset to ensure it remains uncontaminated and relevant for their specific model requirements. Leveraging our Knowledge Library, researchers can apply the approach of integrating programming knowledge to diverse tasks. Nonetheless, it is essential to acknowledge that creating new datasets for other tasks, incorporating algorithms and data structures tags for these tasks, demands a certain amount of human resource investment.

9 CONCLUSION

In this research, we aimed to circumvent the utilization of LLMs' training data and thus constructed a new code generation dataset CodeF. The experiment indicated that problems predating September 2021 surpassed those following this date in terms of the Pass@k metric. We crafted a Knowledge Library specifically tailored for programming algorithm problems, and introduced KareCoder. This method integrates algorithms and data structures knowledge into the intermediate process of code generation, specifically, during the generation of knowledge-aware prompt. Experimental results demonstrated that KareCoder notably outperforms other code generation methods based on LLMs, on the CodeF post2021-9 and the first 500 problems of APPS datasets. This substantiates the research value of integrating algorithms and data structures knowledge into the code generation process.

ACKNOWLEDGMENTS

The work is supported in part by the Natural Science Foundation of Shandong Province, China (Grant No. ZR2021MF059), the National Natural Science Foundation of China (Grant Nos. 62192731, 62072007, 62192733, 61832009, 62192730), the National Key R&D Program (Grant No. 2023YFB4503801) and the Key Program of Hubei (Grant No. JD2023008).

REFERENCES

[1] Gabriel Poesia, Alex Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations*, 2021.

[2] Sijie Shen, Xiang Zhu, Yihong Dong, Qizhi Guo, Yankun Zhen, and Ge Li. Incorporating domain knowledge through task augmentation for front-end javascript code generation. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1533–1543, 2022.

[3] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations*, 2022.

[4] Yihong Dong, Ge Li, and Zhi Jin. Codep: grammatical seq2seq model for general-purpose code generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 188–198, 2023.

[5] Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. Embedding api dependency graph for neural code generation. *Empirical Software Engineering*, 26:1–51, 2021.

[6] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[7] 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.

[8] OpenAI. Chatgpt. https://OpenAI.com/chatgpt, 2023. Accessed: 2023-07-29.

[9] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. Biobert: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics*, 36(4):1234–1240, 2020.

[10] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 8342–8360, 2020.

[11] 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.

[12] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021.

[13] OpenAI. Gpt-3.5. https://platform.OpenAI.com/docs/models/gpt-3-5, 2023. Accessed: 2023-07-29.

[14] Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language model. *arXiv preprint arXiv:2303.06689*, 2023.

[15] Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*, 2023.

[16] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

[17] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *arXiv preprint arXiv:2305.06599*, 2023.

[18] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. Latent predictor networks for code generation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 599–609, 2016.

[19] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. r18treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8984–8991, 2020.

[20] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, 2021.

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, 2020.

[22] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations*, 2022.

[23] 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. *Science*, 378(6624):1092–1097, 2022.

[24] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[25] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[26] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[27] Shuofei Qiao, Yixin Ou, Ningyu Zhang, Xiang Chen, Yunzhi Yao, Shumin Deng, Chuanqi Tan, Fei Huang, and Huajun Chen. Reasoning with language model prompting: A survey. *arXiv preprint arXiv:2212.09597*, 2022.

[28] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. *arXiv preprint arXiv:2305.04091*, 2023.

[29] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[30] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V Le, et al. Least-to-most prompting enables complex reasoning in large language models. In *The Eleventh International Conference on Learning Representations*, 2022.

[31] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[32] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

[33] Antti Laaksonen. Competitive programmer's handbook. *Preprint*, 5, 2017.