MapCoder: Multi-Agent Code Generation for Competitive Problem Solving

Md. Ashraful Islam[1], **Mohammed Eunus Ali**[1], **Md Rizwan Parvez**[2]
[1]Bangladesh University of Engineering and Technology
[2]Qatar Computing Research Institute (QCRI)
{mdashrafulpramanic, mohammed.eunus.ali}@gmail.com, mparvez@hbku.edu.qa

Abstract

Code synthesis, which requires a deep understanding of complex natural language (NL) problem descriptions, generation of code instructions for complex algorithms and data structures, and the successful execution of comprehensive unit tests, presents a significant challenge. Thus, while large language models (LLMs) demonstrate impressive proficiency in natural language processing (NLP), their performance in code generation tasks remains limited. In this paper, we introduce a new approach to code generation tasks leveraging the multi-agent prompting that uniquely replicates the full cycle of program synthesis as observed in human developers. Our framework, **MapCoder**, consists of four LLM agents specifically designed to emulate the stages of this cycle: recalling relevant examples, planning, code generation, and debugging. After conducting thorough experiments, with multiple LLMs ablations and analyses across eight challenging competitive problem-solving and program synthesis benchmarks—MapCoder showcases remarkable code generation capabilities, achieving their new state-of-the-art (pass@1) results—(HumanEval **93.9%**, MBPP **83.1%**, APPS **22.0%**, CodeContests **28.5%**, and xCodeEval **45.3%**). Moreover, our method consistently delivers superior performance across various programming languages and varying problem difficulties. We open-source our framework at `https://github.com/Md-Ashraful-Pramanik/MapCoder`.

1 Introduction

Computer Programming has emerged as an ubiquitous problem-solving tool that brings tremendous benefits to every aspects of our life (Li et al., 2022a; Parvez et al., 2018; Knuth, 1992). To maximize programmers' productivity, and enhance accessibility, automation in program synthesis is paramount. With the growth of LLMs, significant advancements have been made in program synthesis—driving us in an era where we can generate fully executable code, requiring no human intervention (Chowdhery et al., 2022; Nijkamp et al., 2022).

Despite LLMs' initial success and the scaling up of model size and data, many of these models still struggle to perform well on complex problem-solving tasks, especially in competitive programming problems (Austin et al., 2021). To mitigate this gap, in this paper, we introduce **MapCoder**: a **M**ulti-**A**gent **P**rompting Based Code Generation approach that can seamlessly synthesize solutions for competition-level programming problems.

Competitive programming or competition-level code generation, often regarded as the pinnacle of problem-solving, is an challenging task. It requires a deep comprehension of NL problem descriptions, multi-step complex reasoning beyond mere memorization, excellence in algorithms and data structures, and the capability to generate substantial code that produces desired outputs aligned with comprehensive test cases (Khan et al., 2023).

Early approaches utilizing LLMs for code generation employ a direct prompting approach, where LLMs generate code directly from problem descriptions and sample I/O (Chen et al., 2021a). Recent methods like chain-of-thought (Wei et al., 2022a) advocates modular or pseudo code-based generation to enhance planning and reduce errors, while retrieval-based approaches such as Parvez et al. (2021) leverage relevant problems and solutions to guide LLMs' code generations. However, gains in such approaches remains limited in such a complex task like code generation where LLMs' generated code often fails to pass the test cases and they do not feature bug-fixing schema (Ridnik et al., 2024).

A promising solution to the above challenge is self-reflection (Shinn et al., 2023; Chen et al., 2022), which iteratively evaluates the generated code against test cases, reflects on mistakes and

MapCoder：多智能体代码生成以解决竞争性问题

阿什拉夫·伊斯拉姆[1]，穆罕默德·尤努斯·阿里[1]，穆罕默德·里兹万·帕尔韦兹[2] [1]孟加拉国工程技术大学（BUET）[2]卡塔尔计算研究学院（QCRI）{mdashrafulpramanic, mohammed.eunus.ali}@gmail.com, mparvez@hbku.edu.qa,

摘要

代码合成，这需要深入理解复杂的自然语言（NL）问题描述，生成复杂算法和数据结构的代码指令，以及成功执行全面的单元测试，这提出了一个重大挑战。因此，尽管大型语言模型（LLMs）在自然语言处理（NLP）方面表现出令人印象深刻的熟练程度，但它们在代码生成任务中的表现仍然有限。在本文中，我们介绍了一种利用多智能体提示的新方法来生成代码任务，该方法独特地复制了人类开发者观察到的程序合成的完整周期。我们的框架MapCoder由四个专门设计来模拟此周期各个阶段的LLM智能体组成：回忆相关示例、规划、代码生成和调试。经过彻底的实验，包括多个LLMs的消融实验和跨多个基准的分析，涵盖了八个具有挑战性的竞争问题解决和程序合成基准测试——MapCoder展示了卓越的代码生成能力，实现了新的最先进（pass@1）结果——（HumanEval 93.9%，MBPP 83.1%，APPS 22.0%，CodeContests 28.5%，和xCodeEval 45.3%）。此外，我们的方法在各种编程语言和不同问题难度上始终表现出优异的性能。我们在https://github.com/Md-Ashraful-Pramanik/MapCoder上开源了我们的框架。

1 引言

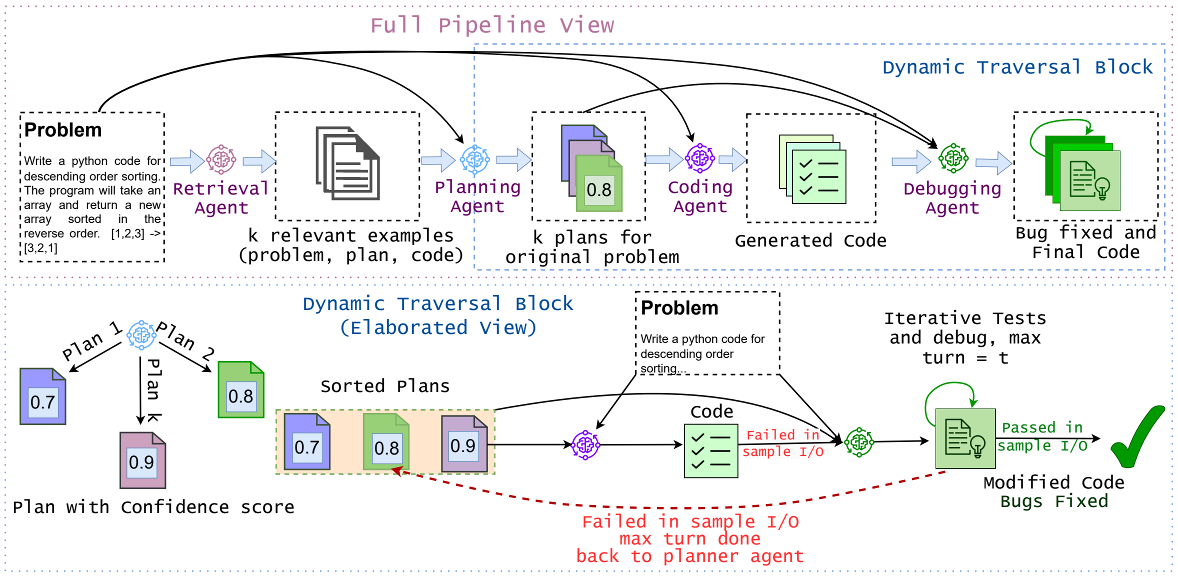
计算机编程已成为一种无处不在的问题解决工具，为我们的生活的各个方面带来了巨大的好处（Li等人，2022a；Parvez等人，2018；Knuth，1992）。为了最大化程序员的效率并提高可访问性，程序综合中的自动化至关重要。随着大型语言模型的增长，意义非凡。在程序合成方面取得了进展——引领我们进入一个可以生成完全可执行的代码的时代，无需人工干预（Chowdhery等人，2022年；Nijkamp等人，2022年）。

尽管LLMs（大型语言模型）最初取得了成功，并且模型规模和数据量都在扩大，但许多此类模型在复杂问题解决任务上仍然难以良好表现，尤其是在竞争性编程问题中（Austin等人，2021）。为了缩小这一差距，本文提出MapCoder：一种基于多智能体提示的代码生成方法，该方法能够无缝合成用于竞争级编程问题的解决方案。

竞技编程或竞赛级代码生成，常被视为解决问题的巅峰，是一项具有挑战性的任务。它需要深入理解自然语言问题描述，超越简单记忆的多步复杂推理，在算法和数据结构方面的卓越能力，以及生成大量代码的能力，这些代码能够与全面测试案例相匹配的预期输出（Khan等人，2023）。

早期利用大型语言模型进行代码生成的方法采用直接提示方法，其中大型语言模型直接从问题描述和示例I/O的生成代码（Chen等人，2021a）。最近的方法，如思维链（Wei等人，2022a）提倡模块化或伪代码生成的，以增强规划和减少错误，而基于检索的方法，如Parvez等人（2021）利用相关问题和解决方案来指导大型语言模型的代码生成。然而，在如此复杂的任务（如代码生成）中，此类方法的收益仍然有限，因为大型语言模型生成的代码往往无法通过测试用例，并且它们没有包含错误修复方案（Ridnik等人，2024）。

一个解决上述挑战的有希望的方法是自我审视（Shinn 等人（2023）；Chen 等人（2022）），该方法迭代地评估生成的代码与测试用例，反思错误之处并改进代码。



Full Pipeline View
Dynamic Traversal Block
Problem
Write a python code for descending order sorting. The program will take an array and return a new array sorted in the reverse order. [1,2,3] -> [3,2,1]
Retrieval Agent
k relevant examples (problem, plan, code)
Planning Agent
0.8
k plans for original problem
Coding Agent
Generated Code
Debugging Agent
Bug fixed and Final Code
Dynamic Traversal Block (Elaborated View)
Plan 1
Plan 2
Plan k
0.7
0.8
0.9
Plan with Confidence score
Sorted Plans
0.7
0.8
0.9
Problem
Write a python code for descending order sorting...
Code
Failed in sample I/O
Iterative Tests and debug, max turn = t
Passed in sample I/O
Modified Code Bugs Fixed
Failed in sample I/O max turn done back to planner agent

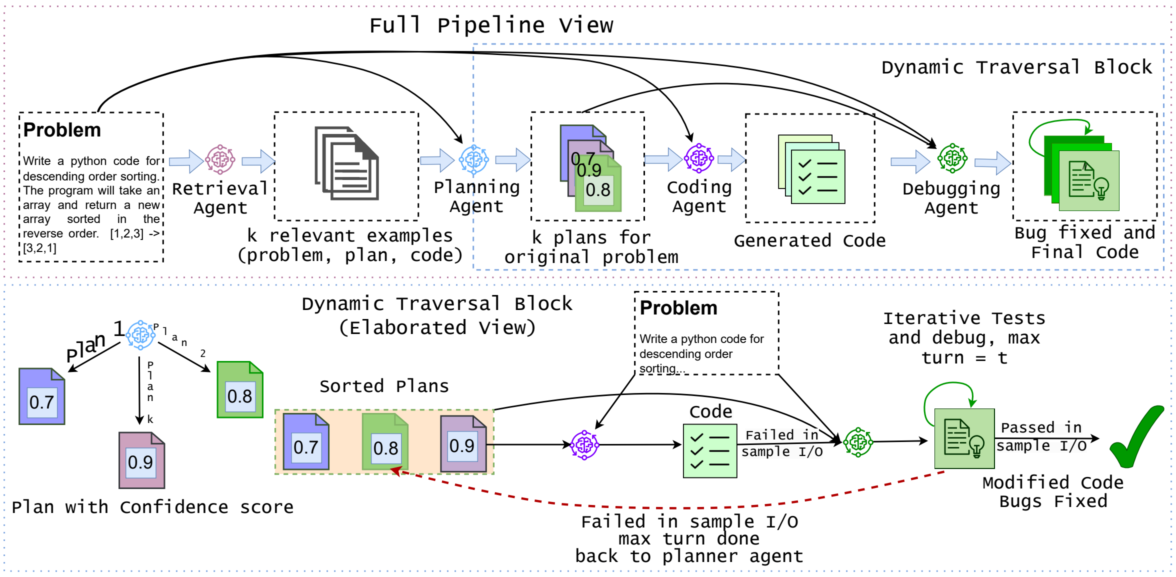
Figure 1: Overview of MapCoder (top). It starts with a retrieval agent that generates relevant examples itself, followed by planning, coding, and iterative debugging agents. Our dynamic traversal (bottom) considers the confidence of the generated plans as their reward scores and leverages them to guide the code generation accordingly.

modifies accordingly. However, such approaches have limitations too. Firstly, while previous studies indicate that superior problem-solving capabilities are attained when using in-context exemplars (Shum et al., 2023; Zhang et al., 2022; Wei et al., 2022a) or plans (Jiang et al., 2023b), these approaches, during both code generation and debugging, only leverage the problem description itself in a zero-shot manner. Consequently, their gains can be limited.

To confront the above challenge, we develop MapCoder augmenting the generation procedure with possible auxiliary supervision. We draw inspiration from human programmers, and how they use various signals/feedback while programming. The human problem-solving cycle involves recalling past solutions, planning, code writing, and debugging. MapCoder imitates these steps using LLM agents - retrieval, planning, coding, and debugging. In contrast to relying on human annotated examples, or external code retrieval models, we empower our retrieval agent to autonomously retrieve relevant problems itself (Yasunaga et al., 2023). Moreover, we design a novel structured pipeline schema that intelligently cascades the LLM agents and incorporates a dynamic iteration protocol to enhance the generation procedure at every step. Figure 1 shows an overview of our approach, MapCoder .

Additionally, existing iterative self-reflection methods rely on extra test cases generated by LLM agents (e.g., AgentCoder (Huang et al., 2023), LATS (Zhou et al., 2023), self-reflection (Shinn et al., 2023)) or external tools, compounding the challenges. Test case generation is equally challenging as code generation (Pacheco et al., 2007), and incorrect test cases can lead to erroneous code. Blindly editing code based on these test cases can undermine problem-solving capabilities. For instance, while self-reflection boosts GPT-4's performance on the HumanEval dataset, it drops by 3% on the MBPP dataset (Shinn et al., 2023). Upon identification, to validate this, on the HumanEval dataset itself, we replace their GPT-4 with ChatGPT, and see that model performance drops by 26.3%. Therefore, our debugging agent performs unit tests and bug fixing using only the sample I/O, without any artifact-more plausible for real-world widespread adoption.

We evaluate MapCoder on seven popular programming synthesis benchmarks including both basic programming like HumanEval, MBPP and challenging competitive program-solving benchmarks like APPS, CodeContests and xCodeEval. With multiple different LLMs including ChatGPT, GPT-4, and Gemini Pro, our approach significantly enhances their problem-solving capabilities - consistently achieving new SOTA performances, outperforming strong baselines like Reflexion (Shinn et al., 2023), and AlphaCodium (Ridnik et al., 2024). Moreover, our method consistently delivers superior performance across various programming languages and varying problem difficulties. Furthermore, with detailed ablation studies, we analyze MapCoder to provide more insights.



Full Pipeline View
Dynamic Traversal Block
Problem
Write a python code for descending order sorting. The program will take an array and return a new array sorted in the reverse order. [1,2,3] -> [3,2,1]
Retrieval Agent
k relevant examples (problem, plan, code)
Planning Agent
k plans for original problem
Coding Agent
Generated Code
Debugging Agent
Bug fixed and Final Code
Dynamic Traversal Block (Elaborated View)
Plan 1
Plan 2
Plan k
Plan with Confidence score
Sorted Plans
Problem
Write a python code for descending order sorting...
Code
Failed in sample I/O
Iterative Tests and debug, max turn = t
Passed in sample I/O
Modified Code Bugs Fixed
Failed in sample I/O max turn done back to planner agent

图1：MapCoder概述（顶部）。它从生成相关示例的检索代理开始，然后是规划、编码和迭代调试代理。我们的动态遍历（底部）考虑生成计划的置信度作为它们的奖励分数，并据此引导代码生成。

相应修改。然而，这种方法的局限性也很多。首先，尽管先前的研究表明，在使用上下文示例（Shum等人，2023；张等人，2022；魏等人，2022a）或策略（江等人，2023b）时，可以获得更优越的问题解决能力，但这些方法在代码生成和调试过程中，仅以零样本方式依赖问题描述本身。因此，其收益可能有限。

为了应对上述挑战，我们开发了MapCoder，通过可能的辅助监督增强生成过程。我们从人类程序员那里汲取灵感，了解他们在编程时如何使用各种信号/反馈。人类问题解决周期包括回忆过去的解决方案、规划、编写代码和调试。MapCoder通过使用LLM代理来模仿这些步骤——检索、规划、编码和调试。与依赖于人类标注的示例或外部代码检索模型相比，我们赋予我们的检索代理自主检索相关问题的能力（Yasunaga等人，2023）。此外，我们设计了一种新颖的结构化管道架构，智能地级联LLM代理，并纳入动态迭代协议，以增强每一步的生成过程。图1展示了我们的方法概述，MapCoder方法概述。

此外，现有的迭代自反方法依赖于LLM代理生成的额外测试用例（例如，AgentCoder（黄等，2023年），LATS（周等，2023年），自我反思（Shinn，2023年））。等，2023) 或外部工具，增加了挑战。测试用例生成与代码生成同样具有挑战性（Pacheco 等，2007），错误的测试用例可能导致错误代码。盲目地根据这些测试用例编辑代码会削弱解决问题的能力。例如，虽然自我反思提高了 GPT-4 在 HumanEval 数据集上的性能，但在 MBPP 数据集上下降了 3%（Shinn 等，2023）。在识别出这一点后，为了验证这一点，我们在 HumanEval 数据集本身上用 Chat-GPT 替换他们的 GPT-4，发现模型性能下降了 26.3%。因此，我们的调试代理仅使用样本 I/O 进行单元测试和错误修复，没有任何人工制品——更符合现实世界的广泛应用。

我们在七个流行的编程合成基准测试中评估MapCoder，包括像HumanEval、MBPP这样的基本编程和像APPS、CodeContests和xCodeEval这样的具有挑战性的竞赛编程基准。通过使用包括ChatGPT、GPT-4和Gemini Pro在内的多个不同的大型语言模型，我们的方法显著增强了它们的问题解决能力——持续实现新的SOTA性能，优于像Reflexion（Shinn等人，2023年）和AlphaCodium（Ridnik等人，2024年）这样的强大基线。此外，我们的方法在各种编程语言和不同难度的问题上始终提供卓越的性能。此外，通过详细的消融研究，我们分析了MapCoder以提供更多洞见。

2 Related Work

Program Synthesis: Program synthesis has a long standing history in AI systems (Manna and Waldinger, 1971). A large number of prior research attempted to address it via search/data flow approaches (Li et al., 2022a; Parisotto and Salakhutdinov, 2017; Polozov and Gulwani, 2015; Gulwani, 2011). LMs, prior to LLMs, attempt to generate code by fine-tuning (i.e., training) neural language models (Wang et al., 2021; Ahmad et al., 2021; Feng et al., 2020; Parvez et al., 2018; Yin and Neubig, 2017; Hellendoorn and Devanbu, 2017; Rabinovich et al., 2017; Hindle et al., 2016), conversational intents or data flow features (Andreas et al., 2020; Yu et al., 2019).

Large Language Models: Various LLMs have been developed for Code synthesis (Li et al., 2022b; Fried et al., 2022; Chen et al., 2021b; Austin et al., 2021; Nijkamp et al., 2022; Allal et al., 2023). Recent open source LLMs include Llama-2 (Touvron et al., 2023), CodeLlama-2 (Roziere et al., 2023), Mistral (Jiang et al., 2023a) Deepseek Coder (Guo et al., 2024), MoTCoder (Li et al., 2023) that are capable of solving many basic programming tasks.

Approach	Self-retrieval	Planning	Additional test cases generation	Debugging
Reflexion	✗	✗	✓	✓
Self-planning	✗	✓	✗	✗
Analogical	✓	✓	✗	✗
AlphaCodium	✗	✗	✓	✓
MapCoder	✓	✓	✗	✓

Table 1: Features in code generation prompt techniques.

Prompting LLMs: As indicated in Section 1, LLM prompting can be summarized into three categories: retrieval (Yasunaga et al., 2023; Parvez et al., 2023, 2021); planning (Wei et al., 2022b; Jiang et al., 2023b); debugging (Ridnik et al., 2024; Chen et al., 2023, 2022; Le et al., 2022) apart from the direct code generation approaches. In contrast, we combine all these paradigms and bridge their gaps (See Table 1). Among others, in different contexts of generic problem-solving, Tree-of-thoughts (Yao et al., 2023), and Cumulative reasoning (Zhang et al., 2023) approaches consider a tree traversal approach to explore different sub-steps towards a solution while our code generation approach mirrors the human programming cycle through various LLM agents. Notably, our traversal does not rely on sub-steps toward the solution but instead utilizes different forms of complete solutions.

3 `MapCoder`

Our goal is to develop a multi-agent code generation approach for competitive problem-solving. In order to do so, our framework, `MapCoder`, replicates the human programming cycle through four LLM agents - retrieval, plan, code, and debug. We devise a pipeline sequence for `MapCoder`, intelligently cascading the agents in a structured way and enhancing each agent's capability by augmenting in-context learning signals from previous agents in the pipeline. However, not all the agent responses/outputs are equally useful. Therefore, additionally, `MapCoder` features an adaptive agent traversal schema to interact among corresponding agents dynamically, iteratively enhancing the generated code by, for example, fixing bugs, while maximizing the usage of the LLM agents. In this section, we first discuss the agents (as per the pipeline), their prompts, and interactions, followed by the dynamic agent traversal protocol in `MapCoder` towards code generation for competitive problem-solving.

3.1 Retrieval Agent

Our first agent, the *Retrieval Agent*, recalls past relevant problem-solving instances, akin to human memory. It finds k (user-defined) similar problems without manual crafting or external retrieval models. Instead, we leverage the LLM agent itself, instructing it to generate such problems. Our prompt extends the analogical prompting principles (Yasunaga et al., 2023), generating examples and their solutions simultaneously, along with additional metadata (e.g., problem description, code, and plan) to provide the following agents as auxiliary data. We adopt a specific sequence of instructions, which is crucial for the prompt's effectiveness. In particular, initially, we instruct the LLM to produce similar and distinct problems and their solutions, facilitating problem planning reverse-engineering. Then, we prompt the LLM to generate solution code step-by-step, allowing post-processing to form the corresponding plan. Finally, we direct the LLM to generate relevant algorithms and provide instructional tutorials, enabling the agent to reflect on underlying algorithms and generate algorithmically similar examples.

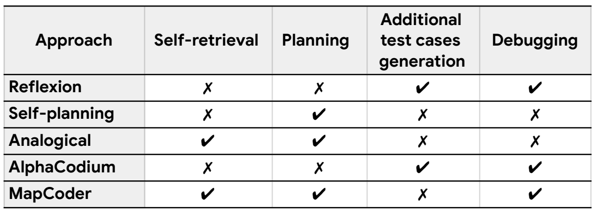
3.2 Planning Agent

The second agent, the *Planning Agent*, aims to create a step-by-step plan for the original problem. Our *Planning Agent* uses examples and their plans

2 相关工作。

程序综合：程序综合在人工智能系统中有着悠久的历史（Manna和Waldinger，1971）。大量先前研究试图通过搜索/数据流方法来解决这一问题（Li等人，2022a；Parisotto和Salakhutdinov，2017；Polozov和Gulwani，2015；Gulwani，2011）。在LLMs之前，语言模型试图通过微调（即训练）神经语言模型来生成代码，或者通过对话意图或数据流特征（Andreas等人，2020；Yu等人，2019）。

大型语言模型：已经开发了各种用于代码合成的大型语言模型（Li等，2022b，Fried等，2022；Chen等，2021b；Austin等，2021；Nijkamp等，2022；Allal等，2023）。近期开源的大型语言模型包括Llama-2（Touvron等，2023）、CodeLlama-2（Roziere等，2023）、Mistral（Jiang等，2023a）、Deepseek Coder（Guo等，2024）、MoTCoder（Li等，2023），它们能够解决许多基本的编程任务。



Approach	Self-retrieval	Planning	Additional test cases generation	Debugging
Reflexion	✗	✗	✔	✔
Self-planning	✗	✔	✗	✗
Analogical	✔	✔	✗	✗
AlphaCodium	✗	✗	✔	✔
MapCoder	✔	✔	✗	✔

表1：代码生成提示技术要素。

LLM提示技术：如第1节所示，LLM提示技术可以总结为三类：检索（Yasunaga等人，2023；Parvez等人，2023，2021）；规划（Wei等人，2022b；Jiang等人，2023b）；调试（Ridnik等人，2024；Chen等人，2023，2022；Le等人，2022），除了直接的代码生成方法。相比之下，我们将这些范式结合起来，弥合它们之间的鸿沟（见表1）。在众多方法中，在通用问题解决的多种情境下，思维树（Yao等人，2023）和累积推理（Zhang等人，2023）方法采用树遍历方法来探索通向解决方案的不同子步骤，而我们的代码生成方法通过多种LLM代理反映了人类编程周期。值得注意的是，我们的遍历不依赖于通向解决方案的子步骤，而是利用不同形式的完整解决方案。

3 地图编码器

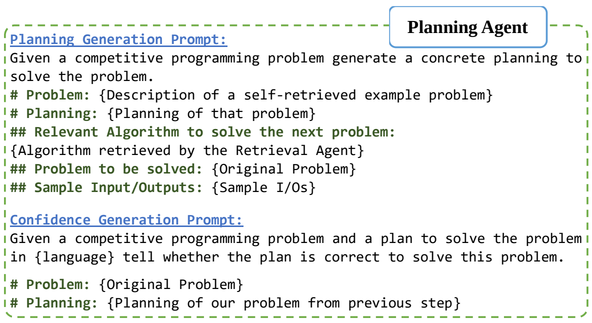
我们的目标是开发一种多智能体代码生成方法，用于竞争性问题解决。为此，我们的框架MapCoder通过四个LLM智能体——检索、计划、代码和调试——来复现人类编程周期。我们为MapCoder设计了一个管道序列，以结构化的方式智能地级联智能体，并通过增强来自管道中先前智能体的上下文学习信号来提高每个智能体的能力。然而，并非所有智能体的响应/输出都同样有用。因此，此外，MapCoder还具备自适应智能体遍历方案，以动态地交互相应的智能体，通过例如修复错误等方式，迭代地增强生成的代码，同时最大化LLM智能体的使用。在本节中，我们首先讨论智能体（按照管道顺序）、它们的提示和交互，然后讨论MapCoder中针对竞争性问题解决的代码生成的动态智能体遍历协议。

3.1 检索的代理

我们的第一个智能体，即 *Retrieval Agent*，能够回忆过去相关的解决问题的实例，类似于人类的记忆。它能够找到 k (用户定义的)相似问题，无需人工制作或使用外部检索模型。相反，我们利用LLM代理本身，指导其生成这些问题。我们的提示扩展了类比提示原理（Yasunaga等人，2023年提出），同时生成示例及其解决方案，以及额外的元数据（例如，问题描述、代码，和计划），为以下智能体提供辅助数据。我们采用特定的指令序列，这对于提示的有效性至关重要。特别是，最初，我们指导LLM生成相似和不同的问题及其解决方案，以促进问题规划的反向工程。然后，我们提示LLM逐步生成解决方案代码，允许后续处理形成相应的计划。最后，我们指导LLM生成相关算法并提供相应的指导教程，使智能体能够反思其底层算法并生成算法上相似的示例。

3.2 规划代表

第二个代理，即 *Planning Agent*，旨在为原始问题创建一个逐步计划。我们的 *Planning Agent* 采用示例及其计划，使用示例及其计划，以确保一致性。


Planning Agent

Planning Generation Prompt:
```
Given a competitive programming problem generate a concrete planning to
solve the problem.
# Problem: {Description of a self-retrieved example problem}
# Planning: {Planning of that problem}
## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by the Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample I/Os}
```
Confidence Generation Prompt:
```
Given a competitive programming problem and a plan to solve the problem
in {language} tell whether the plan is correct to solve this problem.
# Problem: {Original Problem}
# Planning: {Planning of our problem from previous step}
```

Figure 2: Prompt for *Planning Agent*.

obtained from the retrieval agent to generate plans for the original problem. A straightforward approach would be to utilize all examples collectively to generate a single target plan. However, not all retrieved examples hold equal utility. Concatenating examples in a random order may compromise the LLM's ability to generate accurate planning. For instance, Xu et al. (2023) demonstrated that even repeating more relevant information (e.g., query) towards the end of the in-context input aids LLM reasoning more effectively than including relatively less relevant contexts. A similar conclusion of "separating noisy in-context data" can also be drawn from the state-of-the-art retrieval augmented generation approaches like Wang et al. (2023). Therefore, we generate a distinct target plan for each retrieved example. Additionally, multiple plans offer diverse pathways to success.

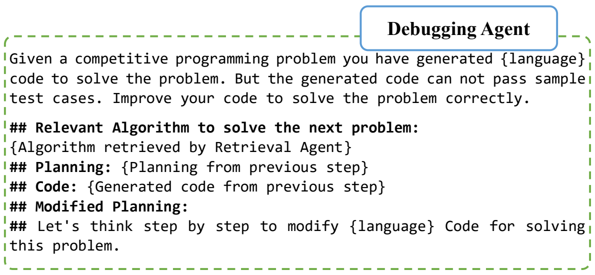
To help the generation steps in the following agents with the utility information for each plan, our designed prompt for the planning agent asks the LLM to generate both plans and a confidence score. Figure 2 shows our prompt got this agent.

3.3 Coding Agent

Next is the *Coding Agent*. It takes the problem description, and a plan from the *Planning Agent* as input and translates the corresponding planning into code to solve the problem. During the traversing of agents, *Coding Agent* takes the original problem and one particular plan from the *Planning Agent*, generates the code, and test on sample I/O. If the initial code fails, the agent transfers it to the next agent for debugging. Otherwise, predicts that as the final solution.

3.4 Debugging Agent

Finally, the *Debugging Agent* utilizes sample I/O from the problem description to rectify bugs in the generated code. Similar to humans cross-checking their plan while fixing bugs, our pipeline supplements the *Debugging Agent* with plans from the *Planning Agent*. This plan-derived debugging significantly enhances bug fixing in `MapCoder`, underscoring the pivotal roles played by both the *Debugging Agent* and the *Planning Agent* in the generation process. We verify this in Section 6. For each plan, this process is repeated t times. The prompt for this step is illustrated in Figure 3. Note that, different from Reflexion (Shinn et al., 2023) and AlphaCodium (Ridnik et al., 2024), our *Debugging Agent* does not require any additional test case generation in the pipeline.


Debugging Agent

```
Given a competitive programming problem you have generated {language}
code to solve the problem. But the generated code can not pass sample
test cases. Improve your code to solve the problem correctly.
## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving
this problem.
```

Figure 3: Prompt for *Debugging Agent*.

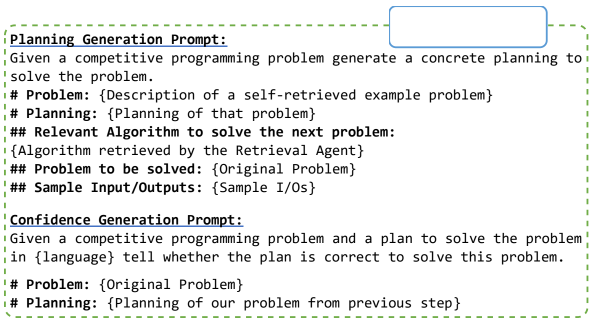
3.5 Dynamic Agent Traversal

The dynamic traversal in `MapCoder` begins with the *Planning Agent*, which outputs the plans for the original problem with confidence scores. These plans are sorted, and the highest-scoring one is sent to the Coding Agent. The Coding Agent translates the plan into code, tested with sample I/Os. If all pass, the code is returned; otherwise, it's passed to *Debugging Agent*. They attempt to rectify the code iteratively up to t times. If successful, the code is returned; otherwise, responsibility shifts back to the *Planning Agent* for the next highest confidence plan. This iterative process continues for k iterations, reflecting a programmer's approach. We summarize our agent traversal in Algorithm A in Appendix. Our algorithm's complexity is $O(kt)$. An example illustrating `MapCoder`'s problem-solving compared to Direct, Chain-of-thought, and Reflexion approaches is in Figure 4. All detailed prompts for each agent are in Appendix B.

4 Experimental Setup

4.1 Datasets

For extensive evaluation, we have used eight benchmark datasets: five from basic programming and three from complex competitive programming domains. Five basic programming datasets are:



```
Planning Generation Prompt:
Given a competitive programming problem generate a concrete planning to
solve the problem.
# Problem: {Description of a self-retrieved example problem}
# Planning: {Planning of that problem}
## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by the Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample I/Os}

Confidence Generation Prompt:
Given a competitive programming problem and a plan to solve the problem
in {language} tell whether the plan is correct to solve this problem.

# Problem: {Original Problem}
# Planning: {Planning of our problem from previous step}
```

图2：提示输入 *Planning Agent*。

从检索代理获取以生成原始问题的计划。一种直接的方法是利用所有示例。然而，并非所有检索到的示例都具有同等效用。以随机顺序连接示例可能会损害LLM生成准确计划的能力。例如，Xu等人（2023）证明了即使在上下文输入的末尾重复更相关的信息（例如查询），也比包括相对不相关的上下文更有助于LLM推理。从最先进的检索增强生成方法（如Wang等人，2023）中也可以得出“分离噪声上下文数据”的类似结论。因此，我们为每个检索到的示例生成一个不同的目标计划。此外，多个计划提供了通往成功的不同途径。

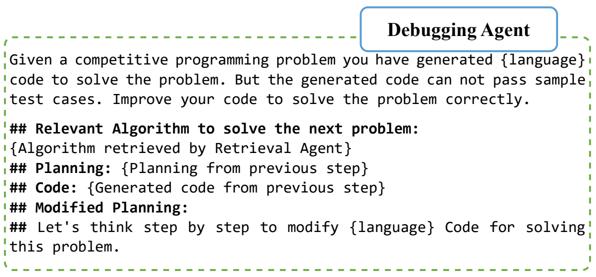
为了帮助以下代理的生成步骤，我们为规划代理设计的提示要求大型语言模型生成计划和置信度得分。如图2所示，我们的提示实现了此代理。

3.3 编码代理（软件）

接下来是 *Coding Agent*。它接收问题描述和来自 *Planning Agent* 的计划作为输入，并将相应的计划转换成代码以解决问题。在遍历代理节点期间，*Coding Agent* 接收原始问题以及来自 *Planning Agent* 的一个特定计划，生成代码，并在示例输入/输出上进行测试。如果初始代码失败，将其转交给下一个代理进行调试。否则，将其预测为最终解决方案。

3.4 调试工具

最后，*Debugging Agent*利用问题描述中的示例输入/输出来修复生成代码中的错误，类似于人类以某种方式交叉检查。他们的计划在修复错误的同时制定，我们的管道通过*Planning Agent*的计划补充了*Debugging Agent*。这种基于计划的调试显著提升了MapCoder的错误修复能力，凸显了*De-bugging Agent*和*Planning Agent*在其中的关键作用。我们将在第6节中对此进行验证。对于每个计划，这个过程会重复t次。图3展示了这一步骤的提示。请注意，与Reflexion（Shinn等，2023）和AlphaCodium（Ridnik等，2024）不同，我们的*De- bugging Agent*在管道中不需要生成任何额外的测试用例。



Debugging Agent

```
Given a competitive programming problem you have generated {language}
code to solve the problem. But the generated code can not pass sample
test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving
this problem.
```

图3：提示信息：版本14。

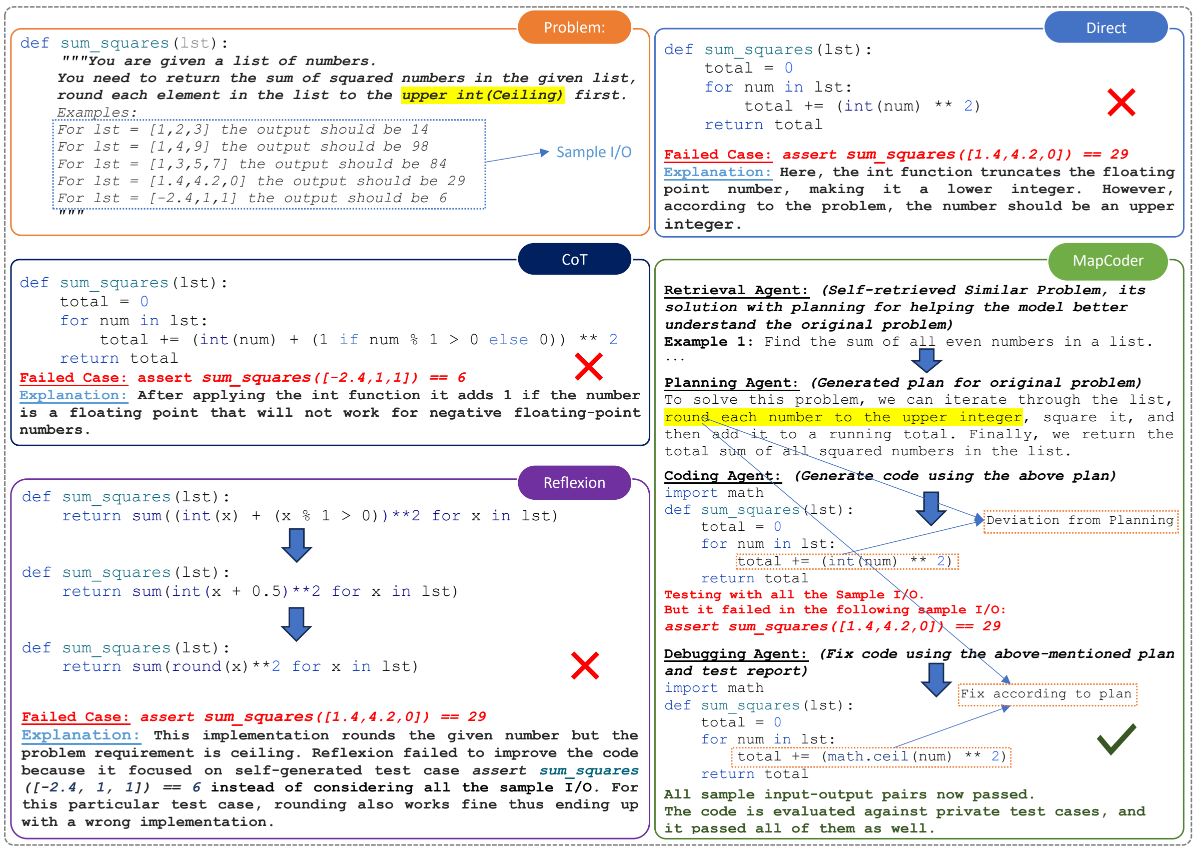
3.5 动态代理遍历

动态遍历过程在MapCoder中从*Planning Agent*开始，它输出原始问题的计划及其置信度评分。这些计划被排序，得分最高的计划被发送给编码代理。编码代理将计划转换为代码，并使用示例输入/输出进行测试。如果全部测试通过，则返回代码；否则，将其传递给*Debugging Agent*。它们尝试最多迭代t次以修正代码。如果成功，则返回代码；否则，责任转交给*Planning Agent*以处理下一个置信度最高的计划。此迭代过程持续进行k次，反映了程序员的处理方式。我们在附录A中总结了我们的代理遍历算法。我们算法的复杂度为$O(kt)$。图4展示了MapCoder解决问题的示例，与直接、思维链和反思方法进行对比。所有代理的详细提示均可在附录B中找到。

4 实验配置

4.1 数据集

我们使用了八个基准数据集进行广泛评估：五个来自基础编程领域，三个来自复杂竞技编程领域。五个基础编程数据集包括：- 数据集1- 数据集2- 数据集3- 数据集4- 数据集5 。



```
def sum_squares(lst):
    """You are given a list of numbers.
   You need to return the sum of squared numbers in the given list,
   round each element in the list to the upper int(Ceiling) first.
   Examples:
   For lst = [1,2,3] the output should be 14
   For lst = [1,4,9] the output should be 98
   For lst = [1,3,5,7] the output should be 84
   For lst = [1.4,4.2,0] the output should be 29
   For lst = [-2.4,1,1] the output should be 6
    """
```

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) ** 2)
    return total
```

Failed Case: *assert sum_squares([1.4,4.2,0]) == 29*
Explanation: Here, the int function truncates the floating point number, making it a lower integer. However, according to the problem, the number should be an upper integer.

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) + (1 if num % 1 > 0 else 0)) ** 2
    return total
```

Failed Case: *assert sum_squares([-2.4,1,1]) == 6*
Explanation: After applying the int function it adds 1 if the number is a floating point that will not work for negative floating-point numbers.

```
def sum_squares(lst):
    return sum((int(x) + (x % 1 > 0))**2 for x in lst)

def sum_squares(lst):
    return sum(int(x + 0.5)**2 for x in lst)

def sum_squares(lst):
    return sum(round(x)**2 for x in lst)
```

Failed Case: *assert sum_squares([1.4,4.2,0]) == 29*
Explanation: This implementation rounds the given number but the problem requirement is ceiling. Reflexion failed to improve the code because it focused on self-generated test case *assert sum_squares ([-2.4, 1, 1]) == 6* instead of considering all the sample I/O. For this particular test case, rounding also works fine thus ending up with a wrong implementation.

Retrieval Agent: ***(Self-retrieved Similar Problem, its solution with planning for helping the model better understand the original problem)***
Example 1: Find the sum of all even numbers in a list.
...

Planning Agent: ***(Generated plan for original problem)***
To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

Coding Agent: ***(Generate code using the above plan)***

```
import math
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) ** 2)
    return total
```

Testing with all the Sample I/O.
But it failed in the following sample I/O:
assert sum_squares([1.4,4.2,0]) == 29

Debugging Agent: ***(Fix code using the above-mentioned plan and test report)***

```
import math
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (math.ceil(num) ** 2)
    return total
```

All sample input-output pairs now passed.
The code is evaluated against private test cases, and it passed all of them as well.

Figure 4: Example problem and solution generation using Direct, CoT, Reflexion, and MapCoder prompts. MapCoder explores high-utility plans first and uniquely features a plan-derived debugging for enhanced bug fixing.

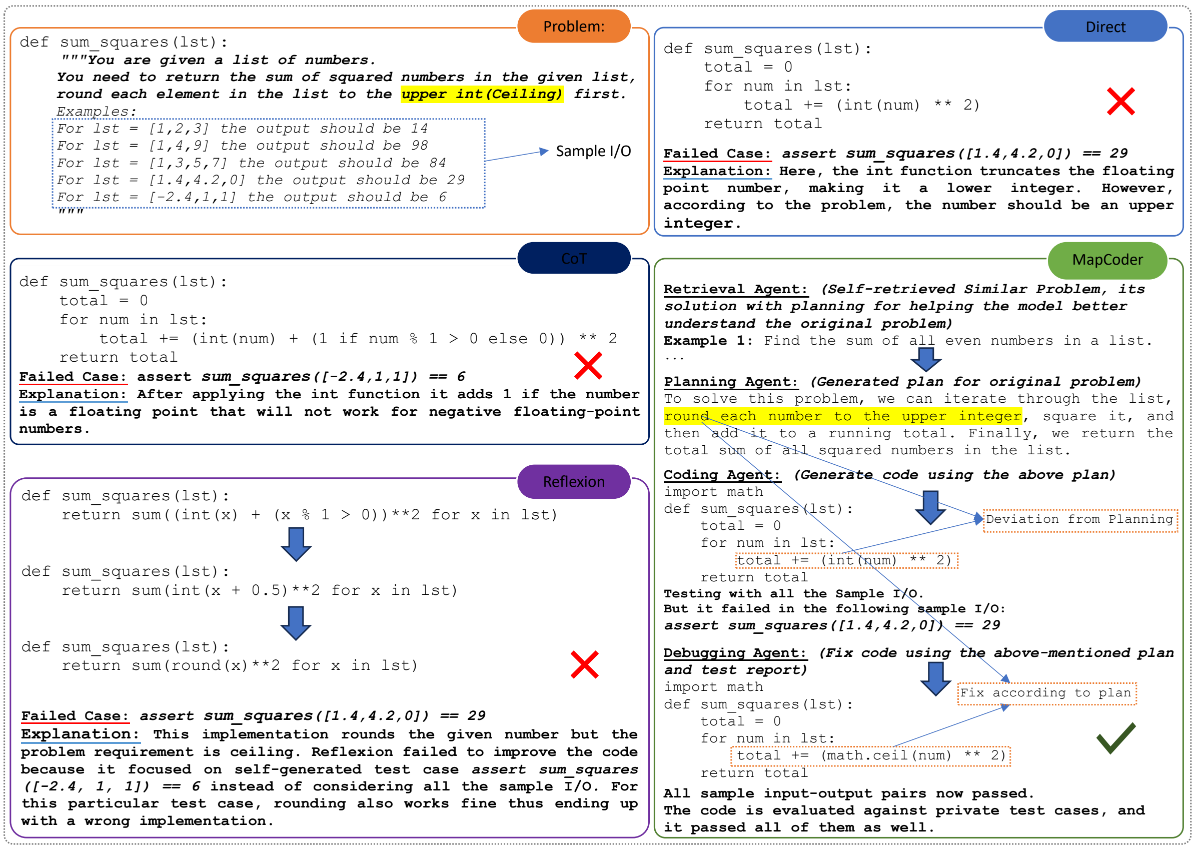
HumanEval (Chen et al., 2021a), **HumanEval-ET** (Dong et al., 2023a), **EvalPlus** (Liu et al., 2023), **MBPP**) (Austin et al., 2021), and **MBPP-ET** (Dong et al., 2023a). HumanEval-ET, EvalPlus extend HumanEval and MBPP-ET comprehends MBPP by incorporating more test cases. The problem set size of HumanEval and MBPP (and their extensions) are 164 and 397, respectively. Due to the absence of sample I/O in MBPP and MBPP-ET, our approach for code moderation involves randomly removing one test-case from MBPP-ET for each problem and provide this test-case as a sample I/O for the problem. Importantly, this removed test-case is carefully selected to ensure mutual exclusivity from the hidden test sets in MBPP and MBPP-ET. Three competitive programming datasets are: Automated Programming Progress Standard (**APPS**), **xCodeEval** (Khan et al., 2023), and **CodeContest**, where we have used 150, 106, and 156 problems, respectively, in our experiments.

4.2 Baselines

We have compared MapCoder with several baselines and state-of-the-art approaches. **Direct** Prompting instructs language models to generate code without explicit guidance, relying on their inherent capabilities of LLM. Chain of Thought Prompting (**CoT**) (Wei et al., 2022b) breaks down problems into step-by-step solutions, enabling effective tackling of complex tasks. **Self-Planning** Prompting (Jiang et al., 2023b) divides the code generation task into planning and implementation phases. **Analogical Reasoning** Prompting (Yasunaga et al., 2023) instructs models to recall relevant problems from training data. **Reflexion** (Shinn et al., 2023) provides verbal feedback to enhance solutions based on unit test results. **Self-collaboration** (Dong et al., 2023b) proposes a framework where different LLMs act as analyst, coder, and tester to cooperatively generate code for complex tasks, achieving better performance than directly using a single LLM. **AlphaCodium** (Ridnik et al., 2024) iteratively refines code based on AI-generated input-output tests.

4.3 Foundation Models, Evaluation Metric, k, and t

With $k = t = 5$ in HumanEval, and $k = t = 3$ for others, we evaluate all the datasets using ChatGPT (gpt-3.5-turbo-1106), GPT-4 (gpt-4-1106-preview)



```
def sum_squares(lst):
    """You are given a list of numbers.
    You need to return the sum of squared numbers in the given list,
    round each element in the list to the upper int(Ceiling) first.
    Examples:
    For lst = [1,2,3] the output should be 14
    For lst = [1,4,9] the output should be 98
    For lst = [1,3,5,7] the output should be 84
    For lst = [1.4,4.2,0] the output should be 29
    For lst = [-2.4,1,1] the output should be 6
    """
```

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) ** 2)
    return total
```

Failed Case: *assert sum_squares([1.4,4.2,0]) == 29*
Explanation: Here, the int function truncates the floating point number, making it a lower integer. However, according to the problem, the number should be an upper integer.

```
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) + (1 if num % 1 > 0 else 0)) ** 2
    return total
```

Failed Case: assert *sum_squares([-2.4,1,1]) == 6*
Explanation: After applying the int function it adds 1 if the number is a floating point that will not work for negative floating-point numbers.

```
def sum_squares(lst):
    return sum((int(x) + (x % 1 > 0))**2 for x in lst)

def sum_squares(lst):
    return sum(int(x + 0.5)**2 for x in lst)

def sum_squares(lst):
    return sum(round(x)**2 for x in lst)
```

Failed Case: *assert sum_squares([1.4,4.2,0]) == 29*
Explanation: This implementation rounds the given number but the problem requirement is ceiling. Reflexion failed to improve the code because it focused on self-generated test case *assert sum_squares ([-2.4, 1, 1]) == 6* instead of considering all the sample I/O. For this particular test case, rounding also works fine thus ending up with a wrong implementation.

Retrieval Agent: ***(Self-retrieved Similar Problem, its solution with planning for helping the model better understand the original problem)***
Example 1: Find the sum of all even numbers in a list.
...

Planning Agent: ***(Generated plan for original problem)***
To solve this problem, we can iterate through the list, round each number to the upper integer, square it, and then add it to a running total. Finally, we return the total sum of all squared numbers in the list.

Coding Agent: ***(Generate code using the above plan)***

```
import math
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (int(num) ** 2)
    return total
```

Testing with all the Sample I/O.
But it failed in the following sample I/O:
assert sum_squares([1.4,4.2,0]) == 29

Debugging Agent: ***(Fix code using the above-mentioned plan and test report)***

```
import math
def sum_squares(lst):
    total = 0
    for num in lst:
        total += (math.ceil(num) ** 2)
    return total
```

All sample input-output pairs now passed.
The code is evaluated against private test cases, and it passed all of them as well.

图4：使用Direct Prompt、CoT、Reflexion和MapCoder提示生成示例问题和解决方案。MapCoder首先探索高效用计划，并独特地提供基于计划的调试功能，以增强错误纠正。

HumanEval（陈等人，2021a），HumanEval-ET（董等人，2023a），EvalPlus（刘等人，2023），MBPP（奥斯汀等人，2021），以及MBPP-ET（董等人，2023a）。HumanEval-ET和EvalPlus是HumanEval的扩展，而MBPP-ET通过增加更多测试案例，从而包含MBPP。HumanEval及其扩展MBPP的问题集大小分别为164和397。由于MBPP和MBPP-ET缺乏样本I/O，我们的代码审查方法涉及对每个问题从MBPP-ET中随机删除一个测试案例，并将此测试案例作为问题的样本I/O提供。重要的是，这个被删除的测试案例经过精心挑选，以确保其与MBPP和MBPP-ET中的隐藏测试集不重叠。三个竞技编程数据集是：自动编程进度标准（APPS）、xCodeEval（Khan等人，2023）和CodeContest，在实验中，我们分别使用了150、106和156个问题。

4.2 基准线.

我们已将MapCoder与多个基线和最先进的方法进行了比较。直接提示（Direct Prompting）指导语言模型生成文本。这种方法旨在使模型能够根据给定的提示生成连贯、相关的输出。代码无需明确指导，依靠大型语言模型的内在能力。思维链提示（CoT）（Wei等人，2022b）将问题分解为逐步解决方案，使有效应对复杂任务成为可能。自我规划提示（Jiang等人，2023b）将代码生成任务分为规划和实施阶段。类比推理提示（Yasunaga等人，2023）指导模型回忆训练数据中的相关问题。反思（Shinn等人，2023）根据单元测试结果提供口头反馈以增强解决方案。自我协作（Dong等人，2023b）提出一个框架，其中不同的LLM分别担任分析师、编码者和测试员，协同生成复杂任务的代码，比直接使用单个LLM获得更好的性能。AlphaCodium（Ridnik等人，2024）根据AI生成的输入输出测试迭代优化代码。

4.3 基础模型、评估指标体系、版本0 和 版本1

使用 $k = t = 5$ 在 HumanEval 中，以及 $k = t = 3$ 对其他人，我们使用 ChatGPT（gpt-3.5-turbo-1106）和 GPT-4（gpt-4-1106-preview），评估所有数据集，其他人，。

		Simple Problems					Contest-Level Problems		
LLM	Approach	HumanEval	HumanEval ET	EvalPlus	MBPP	MBPP ET	APPS	xCodeEval	CodeContest
ChatGPT	Direct	48.1%	37.2%	66.5%	49.8%	37.7%	8.0%	17.9%	5.5%
	CoT	68.9%	55.5%	65.2%	54.5%	39.6%	7.3%	23.6%	6.1%
	Self-Planning	60.3%	46.2%	-	55.7%	41.9%	9.3%	18.9%	6.1%
	Analogical	63.4%	50.6%	59.1%	70.5%	46.1%	6.7%	15.1%	7.3%
	Reflexion	67.1%	49.4%	62.2%	73.0%	47.4%	-	-	-
	Self-collaboration	74.4%	56.1%	-	68.2%	49.5%	-	-	-
	MapCoder	**80.5%** ↑ 67.3%	**70.1%** ↑ 88.5%	**71.3%** ↑ 7.3%	**78.3%** ↑ 57.3%	**54.4%** ↑ 44.3%	**11.3%** ↑ 41.3%	**27.4%** ↑ 52.6%	**12.7%** ↑ 132.8%
GPT4	Direct	80.1%	73.8%	81.7%	81.1%	54.7%	12.7%	32.1%	12.1%
	CoT	89.0%	61.6%	-	82.4%	56.2%	11.3%	36.8%	5.5%
	Self-Planning	85.4%	62.2%	-	75.8%	50.4%	14.7%	34.0%	10.9%
	Analogical	66.5%	48.8%	62.2%	58.4%	40.3%	12.0%	26.4%	10.9%
	Reflexion	91.0%	78.7%	81.7%	78.3%	51.9%	-	-	-
	MapCoder	**93.9%** ↑ 17.2%	**82.9%** ↑ 12.4%	**83.5%** ↑ 2.2%	**83.1%** ↑ 2.5%	**57.7%** ↑ 5.5%	**22.0%** ↑ 73.7%	**45.3%** ↑ 41.2%	**28.5%** ↑ 135.1%

Table 2: Pass@1 results for different approaches. The results of the yellow and blue colored cells are obtained from Jiang et al. (2023b) and Shinn et al. (2023), respectively. The results of the Self-collaboration Dong et al. (2023b) paper are collected from their paper. The green texts indicate the state-of-the-art results, and the red text is gain over Direct Prompting approach.

from OpenAI and Gemini Pro from Google. We have also evaluated our method using an open-source LLM, Mistral-7B-instruct. We have used the Pass@k evaluation metric, where the model is considered successful if at least one of the k generated solutions is correct.

5 Results

In this section, we evaluate the code generation capabilities of our framework, `MapCoder`, for competitive problem solving. Our experimental results are reported in Table 2. Overall, `MapCoder` shows a tremendous excellence in code generation, significantly outperforms all baselines, and achieves new state-of-the-art results in all benchmarks. In general the scales with GPT-4 are higher than ChatGPT.

5.1 Performance on basic code generation

The highest scale of performance (Pass@1) scores are observed in simple program synthesis tasks like HumanEval, MBPP in Table 2. Though with the simpler problem (non-contests) datasets such as HumanEval, HumanEval-ET, the current state-of-the-art method, Reflexion (Shinn et al., 2023) perform reasonably well, this approach does not generalize across varying datasets depicting a wide variety of problems. Self-reflection techniques enhance GPT-4's performance on HumanEval but result in a 3% decrease on the MBPP dataset. Similarly, with ChatGPT, there's a notable 26.3% drop in performance where in several cases their AI generated test cases are incorrect. We observe that 8% of failures in HumanEval and 15% in MBPP is caused by their AI generates incorrect test cases while our approach is independent of AI test cases, and consistently improves code generations in general. Consequently, even in HumanEval, with GPT-4, our Pass@1 surpasses Reflexion by ∼3%. On top, in all four simple programming datasets, `MapCoder` enhances the Direct prompting significantly with a maximum of 88% on HumanEvalET by ChatGPT.

5.2 Performance on competitive problem solving

The significance of `MapCoder` shines through clearly when evaluated in competitive problem-solving contexts. Across datasets such as APPS, xCodeEval, and CodeContests, `MapCoder` demonstrates substantial enhancements over Direct prompting methods, with improvements of 41.3%, 52.6%, and 132.8% for ChatGPT, and 73.7%, 41.2%, and 135.1% for GPT4, respectively. Notably, the most challenging datasets are APPS and CodeContest, where `MapCoder`'s performance stands out prominently. We deliberately com-

		Simple Problems					Contest-Level Problems		
LLM	Approach	HumanEval	HumanEval ET	EvalPlus	MBPP	MBPP ET	APPS	xCodeEval	CodeContest
ChatGPT	Direct	48.1%	37.2%	66.5%	49.8%	37.7%	8.0%	17.9%	5.5%
	CoT	68.9%	55.5%	65.2%	54.5%	39.6%	7.3%	23.6%	6.1%
	Self-Planning	60.3%	46.2%	-	55.7%	41.9%	9.3%	18.9%	6.1%
	Analogical	63.4%	50.6%	59.1%	70.5%	46.1%	6.7%	15.1%	7.3%
	Reflexion	67.1%	49.4%	62.2%	73.0%	47.4%	-	-	-
	Self-collaboration	74.4%	56.1%	-	68.2%	49.5%	-	-	-
	MapCoder	**80.5%** **↑67.3%**	**70.1%** **↑88.5%**	**71.3%** **↑7.3%**	**78.3%** **↑57.3%**	**54.4%** **↑44.3%**	**11.3%** **↑41.3%**	**27.4%** **↑52.6%**	**12.7%** **↑132.8%**
GPT4	Direct	80.1%	73.8%	81.7%	81.1%	54.7%	12.7%	32.1%	12.1%
	CoT	89.0%	61.6%	-	82.4%	56.2%	11.3%	36.8%	5.5%
	Self-Planning	85.4%	62.2%	-	75.8%	50.4%	14.7%	34.0%	10.9%
	Analogical	66.5%	48.8%	62.2%	58.4%	40.3%	12.0%	26.4%	10.9%
	Reflexion	91.0%	78.7%	81.7%	78.3%	51.9%	-	-	-
	MapCoder	**93.9%** **↑17.2%**	**82.9%** **↑12.4%**	**83.5%** **↑2.2%**	**83.1%** **↑2.5%**	**57.7%** **↑5.5%**	**22.0%** **↑73.7%**	**45.3%** **↑41.2%**	**28.5%** **↑135.1%**

表2：不同方法的Pass@1结果。黄色和蓝色单元格的结果分别来自Jiang等人（2023b）和Shinn等人（2023）。Self-collaboration by Dong et al. (2023b)的研究结果来自他们的研究。绿色文本表示最先进的结果，红色文本则表示相对于Direct Prompting方法的增益。

来自OpenAI和Google的Gemini Pro（谷歌）。我们还使用开源语言模型（LLM）Mistral-7B-instruct（开源）评估了我们的方法。我们使用了Pass@k（评估指标）评估指标，其中如果至少有一个生成的版本v0解决方案是正确的，则模型被认为成功。

5 结果

本节中，我们评估了我们框架MapCoder在竞争性问题解决中的代码生成能力。我们的实验结果见表2。总体而言，MapCoder在代码生成方面表现出卓越的才能，显著优于所有基线，并在所有基准测试中实现了新的最先进结果。一般来说，与GPT-4相比，Chat-GPT的规模更高。

5.1 基本代码生成性能评估

在简单的程序综合任务（如HumanEval、MBPP）中观察到最高的Pass@1得分。尽管在更简单的问题（非竞赛类）数据集（如HumanEval、HumanEval-ET）中，当前最先进的方法Reflexion（Shinn等人，2023）表现相当不错，但这种方法并不能推广到跨越描述各种各样问题的不同数据集。自我反思技术能够提升，跨越描述各种各样问题的不同数据集。

GPT-4在HumanEval上的表现良好，但在MBPP数据集上结果下降了3%。同样，与ChatGPT相比，性能下降了26.3%，其中在几个案例中，它们生成的AI测试用例存在错误。我们观察到，HumanEval中有8%的失败是由它们生成的AI测试用例存在错误引起的，而我们的方法与AI测试用例无关，并且始终在一般情况下提高代码生成质量。因此，即使在HumanEval中，使用GPT-4，我们的Pass@1率也超过了Reflexion的v1版本3%。此外，在所有四个简单的编程数据集中，MapCoder通过ChatGPT显著提高了直接提示，最高达到HumanEval ET的88%。

5.2 竞争性问题解决能力表现

当在竞争性问题解决环境中评估时，MapCoder 的重要性表现得非常明显。在 APPS、xCodeEval 和 CodeContests 等数据集中，MapCoder 相比直接提示方法表现出显著的提升，ChatGPT 分别提高了 41.3%、52.6% 和 132.8%，GPT4 分别提高了 73.7%、41.2% 和 135.1%。值得注意的是，最具挑战性的数据集是 APPS 和 Code Contest，MapCoder 的性能在这里尤为突出。我们故意进行了评估，以突出 MapCoder 在这些数据集上的表现。

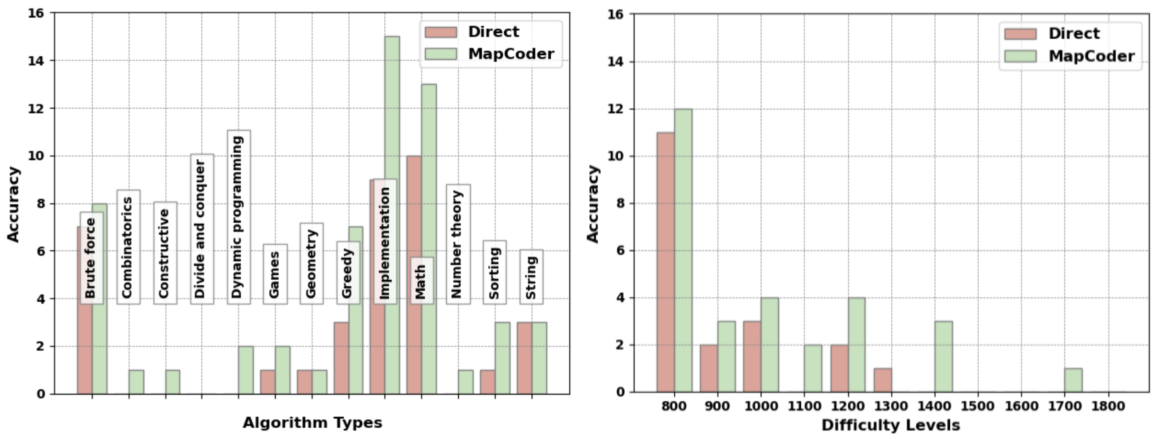


Figure 5: The number of correct answers wrt algorithm types (tags) and difficulty levels (xCodeEval dataset).

pare against strong baselines on these datasets, regardless of whether they are prompt-based or not. Importantly, on CodeContest our Pass@1 results match the Pass@5 scores of the concurrent state-of-the-art model AlphaCodium (Ridnik et al., 2024): 28.5% vs. their 29% (see Table 3). Furthermore, our Pass@5 results demonstrate an additional improvement of 12.8%. On APPS, `MapCoder` consistently surpasses the Pass@1 scores of all baseline prompts for both ChatGPT and GPT-4.

CodeContest (Pass@5)		
Approach	**ChatGPT**	**GPT4**
Direct	11.2%	18.8%
AlphaCodium	17.0%	29.0%
MapCoder	**18.2% (↑ 63.1%)**	**35.2% (↑ 87.1%)**

Table 3: Pass@5 results on CodeContest dataset. AlphCodium result are from Ridnik et al. (2024). The green cells indicate the SoTA and the red text indicates improvement w.r.t Direct approach.

5.3 Performance with Varying Difficulty Levels

The APPS dataset comprises problems categorized into three difficulty levels: (i) Introductory, (ii) Interview, and (iii) Competition. Figure 6 illustrates the performance of various competitive approaches for these three categories. The results reveal that our `MapCoder` excels across all problem categories, with highest gain in competitive problem-solving indicating its superior code generation capabilities in general, and on top, remarkable effectiveness in competitive problem-solving. In order to gather more understanding on what algorithm problems it's capable of solving and in fact much difficulty level it can solve, we have also conducted a comparison between `MapCoder` and the Direct approach, considering the difficulty levels[1] and tags[2] present in the xCodeEval dataset. The results of this comparison are depicted in Figure 5. This comparison showcases that `MapCoder` is effective across various algorithm types and exhibits superior performance even in higher difficulty levels, compared to the Direct approach. However, beyond (mid-level: difficulties>1000), its gains are still limited.

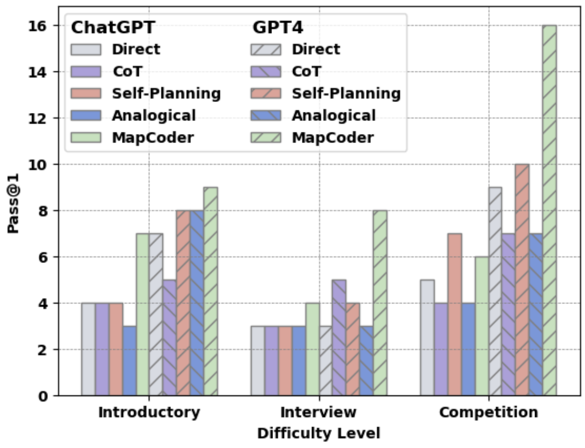


Figure 6: Performance vs problem types (APPS).

5.4 Performance Across Different LLMs

To show the robustness of `MapCoder` across various LLMs, we evaluate `MapCoder` using Gemini Pro, a different family of SoTA LLM in Table 4. We also evaluate `MapCoder` using an open-source LLM Mistral-7B instruct in Table 5. As expected, our method shows performance gains over other baseline approaches in equitable trends on both simple (HumanEval) and contest-level problems (CodeContest).

[1]Difficulty levels in xCodeEval dataset represents an integer number, a higher value means more difficult problem

[2]Tags in xCodeEval dataset represents algorithm type that can be used to solve the problem i.e., greedy, dp, brute-force, constructive, and so on.

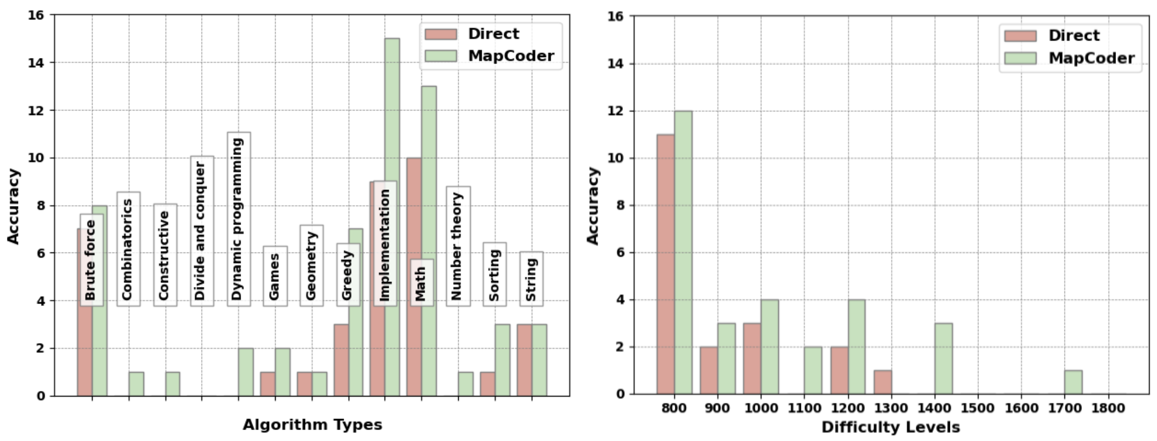


图5：关于算法标签和难度级别（xCodeEval数据集），正确率。

与这些数据集上的强大基线相比，无论它们是否基于提示，都表现出色。值得注意的是，在CodeContest上，我们的Pass@1结果与同期最先进的模型AlphaCodium（Ridnik等人，2024年）的Pass@5分数相当：28.5%与29%（见表3）。此外，我们的Pass@5结果还显示出额外的12.8%的改进。在APPS上，MapCoder始终优于ChatGPT和GPT-4模型的所有基线提示的Pass@1分数。

CodeContest (Pass@5)		
Approach	ChatGPT	GPT4
Direct	11.2%	18.8%
AlphaCodium	17.0%	29.0%
MapCoder	**18.2% (↑ 63.1%)**	**35.2% (↑ 87.1%)**

表3：CodeContest数据集上的Pass@5结果。Alph-Codium结果来自Ridnik等人（2024）。绿色单元格表示SOTA，红色文本表示相对于直接方法的改进。

5.3 不同难度级别上的性能表现

APPS数据集包含分为三个难度级别的题目：（i）入门，（ii）面试，和（iii）竞赛。图6展示了针对这三个类别的各种竞争方法的性能。结果表明，我们的MapCoder在所有问题类别中表现卓越，在竞争性问题解决中的最高增益表明其在一般代码生成能力上的优越性，此外，在竞争性问题解决方面表现出显著的有效性。为了更深入地了解它能解决哪些算法问题以及实际上它能解决多少难度级别的题目，我们还进行了MapCoder与直接方法之间的比较，考虑到xCodeEval数据集中存在的难度级别[1]和标签[2]。这一比较显示，MapCoder在多种算法类型中均表现出有效性，并且与直接方法相比，即使在更高的难度级别上，其性能也展现出更优越的性能。然而，在（中等难度：难度>1000）以上，其提升仍然有限。

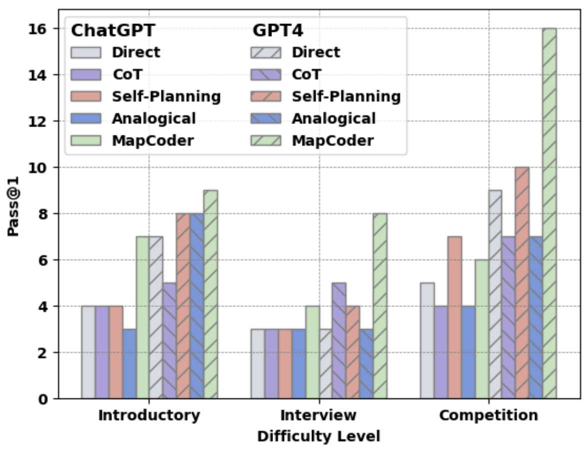


图6：性能与问题类型（APPS）对比。

5.4 不同LLM的性能表现

为了展示MapCoder在各种LLM上的鲁棒性，我们在表4中使用Gemini Pro评估MapCoder，它是与最先进LLM（SoTA）不同家族的LLM。我们还在表5中使用开源LLM Mistral-7B instruct评估MapCoder。正如预期的那样，我们的方法在简单问题（HumanEval）和竞赛级问题（CodeContest）上，与其它基线方法相比，在性能上有所提升，且趋势一致。　　我们的方法在简单问题（HumanEval）和竞赛级问题（Code Contest）上，与其它基线方法相比，在性能上有所提升，且趋势一致。

[1]Difficulty levels in xCodeEval dataset represents an integer number, a higher value means more difficult problem

[2]Tags in xCodeEval dataset represents algorithm type that can be used to solve the problem i.e., greedy, dp, brute-force, constructive, and so on.

LLM	Approach	HumanEval	CodeContest
Gemini	Direct	64.6%	3.6%
	CoT	66.5%	**4.8%**
	MapCoder	**69.5% (↑ 7.5%)**	**4.8% (↑ 32.0%)**

Table 4: Pass@1 results with using Gemini Pro. The red text is gain over Direct Prompting approach.

LLM	Approach	HumanEval	HumanEval-ET
Mistral	Direct	27.3%	27.3%
	CoT	45.5%%	42.4%
	MapCoder	**57.6% (↑ 111.1%)**	**48.5% (↑ 77.8%)**

Table 5: Pass@1 results with using Mistral-7B-instruct. The red text is gain over Direct Prompting approach.

5.5 Performance Across Different Programming Languages

Furthermore, we evaluate model performances using MapCoder across different programming languages. We utilize the xCodeEval dataset, which features multiple languages. Figure 7 shows that consistent proficiency across different programming languages is achieved by MapCoder with respect to baselines.

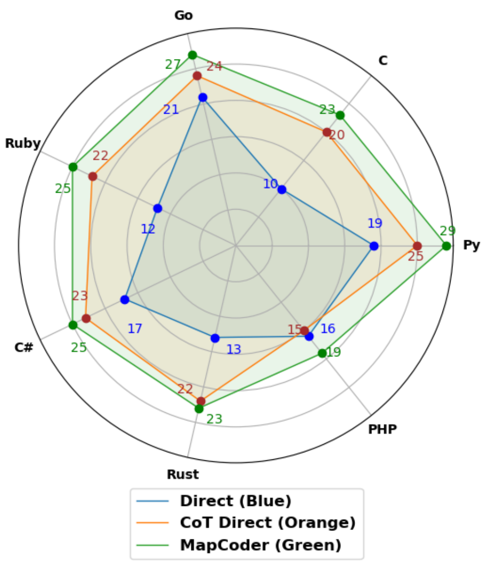


Figure 7: The number of correct answers wrt different programming languages (xCodeEval dataset).

6 Ablations Studies and Analyses

We present the ablation study of the MapCoder on HumanEval dataset as the problems are simpler and easy to diagnose by us humans.

6.1 Impact of Different Agents

We have also conducted a study by excluding certain agents from our MapCoder, which helps us investigate each agent's impact in our whole pipeline. As expected, the results (Table 6) show that every agent has its role in the pipeline as turning off any agent decreases the performance of MapCoder. Furthermore, we observe that the Debugging Agent has the most significant impact on the pipeline, as evidenced by a performance drop of 17.5% when excluding this agent exclusively, and an avg performance drop of 24.83% in all cases. The *Planning agent* has the second best important with avg drop of 16.7% in all cases. In Table 6), we perform an ablation study of our multi-agent framework investigate each agent's impact in our whole pipeline.

Retrieval Agent	Planning Agent	Debugging Agent	Pass@1	Performance Drop
✗	✗	✓	68.0%	15.0%
✗	✓	✓	76.0%	5.0%
✗	✓	✗	52.0%	35.0%
✓	✗	✓	70.0%	12.5%
✓	✓	✗	66.0%	17.5%
✓	✗	✗	62.0%	22.5%
✓	✓	✓	80.0%	-

Table 6: Pass@1 results for different versions of MapCoder (by using ChatGPT on HumanEval dataset).

6.2 Qualitative Example

To verify the above numerical significance, and to understand how our method enhance the code generation, we have performed a qualitative analysis to find the underlying reason for the superior performance of MapCoder over other competitive prompting approaches. An example problem and the output with the explanation of Direct, CoT, Reflexion, and MapCoder prompting is shown in Figure 4. This example demonstrates how the *Debugging Agent* fixes the bugs leveraging the plan as a guide from the *Planning Agent*. This verifies the impact of these two most significant agents. We present more detailed examples in Appendix.

6.3 Impact of k and t

MapCoder involves two hyper-parameters: the number of self-retrieved exemplars, k, and the number of debugging attempts, t. Our findings (Table 7) reveal that higher k, t is proportionate performance gain at the expense of time.

Dataset Name	k \ t	0	3	5
HumanEval	3	62.8%	76.8%	**80.5%**
	5	65.9%	79.9%	80.5%
HumanEval-ET	3	57.3%	61.0%	**70.1%**
	5	57.9%	67.1%	67.1%

Table 7: Pass@1 results by varying k and t.

LLM	Approach	HumanEval	CodeContest
Gemini	Direct	64.6%	3.6%
	CoT	66.5%	**4.8%**
	MapCoder	**69.5% (↑ 7.5%)**	**4.8% (↑ 32.0%)**

表4：使用Gemini Pro的Pass@1结果。红色文字表示相对于Direct Prompting方法的增益，。

LLM	Approach	HumanEval	HumanEval-ET
Mistral	Direct	27.3%	27.3%
	CoT	45.5%%	42.4%
	MapCoder	**57.6% (↑ 111.1%)**	**48.5% (↑ 77.8%)**

表5：采用Mistral-7B-instruct模型后的Pass@1指标结果。红色文字表示相对于Direct Prompting方法的增益。

5.5 在不同编程语言中的性能指标

此外，我们使用MapCoder在不同编程语言中评估模型性能。我们利用包含多种语言的xCodeEval数据集。图7显示，MapCoder与基准测试相比，实现了在不同编程语言中的持续性能。

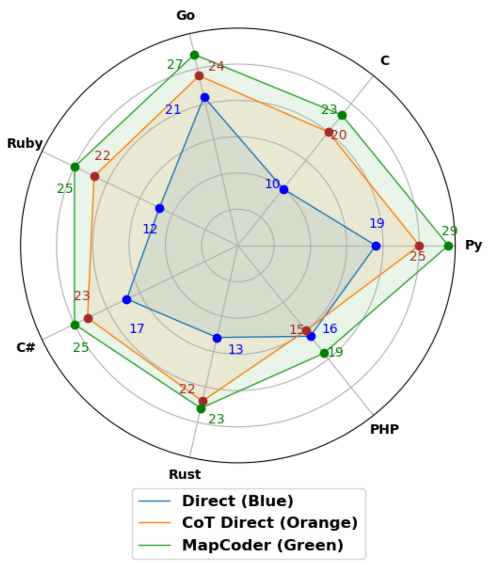


图7：关于不同编程语言的正确答案数量（xCodeEval数据集）。

6 消融术研究与分析

我们展示了MapCoder在HumanEval数据集上的消融实验，因为问题较为简单，易于我们人类诊断。

6.1 不同药物剂型的影响

我们还在MapCoder中排除了某些代理，通过这项研究，我们能够分析每个代理在整个流程中的影响。

如预期，结果（表6）显示每个代理在管道中都有其作用，因为关闭任何代理都会降低MapCoder的性能。此外，我们观察到调试代理对管道的影响最为显著，如排除此代理时性能下降17.5%，在所有情况下平均性能下降24.83%。其中 *Planning* 和 *agent* 在所有情况下平均下降16.7%，是第二重要的。如表6所示，我们对我们多代理框架进行了消融研究，以研究每个代理在我们整个管道中的影响。

Retrieval Agent	Planning Agent	Debugging Agent	Pass@1	Performance Drop
✗	✗	✔	68.0%	15.0%
✗	✔	✔	76.0%	5.0%
✗	✔	✗	52.0%	35.0%
✔	✗	✔	70.0%	12.5%
✔	✔	✗	66.0%	17.5%
✔	✗	✗	62.0%	22.5%
✔	✔	✔	80.0%	-

表6：不同版本的MapCoder（在HumanEval数据集上使用ChatGPT,）Pass@1率。

6.2 定性示例。

为了验证上述数值的显著性，并了解我们的方法如何提升代码生成，我们进行了一项定性分析，以探究MapCoder优于其他提示方法的原因。图4展示了直接、CoT、Reflexion和MapCoder提示的示例问题及其输出，并附有说明。此示例展示了*De- bugging Agent*如何借助*Planning Agent*的计划修复bug。这验证了这两个关键因素的影响。我们在附录中提供了更详细的示例。

6.3 k 及 t 的影响

MapCoder涉及两个超参数：数量为k的自我检索示例，以及数量为t的调试尝试。研究发现（见表7）显示，更高的k，t性能提升与成正比，但以时间为代价。

Dataset Name	k \ t	0	3	5
HumanEval	3	62.8%	76.8%	**80.5%**
	5	65.9%	79.9%	80.5%
HumanEval-ET	3	57.3%	61.0%	**70.1%**
	5	57.9%	67.1%	67.1%

表7，通过调整k和t，一次通过率结果。

		Average for MapCoder		Average for Direct Prompting		Accuracy Enhancement
LLM	Dataset	API Calls	Tokens (k)	API Calls	Tokens (k)	
ChatGPT	HumanEval	17	10.41	1	0.26	67.3%
	MBPP	12	4.84	1	0.29	57.3%
	APPS	21	26.57	1	0.66	41.3%
	xCodeEval	19	24.10	1	0.64	52.6%
	CodeContest	23	34.95	1	0.80	132.8%
GPT4	HumanEval	15	12.75	1	0.43	17.2%
	MBPP	8	4.96	1	0.57	2.5%
	APPS	19	31.80	1	0.82	73.7%
	xCodeEval	14	23.45	1	0.85	41.2%
	CodeContest	19	38.70	1	1.11	135.1%
Average		16.7	21.25	1	0.64	62.1%

Table 8: Average number of API calls, thousands of tokens used, required time in minutes to get the API response.

6.4 Impact of Number of Sample I/Os

Given the limited number of sample I/Os in the HumanEval dataset (average of 2.82 per problem), we supplemented it with an additional 5 sample I/Os from the HumanEval-ET dataset. Experiments with this augmented set showed an 1.5% performance gain.

6.5 Error Analysis and Challenges

Although MapCoder demonstrates strong performance compared to other methods, it faces challenges in certain algorithmic domains. For example, Figure 5 illustrates MapCoder's reduced performance on more difficult problems requiring precise problem understanding and concrete planning—capabilities still lacking in LLMs. In the xCodeEval dataset (see Figure 5), it solves a limited number of problems in categories like Combinatorics, Constructive, Number Theory, Divide and Conquer, and Dynamic Programming (DP). Manual inspection of five DP category problems reveals occasional misinterpretation of problems, attempts to solve using greedy or brute-force approaches, and struggles with accurate DP table construction when recognizing the need for a DP solution.

7 Conclusion and Future Work

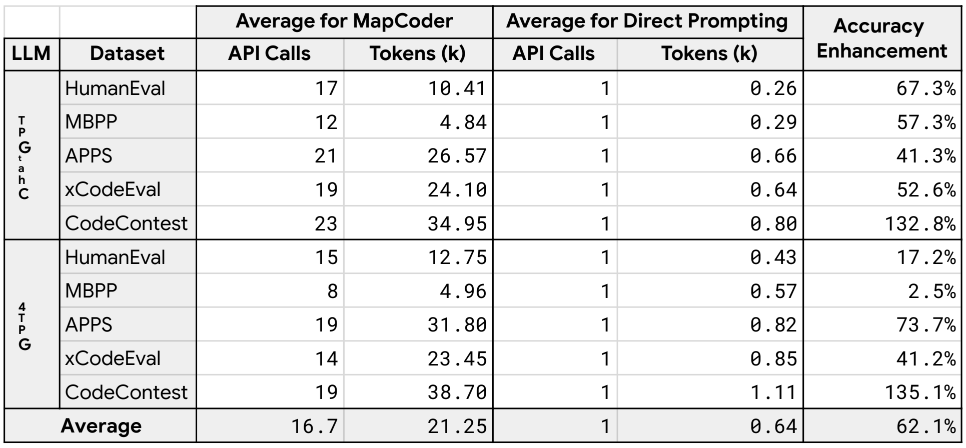
In this paper, we introduce MapCoder, a novel framework for effective code generation in complex problem-solving tasks, leveraging the multi-agent prompting capabilities of LLMs. MapCoder captures the complete problem-solving cycle by employing four agents - retrieval, planning, coding, and debugging - which dynamically interact to produce high-quality outputs. Evaluation across major benchmarks, including basic and competitive programming datasets, demonstrates MapCoder's consistent outperformance of well-established baselines and SoTA approaches across various metrics. Future work aims to extend this approach to other domains like question answering and mathematical reasoning, expanding its scope and impact.

8 Limitations

Among the limitations of our work, firstly, MapCoder generates a large number of tokens, which may pose challenges in resource-constrained environments. Table 8 shows the number of average API calls and token consumption with the default k and t (i.e., with respect to the reported performance) while Table 7) shows how k, t can be adjusted to proportionate the performance gain at the expense of time/token. We have not addressed the problem of minimizing tokens/API-calls in this paper and leave it for future works. Secondly, our method currently relies on sample input-output (I/O) pairs for bug fixing. Although sample I/Os provide valuable insights for LLMs' code generation, their limited number may not always capture the full spectrum of possible test cases. Consequently, enhancing the quality of additional test case generation could reduce our reliance on sample I/Os and further improve the robustness of our approach. Additionally, future exploration of open-source code generation models, such as CodeLLaMa, LLaMa3, Mixtral 8x7B could offer valuable insights and potential enhancements to our approach. Another important concern is that while running machine-generated code, it is advisable to run it inside a sandbox to avoid any potential risks.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training



		Average for MapCoder		Average for Direct Prompting		Accuracy Enhancement
LLM	Dataset	API Calls	Tokens (k)	API Calls	Tokens (k)	
ChatGPT	HumanEval	17	10.41	1	0.26	67.3%
	MBPP	12	4.84	1	0.29	57.3%
	APPS	21	26.57	1	0.66	41.3%
	xCodeEval	19	24.10	1	0.64	52.6%
	CodeContest	23	34.95	1	0.80	132.8%
GPT4	HumanEval	15	12.75	1	0.43	17.2%
	MBPP	8	4.96	1	0.57	2.5%
	APPS	19	31.80	1	0.82	73.7%
	xCodeEval	14	23.45	1	0.85	41.2%
	CodeContest	19	38.70	1	1.11	135.1%
Average		16.7	21.25	1	0.64	62.1%

表8：平均API调用次数，使用的token数量（千），所需时间（分钟）获取API响应。

6.4 样本输入/输出数量对影响

考虑到HumanEval数据集中样本输入输出数量有限（每个问题平均2.82个），我们从HumanEval-ET数据集中补充了额外的5个样本输入输出。使用这个增强集进行的实验显示性能提升了1.5%，性能提升了1.5%。

6.5 错误分析与挑战及难点

尽管与其它方法相比，MapCoder展现出强大的性能，但在某些算法领域它仍面临挑战。例如，图5展示了MapCoder在需要精确问题理解和具体规划的更难问题上的性能下降——这些能力在LLMs中仍显不足。在xCodeEval数据集中，它仅解决了组合数学、构造性、数论、分而治之以及动态规划（DP）等类别中有限数量的问题。对五个DP类别问题进行人工检查揭示，偶尔会误解问题，尝试使用贪婪或暴力方法解决问题，并在需要DP解决方案时，在准确构建DP表时遇到困难。

7 结论与未来工作

在这篇论文中，我们介绍了MapCoder，这是一个用于在复杂问题解决任务中有效生成代码的新颖框架，利用了LLMs的多智能体提示能力。MapCoder通过采用四个智能体——检索、规划、编码和调试——来捕捉完整的问题解决周期，这些智能体动态交互以产生高质量的输出。在包括基本和竞技编程数据集在内的主要基准测试中的评估表明，MapCoder的性能了。一致超越长期建立的基准和最先进技术的各项指标。未来工作旨在将此方法扩展到其他领域，例如问答和数学推理，扩大其应用范围和影响力。

8 局限性（如为文档或章节的一部分，请添加副标题或解

在我们的工作局限性中，首先，MapCoder生成大量标记，这可能在资源受限的环境中带来挑战。表8显示了默认v0和v1，即与报告的性能相关的平均API调用次数和标记消耗，而表7显示了如何调整v3、v4以牺牲时间/标记来成比例地提高性能。本文未解决最小化标记/API调用的难题，将其留待未来研究。其次，我们的方法目前依赖于样本输入输出（I/O）对进行错误修复。尽管样本I/O为LLMs的代码生成提供了有价值的见解，但它们的数量有限可能无法始终捕捉到所有可能的测试用例的全貌。因此，提高额外测试用例生成质量可以减少我们对样本I/O的依赖，并进一步提高我们方法的鲁棒性。此外，未来对开源代码生成模型（如CodeL-Lama、LLaMa3、Mixtral 8x7B）的探索可以提供有价值的见解和对我们方法的潜在改进。另一个重要问题是，建议在运行机器生成的代码时，在沙盒中运行以避免潜在风险。

参考文献

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray 和 Kai-Wei Chang. 2021年，统一预训练

for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy McGovern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. Task-oriented dialogue as dataflow synthesis. *Transactions of the Association for Computational Linguistics*, 8:556–571.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*.

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

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. 2021b. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Yihong Dong, Jiazheng Ding, Xue Jiang, Zhuo Li, Ge Li, and Zhi Jin. 2023a. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023b. Self-collaboration code generation via chatgpt.

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

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices*, 46(1):317–330.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 763–773, New York, NY, USA. ACM.

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM*, 59(5):122–131.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. Mistral 7b.

促进程序理解和生成。*arXiv*，*preprint arXiv:2103.06333*。

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, 等. 2023. Santacoder：不要伸手摘星！版本2。

Jacob Andreas, John Bufe, David Burkett, Charles Chen, Josh Clausman, Jean Crawford, Kate Crim, Jordan DeLoach, Leah Dorner, Jason Eisner, Hao Fang, Alan Guo, David Hall, Kristin Hayes, Kellie Hill, Diana Ho, Wendy Iwaszuk, Smriti Jha, Dan Klein, Jayant Krishnamurthy, Theo Lanman, Percy Liang, Christopher H. Lin, Ilya Lintsbakh, Andy Mc-Govern, Aleksandr Nisnevich, Adam Pauls, Dmitrij Petters, Brent Read, Dan Roth, Subhro Roy, Jesse Rusak, Beth Short, Div Slomin, Ben Snyder, Stephon Striplin, Yu Su, Zachary Tellman, Sam Thomson, Andrei Vorobev, Izabela Witoszko, Jason Wolfe, Abby Wray, Yuchen Zhang, and Alexander Zotov. 2020. 基于数据流合成的任务导向对话。*Trans-actions of the Association for Computational Linguis-tics*，第8卷，第556-571页。

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le， 等。2021. 基于大型语言模型的程序综合。*arXiv preprint arXiv:2108.07732*。

陈贝，张峰吉，阮安，赞道光，林泽琪，楼建光，陈伟竹。2022. Codet：基于生成测试的代码生成。*arXiv preprint arXiv:2207.10397*。

陈马克，杰瑞·特沃雷克，金孝宇，袁奇明，奥利维拉·平托，贾里德·卡普兰，哈里·爱德华兹，尤里·伯达，尼古拉斯·约瑟夫，格雷格·布鲁克曼，亚历克斯·雷，劳尔·普里，格雷琴·克鲁格，迈克尔·彼得罗夫，海迪·卡拉夫，吉里什·萨斯特里，帕梅拉·米什金，布鲁克·陈，斯科特·格雷，尼克·莱德，米哈伊尔·帕夫洛夫，阿莱西娅·鲍威尔，卢卡斯·凯撒，穆罕默德·巴维里安，克莱门斯·温特，菲利普·蒂莱特，费利佩·佩特罗斯基·苏查，戴夫·坎明斯，马蒂亚斯·普拉珀特，福蒂奥斯·查恩齐斯，伊丽莎白·巴恩斯，艾瑞尔·赫伯特-沃斯，威廉·赫本·古斯，亚历克斯·尼科尔，亚历克斯·帕伊诺，尼古拉斯·特扎克，杰杰·唐，伊戈尔·巴布什金，苏奇尔·巴拉吉，沙南图·贾因，威廉·桑德斯，克里斯托弗·赫斯，安德鲁·N·卡尔，简·莱克，乔什·阿奇亚姆，维达恩特·米斯拉，伊万·莫里卡瓦，亚历克·拉德福德，马修·奈特，迈尔斯·布兰达奇，米拉·穆拉蒂，凯蒂·梅耶，彼得·韦林德，鲍勃·麦格鲁，达里奥·阿莫迪，山姆·麦克安德利斯，伊利亚·苏茨克维尔，沃伊切赫·扎伦巴。2021a. 对基于代码训练的大型语言模型进行的评估。

陈马克，杰瑞·特沃雷克，金孝宇，袁奇明，奥利维拉·平托，哈恩里克·庞德，贾里德·卡普兰，哈里·爱德华兹，尤里·伯达，尼古拉斯·约瑟夫，格雷格·布罗克曼，等。2021b。评估基于代码训练的大型语言模型。（版本10）（版本11）。

陈欣云，林马克斯韦尔，纳塔纳埃尔·沙尔利，周登尼。2023。教大型语言模型自我调试。第12卷。

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann， 等. 2022. Palm：通过路径技术扩展语言模型。{第13卷} {第14卷}。

董一红，丁家珍，蒋雪，李卓，李鸽，金志。2023a. CodeScore：通过学习代码执行来评估代码生成，*arXivpreprint arXiv:2301.09043*。

东易红，薛江，金志，李格。2023b. 通过ChatGPT，进行自我协作代码生成。

张银凤，郭大亚，唐杜宇，端楠，冯西澳城，龚明，寿林军，秦冰，刘婷，姜大新，等。2020。Codebert：编程和自然语言预训练模型。在 {v*} {v*}，第1536-1547页。

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, 及 Mike Lewis. 2022. Incoder：该生成模型用于代码补全和合成。*arXiv preprint arXiv:2204.05999*。

Sumit Gulwani. 2011. 通过输入输出示例在电子表格里自动化字符串处理。*ACM Sigplan Notices*，46 (1)：317-330。

郭垚、朱启豪、杨德健、谢振达、董凯、张文涛、陈冠廷、毕晓、吴Y、李YK等。2024. Deepseek-coder（深度探索编码器）：大型语言模型与编程相遇——代码智能的兴起。

Vincent J. Hellendoorn 和 Premkumar Devanbu. 2017, 深度神经网络是建模源代码的最佳选择吗？在 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineer-ing*，ESEC/FSE 2017，第763-773页，纽约，美国。ACM（美国计算机协会）

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, 和 Premkumar Devanbu. 2016. 软件的自然性。*Commun. ACM*, 第59卷 第5期:122–131。

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck 和 Heming Cui. 2023. Agentcoder：基于多智能体的代码生成，结合迭代测试和优化。第27卷。

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guil- laume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, 和 William El Sayed. 2023a. Mistral 7b.

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

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

Donald E Knuth. 1992. Literate programming. *CSLI Lecture Notes, Stanford, CA: Center for the Study of Language and Information (CSLI), 1992*.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328.

Jingyao Li, Pengguang Chen, and Jiaya Jia. 2023. Motcoder: Elevating large language models with modular of thought for challenging programming tasks. *arXiv preprint arXiv:2312.15960*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022a. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022b. Competition-level code generation with alphacode.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zohar Manna and Richard J. Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM*, 14(3):151–165.

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

Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 75–84. IEEE.

Emilio Parisotto and Ruslan Salakhutdinov. 2017. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2018. Building language models for text with named entities. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2373–2383, Melbourne, Australia. Association for Computational Linguistics.

Md Rizwan Parvez, Jianfeng Chi, Wasi Uddin Ahmad, Yuan Tian, and Kai-Wei Chang. 2023. Retrieval enhanced data augmentation for question answering on privacy policies. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 201–210, Dubrovnik, Croatia. Association for Computational Linguistics.

Oleksandr Polozov and Sumit Gulwani. 2015. Flashmeta: A framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *CoRR*, abs/1704.07535.

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

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

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Kashun Shum, Shizhe Diao, and Tong Zhang. 2023. Automatic prompt augmentation and selection with chain-of-thought from labeled data. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 12113–12139, Singapore. Association for Computational Linguistics.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

薛江，董一鸿，王乐成，商奇伟，李戈。2023b. 基于大型语言模型的自我规划代码生成。
arXiv preprint arXiv:2303.06689。

Mohammad Abdullah Matin Khan, Saiful Bari M, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, 和 Shafiq Joty. 2023. xcodeeval：一个大规模多语言多任务基准测试，用于实现代码理解、生成、翻译与检索。*arXiv preprintarXiv:2303.03004*。

唐纳德·E·克努特。1992年。《文学化编程》。第4版/第5版/第6版。

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, 和 Steven Chu Hong Hoi. 2022. Coderl: 通过预训练模型, 和深度强化学习技术掌握代码生成。第7/8卷，35: 21314–21328。

李静瑶，陈鹏光，贾佳亚。2023. Mot-coder：通过模块化思维提升大型语言模型以应对具有挑战性的编程任务。*arXivpreprint arXiv:2312.15960*。

李宇佳，大卫·崔，郑俊勇，内特·库什曼，朱利安·施里特维瑟，雷米·勒布隆，汤姆·埃克莱斯，詹姆斯·基林，费利克斯·吉门诺，奥古斯丁·达·拉戈，等。2022a. 发表在《期刊名》*Science*，378(6624)：1092–1097。

李宇佳，大卫·崔，郑俊勇，内特·库什曼，朱利安·施里特维瑟，雷米·勒布隆，汤姆·埃克莱斯，詹姆斯·基林，费利克斯·吉门诺，奥古斯丁·达·拉戈，托马斯·休伯特，彼得·卓宜，西普里恩·德·马斯松·德·奥图姆，伊戈尔·巴布什金，陈欣云，黄波森，约翰内斯·韦尔布，斯文·高瓦尔，阿列克谢·切列帕诺夫，詹姆斯·莫洛伊，丹尼尔·曼科夫茨，艾斯梅·萨瑟兰·罗布森，普什梅特·科利，南多·德·弗雷塔斯，科拉伊·卡夫库库奥卢，奥里奥尔·维尼亚尔斯。2022b. 使用alphacode进行竞争级代码生成。

李华伟，夏春秋，王宇瑶，张凌明。2023。你的代码真的是由Chat-GPT生成的吗？对用于代码生成的大型语言模型的严格评估。在*Thirty-seventh*, *Conference on Neural Information Processing Sys-*, *tems*。

Zohar Manna 和 Richard J. Waldinger. 1971. 朝向自动程序合成。*Commun. ACM*, 第14卷 第3期:151–165。

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, Caiming Xiong. 2022. Codegen：一个开源的大型语言模型，用于代码的多轮程序合成（v16）。

卡洛斯·帕切科，舒文杜·K拉里，迈克尔·D·恩斯特，托马斯·鲍尔。2007。基于反馈的随机测试生成。在《IEEE Transactions on Software Engineering》第17卷第18期，第75-84页。IEEE,。

埃米利奥·帕里索托，鲁斯兰·萨拉克图丁诺夫。2017年。神经映射：用于深度强化学习的结构化记忆。*arXiv preprint arXiv:1702.08360*。

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray，Kai-Wei Chang. 2021，基于检索的增强代码生成与摘要。{第20卷}。

Md Rizwan Parvez, Saikat Chakraborty, Baishakhi Ray 和 Kai-Wei Chang. 2018. 为包含命名实体的文本构建语言模型。在 *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*，第2373-2383页，墨尔本，澳大利亚。计算语言学协会。

Md Rizwan Parvez, Jianfeng Chi, Wasi Uddin Ahmad, Yuan Tian, 和 Kai-Wei Chang. 2023年, 隐私政策问答中的检索增强数据增强。在 第24卷, 第25卷, 第26卷，第201-210页，杜布罗夫尼克，克罗地亚。计算语言学协会（ACL）。

Oleksandr Polozov 和 Sumit Gulwani. 2015, Flash-Meta：用于归纳程序综合的框架（一种方法）。发表在第27卷、第28卷、第29卷第107–126页。

Maxim Rabinovich, Mitchell Stern，和 Dan Klein. 2017. 关于代码生成和语义解析的抽象语法网络。*CoRR*, arXiv:1704.07535。

Tal Ridnik, Dedy Kredo 和 Itamar Friedman. 2024. 使用 Alphacodium 进行代码生成：从提示工程到流程工程。提示工程与流程工程是两种重要的编程方法，分别涉及如何设计输入提示和优化代码执行流程。*arXiv preprint arXiv:2401.08500*。

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, 等. 2023, Code llama：开源代码基础模型（LLaMA）*arXiv preprint arXiv:2308.12950*。

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R. Narasimhan 和 Shunyu Yao. 2023. 反思：具有语言强化学习的语言代理。在第 35 卷 36 期。

Kashun Shum, Shizhe Diao 和 Tong Zhang. 2023. 基于标签数据的思维链自动提示增强与选择。在 *Findings of the Association for Computational Linguistics: EMNLP 2023*，第 12113–12139 页，新加坡，计算语言学协会。

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale，等. 2023. Llama 2：开源平台和微调聊天机器人。版本40 版本41。

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *EMNLP*, pages 8696–8708.

Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022a. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022b. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Xiaohan Xu, Chongyang Tao, Tao Shen, Can Xu, Hongbo Xu, Guodong Long, and Jian guang Lou. 2023. Re-reading improves reasoning in language models.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*.

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, and Denny Zhou. 2023. Large language models as analogical reasoners. *arXiv preprint arXiv:2310.01714*.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Yifan Zhang, Jingqin Yang, Yang Yuan, and Andrew Chi-Chih Yao. 2023. Cumulative reasoning with large language models. *arXiv preprint arXiv:2308.04371*.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493*.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2023. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*.

Appendix

A Algorithm of `MapCoder`

Algorithm 1 shows the pseudo-code of our prompting technique.

Algorithm 1 `MapCoder`

```
1:  $k \leftarrow$ number of self-retrieved exemplars
2:  $t \leftarrow$ number of debugging attempts
3:
4:  $exemplars \leftarrow$ RetrivalAgent($k$)
5:
6:  $plans \leftarrow$ empty array of size $k$
7:  for $example$ in $exemplars$ do
8:      $plans[i] \leftarrow$ PlanningAgent($example$)
9:  end for
10:
11: $plans \leftarrow$ SortByConfidence($plans$)
12:
13: for $i \leftarrow 1$ to $k$ do
14:     $code \leftarrow$ CodingAgent($code$, $plan[i]$)
15:     $passed, log \leftarrow$ test($code$, $sample\_io$)
16:     if $passed$ then
17:         Return $code$
18:     else
19:         for $j \leftarrow 1$ to $t$ do
20:             $code \leftarrow$ DebuggingAgent($code$, $log$)
21:             $passed, log \leftarrow$ test($code$, $sample\_io$)
22:             if $passed$ then
23:                 Return $code$
24:             end if
25:         end for
26:     end if
27: end for
28: Return $code$
```

B Details Promptings of `MapCoder`

The detailed prompting of the Retrieval Agent, Planning Agent, Coding Agent, and Debugging Agent are shown in Figure 8, 9, and 10 respectively. Note that we adopt a specific sequence of instructions in the prompt for Retrieval Agent which is a crucial design choice.

C Example Problem

Two complete examples of how `MapCoder` works by showing all the prompts and responses for all four agents is given in this link.

王越，王威仕，沙菲克·乔蒂，和史蒂文·C·H霍伊。2021. Codet5：用于代码理解和生成的标识符感知统一预训练编码器-解码器模型。载于 *EMNLP*，第 8696–8708 页。

Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, 和 Graham Neubig. 2023. 学习过滤上下文以增强检索增强生成。*arXiv preprint arXiv:2311.08377*。

魏 Jason，王雪志 Xuezhi，戴尔·舒尔曼斯 Dale Schuurmans，马滕·博斯马 Maarten Bosma，夏斐 Fei，Ed Chi，Quoc V Le，周登尼 Denny，等人。2022a。思维链提示引发大型语言模型的推理。第35卷，第24824-24837页。

魏杰森，王雪芝，戴尔·舒尔曼斯，马滕·博斯玛，夏斐，艾德·奇，吴国伟·莱，周登尼，等。2022b。思维链提示引发大型语言模型的推理。第5卷 第6期，第35卷 第24824-24837页。

许夏焕，陶长阳，沈涛，徐灿，徐宏博，龙国栋，楼建光。2023. 重读提升语言模型中的推理能力。

思维之树：通过大型语言模型进行精心的问题解决方法。第7卷，第8卷。

Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H Chi, 和 Denny Zhou. 2023. 大型语言模型作为类比推理者。（第9卷）。

尹彭程和格雷厄姆·纽比格。2017。通用代码生成的句法神经网络模型。Vol. 10，arXiv:1704.01696。

陶宇，张瑞，二羊，李思怡，薛艾瑞克，潘博，林西维多利亚，陈毅然，石天泽，李紫涵，姜有轩，山根道弘，沈圣洛，陈涛，亚历山大·法布里，李子凡，陈露瑶，张宇文，迪希塔·谢雷亚，张文森，熊才明，索cher理查德，拉斯基沃尔特，拉代米尔·拉代夫。2019。CoSQL：面向跨领域自然语言界面数据库的会话式文本到SQL挑战。在 *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*，第1962–1979页，中国香港。计算语言学协会（ACL）。

张一帆，杨静琴，袁洋，以及姚安哲。2023. 基于大型语言模型的累积推理。{第15卷} {第16卷}。

张卓胜，张阿斯顿，李穆，及亚历克斯·斯莫拉。2022。在大语言模型中的自动思维链提示。*arXiv preprint arXiv:2210.03493*。

Andy Zhou，Kai Yan，Michal Shlapentokh-Rothman，Haohan Wang，和 Yu-Xiong Wang。2023。语言代理树搜索统一了语言模型中的推理、行动与规划。*arXiv preprint arXiv:2310.04406*。

附录

MapCoder 算法

算法1展示了我们的提示方法的伪代码。

Algorithm 1 MapCoder

```
1:  $k \leftarrow$ number of self-retrieved exemplars
2:  $t \leftarrow$ number of debugging attempts
3:
4:  $exemplars \leftarrow$ RetrivalAgent($k$)
5:
6:  $plans \leftarrow$ empty array of size $k$
7:  for $example$ in $exemplars$ do
8:      $plans[i] \leftarrow$ PlanningAgent($example$)
9:  end for
10:
11: $plans \leftarrow$ SortByConfidence($plans$)
12:
13: for $i \leftarrow 1$ to $k$ do
14:     $code \leftarrow$ CodingAgent($code, plan[i]$)
15:     $passed, log \leftarrow$ test($code, sample\_io$)
16:     if $passed$ then
17:         Return $code$
18:     else
19:         for $j \leftarrow 1$ to $t$ do
20:             $code \leftarrow$ DebuggingAgent($code, log$)
21:             $passed, log \leftarrow$ test($code, sample\_io$)
22:             if $passed$ then
23:                 Return $code$
24:             end if
25:         end for
26:     end if
27: end for
28: Return $code$
```

B 地图编码器提示

检索代理、规划代理、编码代理和调试代理的详细提示分别显示在图8、9和10中。请注意，我们在检索代理的提示中采用了一个特定的指令顺序，这是一个关键的设计选择。

C语言示例问题

两个完整的示例展示了MapCoder的工作方式，通过显示所有四个代理的所有提示和响应，可在此链接中查看。请点击链接。

Retrieval Agent

```
Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

# Problem:
{Problem Description will be added here}

# Exemplars:
Recall k relevant and distinct problems (different from problem mentioned above). For each problem,
1. describe it
2. generate {language} code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:
----------------
Important:
Your response must follow the following xml format-
<root>
  <problem>
    # Recall k relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.
    <description> # Describe the problem. </description>
    <code> # Let's think step by step to solve this problem in {language} programming language. </code>
    <planning> # Planning to solve this problem. </planning>
  </problem>
  # similarly add more problems here...
  <algorithm>
    # Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking, Recursive, Binary search,
    and so on) that needs to be used to solve the original problem.
    # Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for solving this types
    of problem. Do not generate code.
  </algorithm>
</root>
```

Figure 8: Prompt for self-retrieval Agent.

```
Planning Generation Prompt:
Given a competitive programming problem
generate a concrete planning to solve the
problem.
# Problem: {Description of the example problem}
# Planning: {Planning of the example problem}
## Relevant Algorithm to solve the next
problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample IOs}

----------------
Important: You should give only the planning to
solve the problem. Do not add extra explanation
or words.
```

```
Confidence Generation Prompt:
Given a competitive programming problem and a plan to solve
the problem in {language} tell whether the plan is correct to
solve this problem.

# Problem:  {Original Problem}
# Planning: {Planning of our problem from previous step}

----------------
Important: Your response must follow the following xml format-
<root>
 <explanation> Discuss whether the given competitive
 programming problem is solvable by using the above mentioned
 planning. </explanation>
 <confidence> Confidence score regarding the solvability of
 the problem. Must be an integer between 0 and 100.
 </confidence>
</root>
```

Figure 9: Prompt for Planning Agent. The example problems that are mentioned in this figure will come from the Retrieval Agent.

```
Given a competitive programming problem generate
Python3 code to solve the problem.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved:
{Our Problem Description will be added here}
## Planning: {Planning from the Planning Agent}
## Sample Input/Outputs: {Sample I/Os}
## Let's think step by step.
------------
Important:
## Your response must contain only the {language} code
to solve this problem. Do not add extra explanation or
words.
```

```
Given a competitive programming problem you have generated {language}
code to solve the problem. But the generated code cannot pass sample
test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving
this problem.

----------------
Important:
## Your response must contain the modified planning and then the
{language} code inside ``` block to solve this problem.
```

Figure 10: Prompt for Coding and Debugging Agent.

Retrieval Agent

```
Given a problem, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

# Problem:
{Problem Description will be added here}

# Exemplars:
Recall k relevant and distinct problems (different from problem mentioned above). For each problem,
1. describe it
2. generate {language} code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:
----------------
Important:
Your response must follow the following xml format-
<root>
  <problem>
    # Recall k relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.
    <description> # Describe the problem. </description>
    <code> # Let's think step by step to solve this problem in {language} programming language. </code>
    <planning> # Planning to solve this problem. </planning>
  </problem>
  # similarly add more problems here...
  <algorithm>
    # Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy, Backtracking, Recursive, Binary search,
    and so on) that needs to be used to solve the original problem.
    # Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial for solving this types
    of problem. Do not generate code.
  </algorithm>
</root>
```

图8：自检索界面。

```
Planning Generation Prompt:
Given a competitive programming problem
generate a concrete planning to solve the
problem.
# Problem: {Description of the example problem}
# Planning: {Planning of the example problem}
## Relevant Algorithm to solve the next
problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved: {Original Problem}
## Sample Input/Outputs: {Sample IOs}

----------------
Important: You should give only the planning to
solve the problem. Do not add extra explanation
or words.
```

```
Confidence Generation Prompt:
Given a competitive programming problem and a plan to solve
the problem in {language} tell whether the plan is correct to
solve this problem.

# Problem:  {Original Problem}
# Planning: {Planning of our problem from previous step}

----------------
Important: Your response must follow the following xml format-
<root>
 <explanation> Discuss whether the given competitive
 programming problem is solvable by using the above mentioned
 planning. </explanation>
 <confidence> Confidence score regarding the solvability of
 the problem. Must be an integer between 0 and 100.
 </confidence>
</root>
```

图 9：规划代理提示。图中提到的示例问题将来自检索代理。

```
Given a competitive programming problem generate
Python3 code to solve the problem.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Problem to be solved:
{Our Problem Description will be added here}
## Planning: {Planning from the Planning Agent}
## Sample Input/Outputs: {Sample I/Os}
## Let's think step by step.
------------
Important:
## Your response must contain only the {language} code
to solve this problem. Do not add extra explanation or
words.
```

```
Given a competitive programming problem you have generated {language}
code to solve the problem. But the generated code cannot pass sample
test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem:
{Algorithm retrieved by Retrieval Agent}
## Planning: {Planning from previous step}
## Code: {Generated code from previous step}
## Modified Planning:
## Let's think step by step to modify {language} Code for solving
this problem.

----------------
Important:
## Your response must contain the modified planning and then the
{language} code inside ``` block to solve this problem.
```

图10：编码调试代理提示。