

结构化思维链 Structured CoT：让大模型使用程序结构思考如何写代码


任何简单或复杂的算法都可以由顺序结构、选择结构和循环结构这三种基本结构组合而成。

一、摘要

大型语言模型（下文称为：大模型）在代码生成上表现出了强大的能力。大模型依赖于 prompt 作为输入，思维链是目前用于设计 prompt 的主流方法，在代码生成上取得了目前最好的准确率。但大模型的准确率依旧较低，无法用于实际生产环境。

北京大学李戈、金芝教授团队提出了一种结构化的思维链，显著地提升了大模型在代码生成上的准确率。结构化的思维链约束大模型使用程序结构（例如：顺序、分支和循环结构）去组织思维过程，引导大模型从程序语言的角度去思考如何解决需求。实验结果表明：结构化的思维链稳定地超越了之前的工作（例如：标准的思维链），进一步提升了大模型在代码生成上的性能。

Structured Chain-of-Thought Prompting for Code Generation

Jia Li 
lijia@stu.pku.edu.cn
Peking University
Beijing, China

Yongmin Li
Peking University
Beijing, China
liyongmin@pku.edu.cn

Ge Li
Peking University
Beijing, China
lige@pku.edu.cn

Zhi Jin
Peking University
Beijing, China
zhijin@pku.edu.cn

论文链接：<https://arxiv.org/pdf/2305.06599.pdf>

二、论文概述

大型语言模型（下文称为：大模型）在代码生成上表现出了强大的能力。用户的输入是一条 prompt，其中包括若干个演示样例（需求 - 代码）和一个新的需求。大模型基于 prompt 自动地为新需求生成源代码。

现有研究发现：prompt 的设计对于大模型的性能影响较大。因此，如何设计有效的 prompt 来提升大模型在代码生成上的准确率是软件工程和人工智能领域的一个研究热点。

Chain-of-Thought Prompting（下文称：CoT prompting）是一种用于设计 prompt 的新兴方法，在代码生成上取得了目前最好的准确率。针对一个需求，CoT prompting 先引导大模型思考如何解决需求，生成一段思维链（下文称：CoT）。CoT 指的是一连串的中间推理步骤，描述了如何一步一步地撰写代码。图 1 展示了在代码生成上 CoT 的示例。尽管 CoT prompting 在代码生成上取得了一定程度的提升，但它的准确率依旧较低，无法用于实际生产环境。

```
1. Initialize a result with -999999
2. Iterate through the list of lists
3. Initialize a sum with 0
4. Iterate through the list
5. Add the element to the sum
6. Update result with the maximum of sum and result
7. Divide the result by K
8. Return the result
```

(a) Chain-of-Thought

```
Input: array: list[list], K: int
Output: result: int or float
1: Initialize a result with -999999
2: for _list in the list of lists:
3:     Calculate the sum of the list
4:     if the sum is great than result:
5:         Update the result
6: Divide result by K
7: return result
```

Diagram illustrating the mapping of code lines to control structures:

- Line 2: `for _list in the list of lists:` is part of a **Loop Structure** (lines 2-5).
- Line 4: `if the sum is great than result:` is part of a **Branch Structure** (lines 4-5).
- Lines 6-7: `Divide result by K` and `return result` are part of a **Sequence Structure** (lines 6-7).

(b) Structured Chain-of-Thought

Figure 1: The comparison of a Chain-of-Thoughts (CoT) and our Structured Chain-of-Thought (SCoT).

今年 7 月，北京大学李戈、金芝教授团队（下文称为：研究者们）针对代码生成，提出一种结构化的思维链（Structured Chain-of-Thought，下文称为：SCoT）。

研究者的动机是：源代码具有较强的结构性，例如：独特的程序结构 - 顺序结构、分支结构和循环结构。直觉上来说，一种结构化的思维链（即中间推理步骤）有利于推导出结构化的源代码。

想象一名人类程序员 Allen 在解决一个需求（例如：求取一个列表中的最大值）时的思维过程：

1. 初始化一个变量 Result; 2. 使用循环结构遍历列表中的值; a. 使用分支结构对每个值进行判断, i. 如果它大于 Result, 则更新 Result ii....

显然, 这种基于程序结构的思维过程更贴近程序语言的解题逻辑, 因此有利于引导后续的编码实现。

受到上述分析的启发, 研究者们提出: 使用程序结构来组织思维过程, 得到结构化的思维链 - SCoT。

图 1 (b) 展示了一个 SCoT 的示例。相较于标准的 CoT, SCoT 具有两点不同:

(1) 它使用三种基础程序结构来组织中间推理步骤。

Bohm 和 Jacopini 在 1966 年指出: 任何简单或复杂的算法都可以由顺序结构、分支结构和循环结构这三种基本结构组合而成[1]。因此, 研究者们引入三种基础结构, 并约束大模型使用这三种基础结构去生成思维链。这要求大模型从程序语言的角度去思考如何解决需求, 并使用三种基础结构准确地表达思维过程。

例如在图 1 (b) 中, SCoT 清晰地展示了一个大致的解题流程。其中, 它使用一个循环结构准确地描述了第二行的遍历操作 (例如: 作用域、循环起止点), 并使用一个分支结构去描述不同情况下的处理方法。而在标准的 CoT 中, 第二行和第四行的遍历操作存在歧义, 例如: 作用域模糊。这会误导后续的生成过程, 导致生成错误的代码。

(2) 它包含输入输出结构。

每一个程序都包含输入输出结构, 它指明了程序的输入输出参数及其类型。例如: 图 1 (b) 中的: Input: array: list [list]; Output: result。

研究者们认为, 引入输入输出结构有助于大模型去分析需求和明确程序的出入口。同时, 一个明确的输入输出结构也有利于引导出后续解题的思维过程。

基于上述的 SCoT, 研究者们提出一种新的代码生成方法, 叫做: SCoT prompting。针对一个需求, 它利用大模型先生成一段 SCoT, 然后基于需求和 SCoT 生成相应的源代码。相比于 CoT prompting, SCoT prompting 显式地在思维链中引入程序结构, 以此来引导大模型从程序语言的角度来思考如何解决需求。这进一步释放了大模型在代码生成上的推理能力, 从而提升大模型的准确率。

研究者们将 SCoT prompting 应用至两个大模型 (Codex 和 ChatGPT), 并在三个代码生成数据

集上进行了验证。研究者使用单元测试用例来评估生成的代码的正确性，并计算 Pass@K。实验结果表明：

- 在三个数据集上，SCoT prompting 稳定地超越了目前最好的方法 - CoT prompting。例如，在 Pass@1 上，SCoT prompting 在三个数据集上分别获得了 13.79%、12.31% 和 6.63% 的相对提升；
- 人工评估表明：人类程序员更偏爱基于 SCoT prompting 生成的代码；
- SCoT prompting 在不同的大模型和编程语言上都具有稳定的效果；
- SCoT prompting 具有较强的鲁棒性，不依赖于具体的演示样例和写作风格。

总的来说，本文的贡献可总结为以下几点：

- 一种结构化的思维链 - SCoT，它使用程序结构去组织中间推理步骤；
- 一种新的基于大模型的代码生成方法 - SCoT prompting，它利用大模型先生成结构化的思维链，再生成源代码；
- 进行了大量的定性和定量实验，展示了结构化思维链的有效性。

三、结构化的思维链 - SCoT

标准的思维链（CoT）初始是为自然语言生成任务而设计，使用自然语言顺序地描述如何逐步地解决问题。在代码生成上，CoT 带来的提升有限，大模型的准确率仍旧较低。

在本文中，研究者们提出一种结构化的思维链（Structured CoT, SCoT）。SCoT 显式地引入程序结构去撰写思维链，引导大模型使用程序语言的逻辑去思考如何解决需求。图 2 展示了 SCoT 的一些样例。

```

Input: paren_string: str
Output: list_of_int: List[int]
1: Initialize list_of_int to an empty list
2: for each string in paren_string do
3:   Initialize depth to 0
4:   for each character in string do
5:     if character is '(' then
6:       depth += 1
7:     elif character is ')' then
8:       depth -= 1
9:   append depth to list_of_int
10: return list_of_int

```

(a)

```

Input: string: str, substring: str
Output: count: int
1: Initialize count to 0
2: while substring is not found in string do
3:   if string is empty then
4:     return 0
5:   increment count
6:   remove the first character of string
7: return count

```

(b)

Figure 2: Examples of SCoT in code generation.

现有研究表明：任何简单或复杂的算法都可以由顺序结构、分支结构和循环结构这三种基本结构组合而成。因此，研究者们使用这三种基本结构撰写思维链。三种基本结构的详情如下所示：

- 顺序结构：中间步骤被顺序地组织，所有的步骤位于相同的层级。
- 分支结构：它以一个条件（condition）作为起始，并基于条件的不同结果放置不同的中间步骤。在本文中，分支结构包含三种形式：if ..., if ... else, if ... elif ... else。
- 循环结构：重复地执行一系列中间步骤，直到某项条件不被满足。在本文中，循环结构包括两种形式：for 循环和 while 循环结构。

不同的程序结构可以被嵌套使用，这允许大模型自主地设计更复杂的 SCoT 去解决困难的需求。如图 2 所示，SCoT 灵活地使用各种程序结构去构建一个解题流程。

除了三种基本结构，研究者们还引入了输入输出结构，它包括输入输出参数及其类型。研究者们认为输入输出结构反映了程序的入口和出口。生成输入输出结构有助于澄清需求并引导后续的推理过程。

四、SCoT prompting

基于结构化的思维链（SCoT），研究者们面向代码生成提出一种新的 prompt 设计方法 - SCoT prompting。它引导大模型先生成一段 SCoT，然后再生成相应的源代码。

为了实现 SCoT prompting，研究者们设计了两种特殊的 prompts。第一个 prompt 用于引导大模型基于需求生成一段 SCoT，图 3 展示了该 prompt 的一个示例。这个 prompt 包含若干个人工撰写的演示样例（即：需求 - SCoT）和一个新的需求。这些演示样例覆盖了三种基本程序结构和输入输出结构。斜体字是面向大模型的自然语言指令，描述任务的定义。大模型从演示样例中学习，并为新需求生成相应的 SCoT。

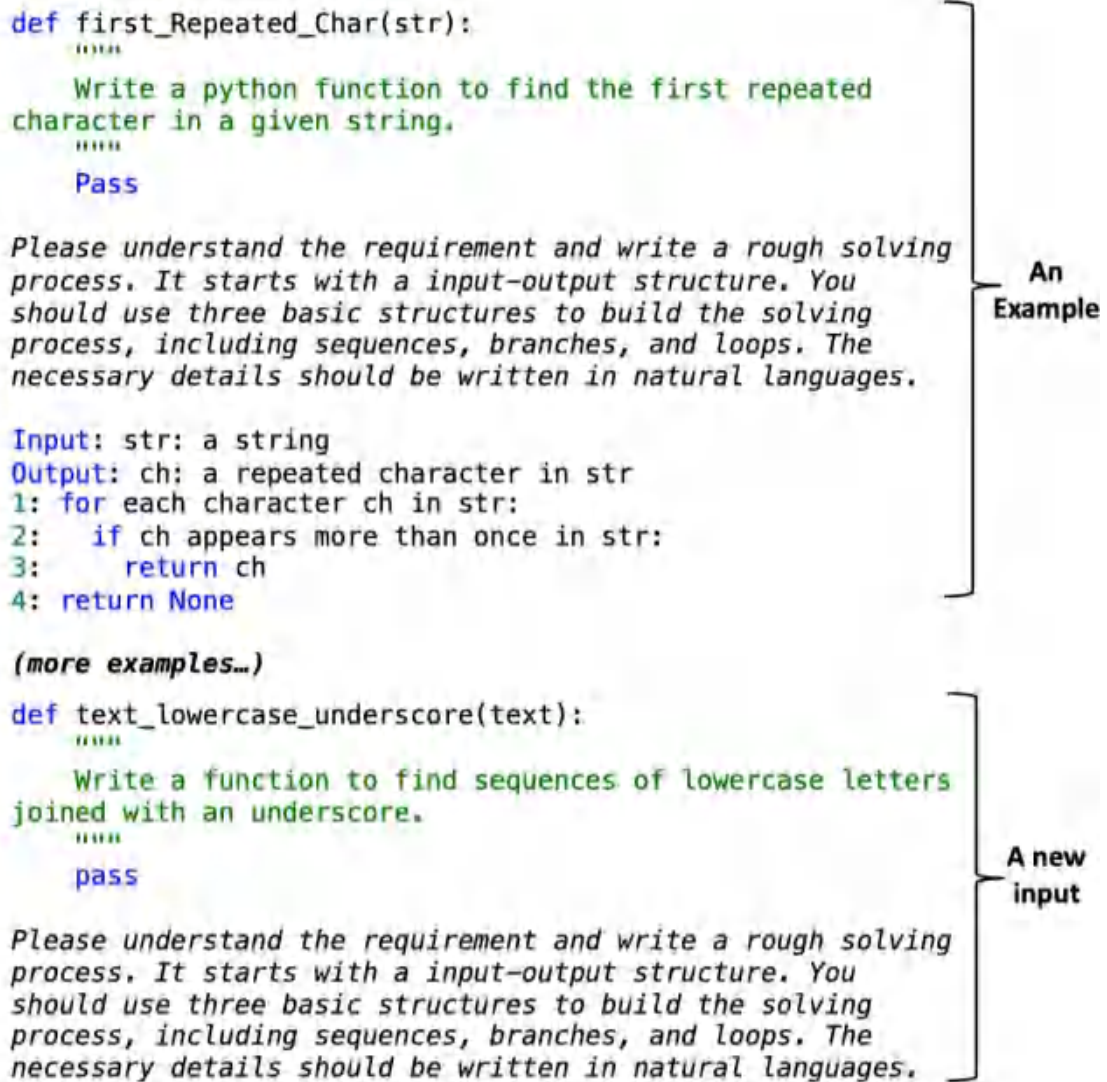


Figure 3: A prompt for generating a SCoT.

生成一段 SCoT 之后，研究者们设计第二种 prompt 来利用大模型生成最终的代码。图 4 展示了第二种 prompt 示例。这个 prompt 包含若干个人工撰写的演示样例（即：需求 - SCoT - 代码），以及新的需求和 SCoT。斜体字是面向大模型的自然语言指令，描述任务的定义。大模型从演示样例中学习，并基于新需求和 SCoT 生成相应的源代码。

```
def first_Repeated_Char(str):
    """
    Write a python function to find the first repeated
    character in a given string.
    Input: str: a string
    Output: ch: a repeated character in str
    1: for each character ch in str:
    2:   if ch appears more than once in str:
    3:     return ch
    4: return None
    """

    # Please check the above solving process and write a
    code based on it. Note that the solving process may contain
    errors.
    h = {}
    for ch in str:
        if ch in h:
            return ch;
        else:
            h[ch] = 0
    return None

(more examples...)
def text_lowercase_underscore(text):
    """
    Write a function to find sequences of lowercase letters
    joined with an underscore.
    Input: text: str
    Output: seq: list of str
    1: Initialize seq as an empty list
    2: for each word in text split by space do
    3:   if word matches the pattern of lowercase
    letters joined with an underscore then
    4:   append the word to seq
    5: return seq
    """

    # Please check the above solving process and write a
    code based on it. Note that the solving process may contain
    errors.
```

Figure 4: A prompt for generating the code.

现有研究发现：多阶段的生成方法容易受到错误积累的影响。类似地，在 SCoT prompting 中，第一步生成的 SCoT 中可能包含噪声（例如：错误步骤）。这些噪声会误导后续的编码实现，导致生成错误的代码。针对这一点，研究者们采用了两种方法来缓解错误积累问题。

- 如图 4 所示，研究者们要求大模型去检查 SCoT，并修复其中可能的错误。这允许大模型选择性地参考 SCoT 并忽略其中的噪声。

- 此外，SCoT prompting 采用了一种两阶段的生成流程，这提供了一个与人交互的窗口。在实际场景中，用户可以先检查 SCoT 的正确性并修复其中问题，然后再使用 SCoT 生成代码。

五、实验设计

研究者设计了一个大规模的评估来回答四个研究问题：

- 问题 1：相较于现有方法，SCoT prompting 在代码生成上的准确率如何？
- 问题 2：人类程序员是否更偏爱 SCoT prompting 生成的代码？
- 问题 3：SCoT prompting 对于不同的演示样例是否是鲁棒的？
- 问题 4：SCoT prompting 中不同程序结构的贡献是怎么样的？

数据集 & 评估指标

研究者在三个流行的代码生成数据集上进行评估，包括：HumanEval、MBPP 和 MBCPP。三个数据集的统计结果如表 1 所示。

研究者们采用单元测试来衡量生成的代码的正确性，并计算 Pass@k。

Table 1: Statistics of the datasets in our experiments.

| Statistics | HumanEval | MBPP | MBCPP |
|-----------------------|-----------|--------|-------|
| Language | Python | Python | C++ |
| # Train | – | 474 | 413 |
| # Test | 164 | 500 | 435 |
| Avg. tests per sample | 7.7 | 3 | 3 |

Baselines

研究者挑选了代码生成上已有的三种 prompting 方法作为 baselines。

- Zero-shot prompting：利用大模型基于需求直接生成源代码，不需要演示样例；

- Few-shot prompting: 随机地挑选一些需求 - 代码对作为演示样例，利用大模型为一个新的需求直接生成源代码；
- Chain-of-Thought prompting: few-shot prompting 的一个变体，采用需求 - 思维链 - 代码作为演示样例，引导大模型先生成一段思维链，再生成源代码。

六、实验结果及分析

问题 1：相较于现有方法，SCoT prompting 在代码生成上的准确率如何？

Table 2: The Pass@k (%) of SCoT prompting and baselines on three code generation benchmarks. The numbers in **red** denote SCoT prompting's relative improvements compared to the SOTA baseline - CoT prompting.

| Base Model | Prompting Technique | HumanEval | | | MBPP | | | MRCPP | | |
|----------------------|---------------------|---------------|--------------|--------------|---------------|--------------|--------------|--------------|--------------|--------------|
| | | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 |
| ChatGPT | Zero-shot prompting | 49.73 | 66.07 | 71.54 | 37.07 | 43.54 | 48.58 | 47.53 | 60.09 | 64.22 |
| | Few-shot prompting | 52.47 | 69.32 | 74.10 | 40.00 | 49.82 | 53.13 | 52.58 | 63.03 | 66.11 |
| | CoT prompting | 53.29 | 69.76 | 75.52 | 41.83 | 51.04 | 54.57 | 53.51 | 63.84 | 67.03 |
| | SCoT Prompting | 60.64 | 73.53 | 77.32 | 46.98 | 55.31 | 58.36 | 57.06 | 65.70 | 68.70 |
| Relative Improvement | | 13.79% | 5.40% | 2.38% | 12.31% | 8.37% | 6.95% | 6.63% | 3.81% | 3.49% |
| Codex | Zero-shot prompting | 40.28 | 61.78 | 68.11 | 27.07 | 43.81 | 47.93 | 40.25 | 54.17 | 60.65 |
| | Few-shot prompting | 42.93 | 62.96 | 70.10 | 33.17 | 45.72 | 49.62 | 44.12 | 57.65 | 62.45 |
| | CoT prompting | 43.79 | 63.41 | 71.56 | 35.66 | 46.37 | 50.11 | 45.79 | 58.92 | 62.56 |
| | SCoT Prompting | 49.82 | 66.56 | 75.14 | 38.29 | 50.74 | 53.16 | 48.34 | 60.77 | 64.19 |
| Relative Improvement | | 13.77% | 4.97% | 5.00% | 7.98% | 8.95% | 6.09% | 5.57% | 3.14% | 2.61% |

研究者将 baselines 和 SCoT prompting 应用至两个大模型（Codex 和 ChatGPT）上，并衡量它们在三个数据集上的 Pass@k。实验结果如表 2 所示。SCoT prompting 在三个数据集上显著地超越了所有的 baselines。相较于 CoT prompting，在 Pass@1 上，SCoT prompting 在三个数据集上分别取得了 13.79%、12.31% 和 6.63% 的相对提升。这些提升显示了 SCoT prompting 在代码生成上的有效性。

问题 2：人类程序员是否更偏爱 SCoT prompting 生成的代码？

代码生成的目的是辅助人类程序员撰写代码。因此，研究者们雇佣了 10 名人类开发者作为评估员，来评估不同方法生成的代码。评估指标如下所示：

- 正确性：代码是否正确地实现了需求；
- 代码异味：代码是否包含代码异味；

- 可维护性：代码的实现是否标准，是否具有较好的可读性。

每项指标的细节请见论文原文。每个指标的分数是一个从 0 到 2 的整数，分数越大则表明在该方面表现越好。人工评估的结果如表 2 所示。SCoT prompting 在三个指标上的得分都稳定地优于 baselines。

Table 3: The results of human evaluation in three aspects. The numbers in **red denote SCoT prompting’s relative improvements compared to the SOTA baseline - CoT prompting. All the p -values are substantially smaller than 0.05.**

| Approach | Correctness | Code Smell | Maintainability |
|----------------------|---------------|---------------|-----------------|
| Zero-shot prompting | 1.012 | 1.523 | 1.372 |
| Few-shot prompting | 1.119 | 1.653 | 1.552 |
| CoT prompting | 1.225 | 1.689 | 1.616 |
| SCoT prompting | 1.412 | 1.869 | 1.873 |
| Relative Improvement | 15.27% | 10.66% | 15.90% |

| | |
|---|--|
| Requirement: # Return True is list elements are monotonically increasing or decreasing. | |
| Few-shot prompting: <pre>def monotonic(l: list): if all(l[i] <= l[i+1] for i in range(len(l)-1)) or all(l[i] >= l[i+1] for i in range(len(l)-1)): return True else: return False</pre> | |
| SCoT prompting (SCoT & Source Code): Input: l: list Output: True or False 1: for each element in l do 2: if the element is greater than its previous element then 3: l increases 4: if the element is less than its previous element then 5: l decreases 6: if both increase and decrease then 7: return False 8: else 9: return True | |
| | <pre>def monotonic(l: list): increasing = False decreasing = False for i in range(1, len(l)): if l[i] > l[i-1]: increasing = True if l[i] < l[i-1]: decreasing = True if increasing and decreasing: return False else: return True</pre> |

Figure 5: Two programs generated by few-shot prompting and SCoT prompting, respectively.

图 5 展示了 few-shot prompting 和 SCoT prompting 在同一个需求上的输出。两个方法生成的代码都通过了所有的测试用例。但 few-shot prompting 生成的代码中包含一条很晦涩难懂的条件语句。在实际场景中，程序员需要花费额外的精力去理解和维护这样的程序。相较之下，SCoT prompting 生成的代码具有较好的可读性，更易于维护。此外，SCoT 清晰地解释了代码的整体行为，可以当做代码的注释，便于后续的维护。

问题 3: SCoT prompting 对于不同的演示样例是否是鲁棒的？

如图 3 和图 4 所示，SCoT prompting 需要一些人工撰写的演示样例来制作 prompt。在真实世界中，不同的用户会写出不同的样例，这可能会导致 SCoT prompting 的性能有一些波动。因此，研究者们探究 SCoT prompting 对于演示样例的鲁棒性。

研究者们从两个方面探究 SCoT prompting 的鲁棒性：

- 样例的选择。研究者们随机地选择多组需求 - 代码对作为种子，然后要求一名标注人员基于不同的种子撰写演示样例。之后，研究者们衡量 SCoT prompting 在不同演示样例上的性能；
- 写作风格。不同的标注人员有不同的写作风格。研究者挑选一组需求 - 代码作为种子，雇佣多名标注人员基于相同的种子撰写演示样例。之后，研究者们衡量 SCoT prompting 在不同演示样例上的性能。

为了比较，研究者们同样衡量了 CoT prompting 在上述场景下的鲁棒性。

Table 5: The Pass@k of CoT prompting and SCoT prompting with different example seeds.

| Seed | CoT prompting | | | SCoT prompting | | |
|--------|---------------|--------|--------|----------------|--------------|--------------|
| | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 |
| Seed A | 53.29 | 69.76 | 75.52 | 60.64 | 73.53 | 77.32 |
| Seed B | 52.81 | 68.97 | 74.55 | 60.27 | 73.11 | 77.16 |
| Seed C | 51.36 | 67.44 | 73.62 | 59.36 | 72.88 | 76.79 |

Table 6: The Pass@k of CoT prompting and SCoT prompting with different annotators.

| Annotator | CoT prompting | | | SCoT prompting | | |
|-------------|---------------|--------|--------|----------------|--------------|--------------|
| | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 |
| Annotator A | 53.29 | 69.76 | 75.52 | 60.64 | 73.53 | 77.32 |
| Annotator B | 51.43 | 67.92 | 73.44 | 59.48 | 72.16 | 76.44 |
| Annotator C | 52.18 | 68.45 | 74.71 | 60.02 | 73.15 | 77.24 |

实验结果如表 5 和表 6 所示。SCoT prompting 对于演示样例具有较强的鲁棒性。它并不依赖于特定的样例或者写作风格，在不同的设置下都优于 CoT prompting。

问题 4: SCoT prompting 中不同程序结构的贡献是怎么样的？

SCoT 中包括三种基本结构和输入输出结构。研究者们进一步探究了不同的程序结构对最终性能的贡献。具体来说，研究者们分别将基本结构和输入输出结构移除，然后衡量 SCoT prompting 在三个数据集上的性能。

Table 4: The results of ablation study. The base model is ChatGPT.

| Prompting Technique | HumanEval | | | MBPP | | | MBCPP | | |
|----------------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 |
| CoT prompting | 53.29 | 69.76 | 75.52 | 41.83 | 51.04 | 54.57 | 53.51 | 63.84 | 67.03 |
| SCoT prompting | 60.64 | 73.53 | 77.32 | 46.98 | 55.31 | 58.36 | 57.06 | 65.70 | 68.70 |
| w/o Basic structures | 55.67 | 70.94 | 76.13 | 43.36 | 53.64 | 56.57 | 54.79 | 64.32 | 67.77 |
| w/o IO structure | 59.65 | 72.79 | 77.12 | 46.13 | 54.76 | 57.88 | 56.61 | 65.01 | 68.42 |

实验结果如表 4 所示。从中可以看出，基本结构和输入输出结构都是必要的。研究者们进一步观察了具体的样例，并定性地分析了不同程序结构的作用。详情可见论文原文。

七、讨论

SCoT 和伪代码的比较

本文的 SCoT 与伪代码具有一些相似之处。二者都包含输入输出结构和一个大致的解题流程。研究者们随机挑选了 100 条生成的 SCoTs。经过人工检查，研究者们发现，26% 的 SCoTs 与伪代码很相近。其余大部分（74%）的 SCoTs 与伪代码不同，因为 SCoT 更加的抽象，不包含具体的实现细节。研究者们认为这种一定程度的相似性也增强了 SCoT prompting 的可用性。在实际场景中，程序员可以通过 SCoT 快速地了解代码的整体行为，也可以使用 SCoT 作为代码注释，便于后续的维护。

Table 7: The comparison of SCoT-P prompting and SCoT prompting. The numbers in red denote SCoT prompting's relative improvements compared to SCoT-P prompting.

| Approach | HumanEval | | | | MBPP | | | MBCPP | | |
|----------------------|-----------|--------|--------|---------|--------|--------|--------|--------|--------|--------|
| | Pass@1 | Pass@3 | Pass@5 | Pass@10 | Pass@1 | Pass@3 | Pass@5 | Pass@1 | Pass@3 | Pass@5 |
| CoT prompting | 53.29 | 69.76 | 75.52 | 41.63 | 51.04 | 54.57 | 53.51 | 63.84 | 67.03 | |
| SCoT-P prompting | 55.23 | 70.33 | 75.94 | 43.28 | 52.16 | 55.72 | 54.25 | 64.09 | 67.78 | |
| SCoT prompting | 60.64 | 73.53 | 77.32 | 46.98 | 55.31 | 58.36 | 57.06 | 65.70 | 68.70 | |
| Relative Improvement | 9.80% | 4.55% | 1.82% | 8.55% | 6.04% | 6.64% | 5.38% | 2.51% | 1.36% | |

为了进一步验证 SCoT 的优越性，研究者们设计了一个变体 - SCoT-P prompting。它与 SCoT prompting 有相同的流程，但采用伪代码作为思维链。表 7 展示了 SCoT prompting 和 SCoT-P prompting 的比较结果。从中可以看出，SCoT prompting 稳定地优于 SCoT-P prompting。这展示了本文 SCoT 在代码生成上的优越性。

SCoT prompting 和排序技术的比较

最近，一些研究人员提出各种排序技术（例如：CodeT）来提升大模型在代码生成上的准确率。针对一个需求，他们先利用大模型生成大量的候选代码，然后利用测试用例或者神经网络对候选代码进行排序，选出其中的 Top-n 个代码作为最终输出。

研究者们并没有将 SCoT prompting 与这类排序技术直接对比，主要原因是：SCoT prompting 和排序技术的应用场景不同，且二者是互补的。SCoT prompting 旨在设计更有效的 prompt 来提升大模型的准确率。排序技术并不关心大模型，而是聚焦于从大模型的输出中挑选出更好的代码。在实际场景中，程序员可以先使用 SCoT prompting 生成大量的候选代码，再使用排序技术挑选最终输出。

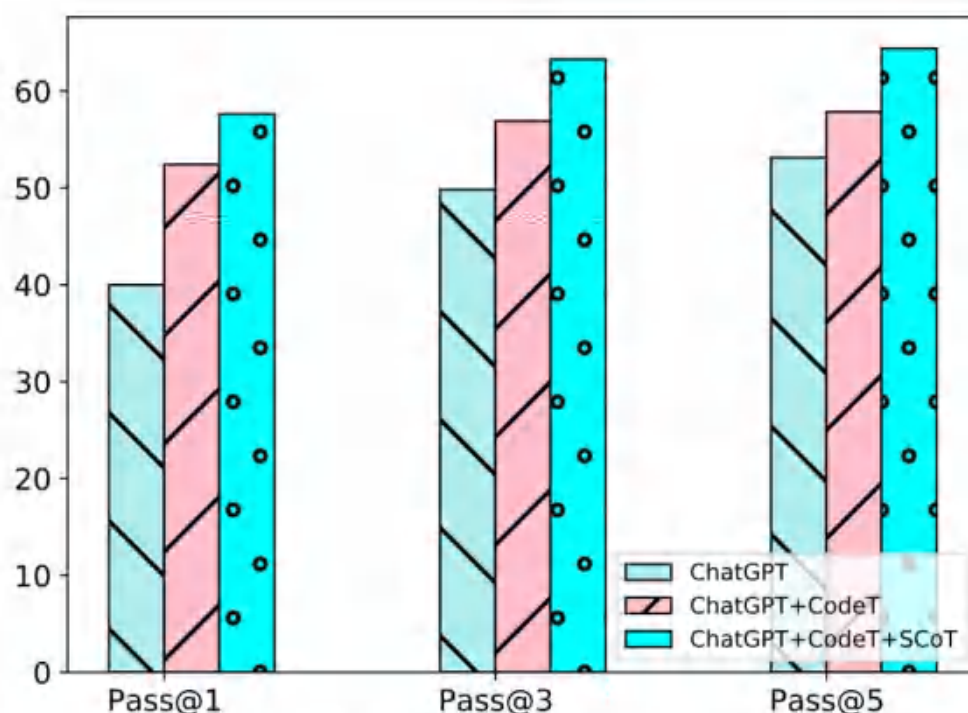


Figure 8: The complementarity between CodeT and SCoT prompting.

为了验证两种方法的互补性，研究者们挑选了一个经典的排序技术 - CodeT。研究者们将 ChatGPT 作为基础模型，逐渐地引入 CodeT 和 SCoT prompting。实验结果如图 8 所示。可以看出，引入两种方法不断地提升 ChatGPT 的准确率。

八、总结和未来工作

本文提出了一种结构化的思维链（SCoT），用于提升大模型在代码生成上的准确率。它约束大模型使用程序结构去组织思维过程，引导大模型从程序语言的角度去思考如何解决需求。在三个 benchmarks

上的实验结果表明了 SCoT 的有效性。

未来，研究者们会进一步探索如何提升大模型在代码生成上的可用性，包括：基于上下文的代码生成、长代码生成等等。**这些研究成果和技术也将有助于 aiXcoder 进行产品研发，打磨更高效、便捷的智能化开发产品。**