When Do Program-of-Thought Works for Reasoning?

Zhen Bi1,2, Ningyu Zhang1,2,*, Yinuo Jiang1,2, Shumin Deng4, Guozhou Zheng1,2,3, Huajun Chen1,2,3,*

1Zhejiang University
2Zhejiang University - Ant Group Joint Laboratory of Knowledge Graph
3Donghai Laboratory
4NUS-NCS Joint Lab, National University of Singapore

{bizhen_zju, zhangningyu, 3200100732, guozhou, huajunsir}@zju.edu.cn, shumin@nus.edu.sg

Abstract

In the realm of embodied artificial intelligence, the reasoning capabilities of Large Language Models (LLMs) play a pivotal role. Although there are effective methods like program-of-thought prompting for LLMs which use programming language to tackle complex reasoning tasks, the specific impact of code data on the improvement of reasoning capabilities remains under-explored. To address this gap, we propose complexity-impacted reasoning score (CIRS), which combines structural and logical attributes, to measure the correlation between code and reasoning abilities. Specifically, we use the abstract syntax tree to encode the structural information and calculate logical complexity by considering the difficulty and the cyclomatic complexity. Through an empirical analysis, we find not all code data of complexity can be learned or understood by LLMs. Optimal level of complexity is critical to the improvement of reasoning abilities by program-aided prompting. Then we design an auto-synthesizing and stratifying algorithm, and apply it to instruction generation for mathematical reasoning and code data filtering for code generation tasks. Extensive results demonstrates the effectiveness of our proposed approach.

Introduction

Large language models (LLMs) (OpenAI 2023; Anil et al. 2023), have emerged as a general-purpose problem-solving methodology for embodied artificial intelligence. In the realm of embodied AI, the reasoning capabilities of LLMs play a pivotal role, especially when agents need to comprehend the semantic intricacies of their environment for effective control (Chen et al. 2022b; Huang et al. 2022, 2023; Wang et al. 2023a). Recent approaches (Chen et al. 2022a; Gao et al. 2022; Cheng et al. 2023), which we term program-of-thought, leverages programming language as a superior prompting mechanism for complex reasoning tasks. In contrast to chain-of-thought prompting (Wei et al. 2022), program-of-thought prompting disentangles the problems into executable code segments and address them step-by-step. However, the correlation between the programming language utilization and the improvement in reasoning ability for LLMs is under-studied. The essential question still remains: When do program-of-thought prompting works for reasoning?

In this work, we propose the Complexity-Impacted Reasoning Score (CIRS), a comprehensive metric for the relationship between code reasoning steps and their impacts on LLMs’ reasoning capacities. We postulate that programming languages hold distinct advantages due to: (1) their superior modeling of intricate structures compared to serialized natural language, (2) their inherent procedure-oriented logic, which assists in addressing multi-step reasoning problems. We posit that our metric should evaluate the code complexity from both structural and logical perspectives.

Specifically, we use abstract syntax tree (AST) to calculate the structural complexity of code reasoning steps (rationales). To retain all structural information in AST that is represented as a tree, our approach leverages three AST indicators (node count, node type, depth), which provides a comprehensive understanding of code structures. Meanwhile, inspired by Halsted (Halstead 1977) and McCabe (McCabe 1976)’s theory, we design a method to calculate

---

*Corresponding Author.
Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.
logical complexity by integrating code difficulty and cyclomatic complexity. Thus, the operators, operands and control flow of the code can be taken into account. We can explicitly compute the complexity of logical inherent in the code.

Through an empirical analysis by our proposed CIRS, we find that not all code data of complexity can be learned and understood by LLMs and current LLMs have limited understanding of symbolic knowledge like code. Code blocks with low complexity contain insufficient knowledge, while those with high complexity could be too difficult for LLMs to learn. Consequently, only code data with an optimal level of complexity (structure&logic), neither too simple nor too intricate, contribute to the effective enhancement of LLMs’ reasoning abilities.

Then, we propose the auto-synthesizing and stratifying algorithm that can automatically generate and filter out the data with the most effective reasoning ability. We apply our algorithm to two scenarios: (1) guiding instruction generation for mathematical reasoning tasks. (2) filtering code data for code generation tasks. Compared to baseline models, our proposed method achieves favorable results in mathematical reasoning and shows effectiveness for code generation tasks. In this paper, our contributions are as follows:

- We propose a novel method to measure reasoning complexity for the code data, termed CIRS. Our approach, which evaluates the code data from both structural and logical perspectives, can accurately gauge the correlation between code complexity and its reasoning ability.
- We empirically analyze the impact of varying complexities, identifying that optimal level of code languages, which is leanable for LLMs, as the pivotal factor in the reasoning abilities of program-of-thought prompting.
- We design an auto-synthesizing and stratifying algorithm and apply our approach to both instruction generation for mathematical reasoning and code data filtering for code generation tasks. Extensive results demonstrate the validity of our proposed perspective.

### Background

Code large language models have demonstrated remarkable capabilities in various tasks such as commonsense reasoning (Madaan et al. 2022), information extraction (Wang, Li, and Ji 2022), mathematical reasoning (Imani, Du, and Shrivastava 2023), robotics manipulation (Huang et al. 2023) and embodied learning agent (Wang et al. 2023a). Generally, code LLMs with larger model parameters are more effective than vanilla LLMs for reasoning. We find that even if Codex (Chen et al. 2021) and GPT-3.5 (Brown et al. 2020) are with same parameters, Codex that is pre-trained on code corpus performs better than GPT-3 on problems such as arithmetic reasoning and structural prediction tasks. Intriguingly, training on code data not only enables the ability of code understanding but may also foster the reasoning ability.

Inspired by Chen et al. (2022a); Gao et al. (2022), we formalize the multiple-step reasoning tasks by using code-format chain-of-thoughts. For program-of-thought prompting, given the input for the reasoning problem Q, we aim to maximize the likelihood of the answer A as $p(A|Q)$.

$$p(A|Q) = p(A|Q, R_c)p(R_c|Q)$$ (1)

where $R_c$ is the solution of the code which will be generated. We enhance the effectiveness of solving multi-step reasoning problems by using code prompts as intermediate steps.

#### Complexity-Impacted Reasoning Score

To measure the the reasoning ability of the code rationale $R_c$, we define the complexity-impacted reasoning score as the product of structural complexity $\text{Score}_{SC}$ and logical complexity $\text{Score}_{LC}$.

$$\text{Score}(R_c) = \text{Score}_{SC}(R_c) \times \text{Score}_{LC}(R_c)$$ (2)

### Structural Complexity

To calculate the structural complexity, we measure the structural complexity of the Abstract Syntax Tree (AST). We design a simple yet effective method by selecting three indicators that can provide a comprehensive understanding of structural information. Therefore, we define the $\text{Score}_{SC}$ as follows:

$$\text{Score}_{SC}(R_c) = \text{Sigmoid}(f(x_{\text{Node}}, x_{\text{Type}}, x_{\text{Depth}}))$$ (3)

where $x_{\text{Node}}, x_{\text{Type}}$ and $x_{\text{Depth}}$ are the features of node count, node types and tree depth in the AST of the code rationale $R_c$. We first use the function $f$ to apply Z-score normalization to the accumulated data $x$ for each feature, and then we aggregate the overall information by mean pooling. Next, we apply the Sigmoid function to transform the data into the range of 0 to 1. The benefit of doing this is to preserve the distribution characteristics of the feature and avoid being influenced by extreme statistical data, whether it is exceptionally large or small. The detailed explanations for three indicators are as follows:

- **Node Count.** The number of nodes reflects the size of the code. Generally, more nodes indicate higher complexity. But node count alone cannot comprehensively measure code complexity because a large code with a simple structure might be easier to understand than a smaller code with a complex structure.

- **Node Types.** Node types help identify the structural elements present in the code, such as conditional statements, loops, and function calls. Different node types play different roles in the code and contribute differently to its complexity. Therefore, tracking the quantity of various node types can enhance our understanding of the structural complexity of the code.

- **Tree Depth.** The depth of the AST reflects the level of nesting in the code. A greater tree depth may imply more complex control flow and logic, making the code harder to understand. It is important that depth alone is also not the sole measurement criterion. A shallow tree with multiple simple branches might be easier to comprehend than a deep tree with a few complex branches.
Logical Complexity We define code logical complexity Score\(_{LC}\) integrating the difficulty \(D\) and cyclomatic complexity \(V\) which is inspired by Halstead Complexity Metrics (Halstead 1977) and McCabe’s Cyclomatic Complexity (McCabe 1976).

\[
\text{Score}_{LC}(R_c) = \text{Sigmoid}(D(R_c) \times V(R_c))
\]

where Difficulty \(D(R_c)\) denotes the difficulty for solving the problem and \(V(R_c)\) means cyclomatic complexity of the rationale \(R_c\). To represent the effort required to comprehend the program, the Difficulty \(D(R_c)\) is defined as:

\[
D(R_c) = \left( \frac{n_1}{2} \right) \cdot \frac{N_2}{n_2}
\]

where \(n_1\) denotes the number of different operators and \(N_2\) denotes the total number of operands in the code, \(n_2\) denotes the number of distinct operands in the code rationale \(R_c\). In this formula, the term \((n_1/2)\) represents the average complexity of operators, while the term \((N_2/n_2)\) represents the average complexity of operands.

To consider the complexity of the logical loops (code control flow), we define the cyclomatic complexity \(V(R_c)\) as:

\[
V(R_c) = E - N + 2
\]

where \(E\) denotes the number of edges in the control flow graph in the code and \(N\) denotes the number of nodes in the control flow graph. We employ the Sigmoid function to constrain the values of code logical complexity. There is a significant correlation between potential program errors and high cyclomatic complexity. We note that high cyclomatic complexity indicates that the program code has complex judgement logic, potentially leading to lower quality. It might be difficult to test and maintain those code with high cyclomatic complexity. Generally, by integrating the difficulty and cyclomatic complexity, both the complexity of the operators, operands, and control flow of the code can be taken into account. Next, we conduct experimental analysis to empirically study the rationality of our method.

Experimental Settings

In order to conduct an unbiased evaluation of all model performances, we use zero-shot and few-shot settings for evaluation. For zero-shot setting, we directly presenting mathematical problems to the model for solution generation, without any demonstrations in the input. For few-shot setting, we choose 3-shot for evaluation where we select three in-context examples with rationales. Our criterion for evaluation is that the answer is considered ultimately correct only if the code executor’s answer is correct.

In the empirical analysis, we conduct an empirical analysis of the variations in different model sizes and complexities in zero-shot setting. We build our own test dataset because there are no publicly available benchmarks. The evaluation is performed on AsDiv (Miao, Liang, and Su 2020), GSM8K (Cobbe et al. 2021), MultiArith (Roy and Roth 2015), and SVAMP (Patel, Bhattamishra, and Goyal 2021), with a selection of 500 instances randomly chosen from each original testset to form the new testsets. We choose gpt-3.5-turbo as the main benchmark model and accuracy (Acc) as our main evaluation metric.
To use CIRS for improving the reasoning ability, we train the model based on the LLaMA-7B (Version 1.0) (Touvron et al. 2023). Vicuna (Chiang et al. 2023) and Falcon (Almazrouei et al. 2023) are selected as the main comparison models and accuracy (Acc) is chosen as the evaluation metric again. Apart from the datasets used in the in-distribution setting, the model is also evaluated on MATH (Hendrycks et al. 2021) and BigBench-Hard (Suzgun et al. 2022) in the out-of-distribution setting. It should be noted that we only choose level-1 problems in MATH. We use algorithmic and multi-step arithmetic reasoning tasks in BIG-Bench Hard. The experimental setup is shown in the supplementary.

Empirical Analysis
In this section, we empirically analyze the impact of different forms of code data. Specifically, we synthesize a totally new dataset and manually partition it using our CIRS. Next, we discuss the impact of code data with different complexities on the reasoning abilities for LLMs. Then we analyze the characteristics of code data with varying complexities and conduct more ablation analysis.

Data Synthesizing

<table>
<thead>
<tr>
<th>Seed source</th>
<th>Seed size</th>
<th>Data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AQuA</td>
<td>97,467</td>
<td>10,160</td>
</tr>
<tr>
<td>GSM8K</td>
<td>7,644</td>
<td>12,812</td>
</tr>
<tr>
<td>MultiArith</td>
<td>600</td>
<td>12,163</td>
</tr>
<tr>
<td>ASDiv</td>
<td>2,306</td>
<td>13,554</td>
</tr>
<tr>
<td>SVAMP</td>
<td>3,954</td>
<td>12,441</td>
</tr>
<tr>
<td><strong>ALL</strong></td>
<td></td>
<td><strong>61,130</strong></td>
</tr>
</tbody>
</table>

Table 1: Statistics of seeds and the generated data size.

To fairly explore the impact of the variations in different complexity scores, it is necessary to avoid errors caused by the dataset itself and generate entirely new forms of code data. The sources of seed data include the training set of GSM8K (Cobbe et al. 2021), MultiArith (Roy and Roth 2015), Asdiv (Miao, Liang, and Su 2020), SVAMP (Patel, Bhattacharya, and Goyal 2021) and AQuA (Ling et al. 2017). In Table 1, we have synthesized over 60,000 samples from five seed datasets. For each dataset, we generate approximately 10,000 samples. We choose as many datasets as possible to ensure the diversity of mathematical problems.

Then, we design a pipeline that can automatically generate high-quality code corpus by leveraging ChatGPT. As shown in Figure 2, we apply a template to define the format and then allow the API to continuously rewrite new questions and their corresponding code-format solutions. In the construction of templates, we randomly select three problems from the seed datasets each time. Next, we automatically filter out the generations that do not conform to Python syntax standards, which results in a collection of high-quality mathematical problems. For all generated data, we randomly sampled 10% and verified its correctness by manual checks and automated validation with GPT-4, ensuring the accuracy within a reasonable margin of error.

After obtaining well-generated code data, we utilize our proposed CIRS and manually split the data into different subsets based on the analysis of code complexity distribution. We put the visualized results in the supplement. Based on different complexity scores, we name the partitioned subsets as low (lower score samples), medium (medium score samples) and high (high score samples).

Impacts of Different Complexity Score

To compare the impact of different code complexities on the reasoning capability of LLMs, we train three models based on LLaMA (Version 1.0) from 7 billion to 65 billion parameters. We randomly select 1,700 instances from each subset (low, medium, high) to build the training and validation dataset for fair comparisons. Results are shown in Figure 3.

Optimal level of code is crucial to the reasoning abilities of program-of-thought prompting. From the results across the four datasets, we note that the model performs optimally when the complexity of the code data is in middle range. This suggests that the learnable symbolic language is crucial to the reasoning abilities of program-aided prompting. The reasoning behind this is that data with overly simplistic complexity, is too simple for LLMs, leading to less noticeable effects. Conversely, when the complexity escalates significantly, the logical semantics and nested structures become difficult to comprehend or learn, which could adversely impact the reasoning capabilities of LLMs.

The larger the number of parameters, the more significant the gain in LLM’s reasoning capabilities. As the model size increases from 7 to 65 billion parameters, the effectiveness of its reasoning capability improves. In fact, most 65 billion fine-tuned models can achieve results comparable to those of gpt-3.5-turbo. It suggests that having a sufficient number of parameters is crucial for emergent reasoning capabilities in LLMs. Furthermore, when the LLM is large enough, the difference in results across various complexities is minimal. This indicates that LLMs with vast parameters are more prone to symbolic data and inherently have the potential to yield strong reasoning capabilities.

Current LLMs have limitations in their understanding capabilities for reasoning. We observe that when data complexity is extremely high, the performance of LLMs tends to decrease. It reflects that there is an inherent limit to the reasoning capabilities of large language models. We argue that: (1) The current architecture of LLMs (such as decoder-only LLM) has limited ability to understand complex knowledge, which also restricts the emergence of their reasoning capabilities. The prerequisite for large models to demonstrate powerful reasoning abilities is their ability to comprehend the structures and logical knowledge embedded in complex data. Therefore, it is necessary to explore model structures with stronger reasoning abilities in future research. (2) Further enhancement in reasoning power requires the reliance on external tools. We know that the scope of reasoning problems is quite broad, not only mathematical reasoning, but also including commonsense or more complex logical reasoning tasks. Therefore, relying solely on the LLM itself is not enough to resolve all issues at once; the assistance of more powerful external tools is required.
The Characteristics of Different CIRS Scores

In Figure 5, we investigate the characteristics of different CIRS scores. The different subsets of CIRS scores exhibit distinct structural and logical differences.

- **Textual, minimal programming.** Samples with lower CIRS scores contain little structural information. Although they do contain some intermediary reasoning processes, these are primarily represented in flat textual descriptions. These samples typically correspond to simpler and structurally, logical insufficient problems.

- **Simple but direct programming.** As CIRS score increases in the code reasoning steps, the presence of programming languages with simple logical semantics and structures also escalates. These samples typically involve simple and straightforward logical operations.

- **Complex programming.** Samples with exceedingly high scores contain substantial amounts of structural function definitions or reasoning processes, which suggests the presence of numerous complex conditional statements and function structures. These samples are typically highly challenging mathematical problems.

Excluding the Effect of the Distribution Itself

To negate the potential skew from data distribution itself, such as enhanced performance in the mid-range data due to its higher frequency of occurrence, we conduct a more in-depth analysis of the evaluation results at different complexity scores. We use the trained 7B model and conduct tests on 2,000 samples with three models (CIRS-low, CIRS-medium, CIRS-high). We use CIRS to measure the output reasoning steps for each model and divide them into four categories (low, medium, high and invalid). In Figure 4, we find CIRS-medium generates the highest number of valid predicted outputs in three distributions (17.8%, 61.1%, 9.3%). We also observe that CIRS-medium demonstrates high accuracy (53.4, 46.1, 47.3) in all three distributions. The accuracy of predictions for each distribution by the model is independent of the quantity of training data. We can conclude that the effectiveness of complexity data is not because of the frequency of data occurrence.

Ablation Analysis for Textual Rationales

To verify the effect of code and textual rationales, we substitute the code-format solving process with textual rationales using the same datasets. We sample 1,700 instances of code data within the mid-range complexity and simultaneously construct a dataset that uses textual rationales. We train both two models based on LLaMA-7B. As shown in Figure 6, the code dataset demonstrates a clear advantage in all four
**Textual, fewer programming**

**Question:** A and B start a business … After 2 years, they … of Rs. 48,000. What is B’s share in the ratio of their investments?

```
...python
# Total investment = 60000+80000 = 140000
# A’s share = (60000/140000) * 48000 = 20571.43
...
B_share_in_profit = 27428.57
print(B_share_in_profit)
```

**Simple, direct programming**

**Question:** A restaurant has 3 chefs. Chef A worked for 8 hours, Chef B worked for 6.5 hours, and Chef C worked for 9.25 hours. How many minutes did the chefs work in total?

```
...python
chef_A_hours = 8
chef_B_hours = 6.5
chef_C_hours = 9.25
total_hours = chef_A_hours + chef_B_hours + chef_C_hours
total_minutes = total_hours * 60
print(total_minutes)
```

**Complex programming**

**Question:** There are 3 prime numbers in ascending order. The multiplication of the first 2 is 77 and that of the last 2 is 91. What is the last number?

```
...python
from sympy import primerange
primes = list(primerange(1, 92))
for i in range(len(primes)):
    for j in range(i+1, len(primes)):
        if primes[i] * primes[j] == 77:
            ...
print(last_prime)
```

Figure 5: As the CIRS score increases, there is a greater presence of logical and structural information in the code.

![Graph showing the comparison of textual and code rationales](image)

**Algorithm 1: Auto-Synthesizing and Stratifying**

**Require:** $T$: Template, $K$: Number of clusters, $J$: Threshold set

**Ensure:** $C$: Cluster assignments

- Dataset $D \leftarrow$ template $T$ filling by leveraging API
- Threshold $J \leftarrow$ threshold set generated by CIRS
- Initialize $C$ with random initial cluster assignments

```
repeat
    Clear all clusters
    for each data point $x$ in $D$ do
        Find the nearest centroid $c_j$ in $C$ to $x$
        Assign $x$ to cluster $c_j$
    end for
    for each cluster $c_i$ in $C$ do
        Recalculate centroid $c_i$ as the mean of all points
    end for
    Remove clusters from $C$ if the average distance to their centroid is not in $J$
until no more updates or maximum iterations reached
return $C$
```

**CIRS for Improving the Reasoning Ability**

In this section, we first describe our auto-synthesizing and stratifying algorithm and then we apply our proposed CIRS to two reasoning tasks to validate its effectiveness.

**Auto-Synthesizing and Stratifying Algorithm**

Based on the processing step in the previous section, we formalize the whole procedure into a pipeline (Algorithm 1) for automatic data generation and stratification. We first do the template $T$ filling by calling APIs and get the synthesized dataset $D$. Then we calculate the distribution of complexity for all synthesized data by CIRS and get the threshold set $J$. Next we design a threshold-based $k$-means clustering method that automatically partitions the dataset according to complexity characteristics. Finally, we apply our proposed algorithm for two scenarios to enhance the reasoning abilities of large language models.

**Usage1: CIRS-guided Instruction Generation**

As stated in the previous section, we know that the trained model with complexity optimal level of code data, exhibits the best reasoning capabilities. Therefore, we employ our algorithm to filter out data from the source dataset to train an enhanced reasoning model. Totally, we collect 40,000 mid-range data samples to train a more powerful language model for reasoning. As shown in Table 2, for in-distribution setting, we find that trained model outperforms Vicuna and Falcon. To eliminate the influence of data distribution, we directly test the model’s performance in the out-of-distribution setting. Our model perform best (the same parameters) in both zero-shot and few-shot prompting. It is worth noting that our approach demonstrates comparable effectiveness to ChatGPT on BigBench-Hard in zero-shot setting. For MATH dataset, we notice that our model still outperforms the baseline models. But our model are much worse than ChatGPT which is due to limitation of code data itself.

**Usage2: CIRS-based Code Filtering**

To validate the effectiveness of our approach in code-related tasks, we use the Algorithm 1 to filter a batch of code instruction data. We first split the Code Alpaca (Chaudhary...
Table 2: Results of mathematical reasoning tasks. †We choose algorithmic and multi-step arithmetic reasoning tasks in BigBench Hard. *Here we use Falcon-Instruct which is fine-tuned on instruction datasets.

Table 3: Results of CIRS-based code filtering tasks.

Related Work
Program-aided Prompting  Chen et al. (2022a) prompting delegates computation steps to an external language interpreter and Gao et al. (2022) generates programs as the intermediate reasoning steps. Cheng et al. (2023) is a neural-symbolic framework and Hu et al. (2023) is a neural symbolic prompting method for complex reasoning tasks. Some methods such as (Wang, Li, and Ji 2022; Li et al. 2023; Bi et al. 2023) leverages code prompting methods for information extraction tasks. Madaan et al. (2022) frames the task of structured commonsense reasoning as code generation. Zhu et al. (2023) distills LLMs into specialized, compact models for reasoning tasks by program-aided prompting.

Reasoning with LLMs  The research on reasoning abilities is a core issue in NLP (Qiao et al. 2023; Huang and Chang 2022; Zhao et al. 2023). The success of LLMs have achieved breakthroughs in various tasks or domains (Imani, Du, and Shrivastava 2023; Yang et al. 2022; Zhang et al. 2022; Chen et al. 2023). Some studies (Gendron et al. 2023; Liu et al. 2023; Varshney et al. 2023; Yuan et al. 2023; Schwartz et al. 2020) are focusing on analyzing the capabilities of LLMs. Wang et al. (2023b) improves LLMs reasoning abilities by fine-tuning alignment paradigm. More and more research efforts (Fu et al. 2023b; Mukherjee et al. 2023) are being devoted to unveiling the origin of a model’s reasoning abilities or focus on enhancing the capability of smaller models. Some works (Wiegreffe, Marasovic, and Smith 2021; Xie et al. 2023) generate rationales to enhance model interpretability. To measure reasoning capabilities, (Fu et al. 2023c) propose a scheme based on complexity prompting. (Fu et al. 2023a) is an evaluation suite that measures LLMs’ multi-step reasoning performance.

Discussion and Conclusion
What kind of data format is crucial for LLM’s reasoning abilities? We explore the reasoning abilities for program-of-thought prompting and it indicates that code data with optimal level of code, characterized by certain logical and structural qualities, is the key factor. Code data is efficient because it is inherently semi-structured and abundant in the natural world. We can prove that: (1) The local structural properties of the data are crucial for improving reasoning abilities, which aligns with Prystawski and Goodman (2023). The logical coherence or a certain amount of knowledge circuitry inherent in the data is necessary. (2) Overly complex structural information and logic are ‘too difficult to learn’ for LLMs. The results in this work demonstrate that knowledge of optimal level complexity is most effective because it is learnable for most LLMs. Meanwhile, we find that as the number of parameters in language models increases, their understanding of complex knowledge also improves.

In this work, we introduce CIRS to measure the relation between code reasoning steps and reasoning abilities. By considering both structural and logical attributes of code data, we use AST to encode the structural information and encode structural feature by difficulty and cyclomatic complexity. Through an empirical analysis, we find that optimal level of code languages plays a crucial role in the reasoning abilities of program-of-thought prompting. We develop the auto-synthesizing and stratifying algorithm that applies mathematical reasoning and code generation tasks. Extensive results prove the effectiveness of the proposed method. In the future, we will expand this work to more scenarios such as commonsense or logical reasoning tasks and train powerful reasoning models with low computational cost.
Acknowledgements

We would like to express gratitude to the anonymous reviewers for their kind comments. This work was supported by the National Natural Science Foundation of China (No. 62206246), the Fundamental Research Funds for the Central Universities (226-2023-00138), Zhejiang Provincial Natural Science Foundation of China (No. LGG22F030011), Ningbo Natural Science Foundation (2021J190), Yongjiang Talent Introduction Programme (2021A-156-G), CCF-Baidu Open Fund, and Information Technology Center and State Key Lab of CAD&CG, Zhejiang University, and NUS-NCS Joint Laboratory (A-0008542-00-00).

References


Huang, W.; Wang, C.; Zhang, R.; Li, Y.; Wu, J.; and Fei-Fei, L. 2023. VoxPoser: Composable 3D Value Maps


