Problem Decomposition for Program Synthesis with Large Language Models

Yanzhuo Jin, Zhaohui Yang, Jieran He, Zichen Qiu, Xu Chen

Abstract— **Program synthesis is the process of generating computer programs as solutions to given problem specifications based on input-output examples or natural language descriptions. The proliferation of Large Language Models has propelled the latest advancements in the field of program synthesis. Recent research focuses on improving program synthesis accuracy, primarily by increasing dataset size, leveraging high-quality datasets for fine-tuning the model, performing multi-step manual annotation to guide model outputs, and training on mixed datasets with different programming languages. However, the impact of prompts in program synthesis is under-explored. This paper aims to maximize the performance of input words to enhance the precision of output code. We introduce auto prompt decomposition to the popular concept of "Chain-of-Thought" (CoT) prompts as the input prompt. Our results demonstrate that this method achieves higher accuracy compared to simple prompts and is on par with multi-step manual annotation on the Enhanced dataset: Multi-Turn Programming Benchmark (MTPB). Also, we show that the Large Language Models may prefer copy the example answers you give in chain of thought step without generating the related answer of authentic problems and we provide a detailed prompt to help avoid this situation.**

I. INTRODUCTION

Code generation typically requires manual input of code by humans. However, the goal of program synthesis is to automatically generate computer programs that satisfy user-specified intentions. Achieving program synthesis not only improves the productivity of experienced programmers but also makes programming more accessible and usable to a wider range of users. However, there are two major challenges in achieving program synthesis: the vast search space of programs and the difficulty in correctly understanding user intentions. [1]

In the past decade, training large-scale artificial neural networks has made remarkable progress, especially with the introduction of the Transformer architecture. [2] To address the vast program space, the current approach typically transforms the program synthesis task into a language modeling problem. [2] This is done by learning the conditional distribution of the next token given the previous tokens and leveraging Transformers and large-scale unsupervised pretraining. This approach has been successful in various modalities. [3], [4], [5] To evaluate the accuracy of current program synthesis generative models, they are usually compared on an open-source validation set,like humaneval. [6]

However, the current accuracy of these program generation models is still not high enough, around 67% as maximum(2023 Mar GPT-4 [Ope23]). [6] Some reasons may be due to the overly strict validation conditions that require all outputs to be correct in order to deem the code as correct,which does not capture the nuances of the model's performance. To improve accuracy, innovative methods have been attempted, such as increasing the number of iterations, using teacher-student frameworks for iterative training[8], use reinforcement learning with agent reflection [7],introducing datasets in different programming languages to enhance the model's logical ability [9], and fine-tuning through the construction of high-quality datasets [6] and multiple rounds of manual annotation for iterative generation [1]. These methods help the model understand user intentions indirectly by passing inputs to synthesize programs, using pseudocode, partial program logic expressions, or natural language paragraphs with input-output examples to understand user intentions. [1] This indirect approach overlooks the possibility of maximizing the performance of input words to enhance the accuracy of output code.

To directly assist the model in understanding user intentions, we believe that providing direct assistance on natural language text is feasible. Currently, in large language models, the "chain of thought prompting" approach is used to give the model the ability to think coherently, similar to human thinking processes.[10] This is done through multi-step reasoning to understand the user's true intentions. This approach has several advantages: coherent thinking, decomposes multi-step problems, optimizes resources,provides interpret ability, applies to various tasks, and elicits chain of thought in language models. In addition, compared to single-turn program synthesis, decomposed multi-step program synthesis methods can focus more on the specifications related to a subprogram and avoid the problem of dealing with complex dependencies between subprograms. [1]

In this paper, we adopt a combination of prompt decomposition and "chain of thought" to directly assist the model in improving its thinking and comprehension level. Our results show that compared to simple prompts, this method achieves higher accuracy on enhanced datasets and performs comparably to manually annotated multi-step approaches. Additionally, we demonstrate that large-scale language models may be more inclined to replicate example answers given in the thought chain rather than generating answers relevant to the real problem. To avoid this situation, we provide detailed prompts.

Our contributions

- We contribute a enhanced datasets from MTPB to demonstrate our idea.

- We provide a specific prompt about how to work the prompt decomposition with chain of thought
- We show that the Large Language Models may prefer copy the example answers you give in chain of thought step without generating the related answer of authentic problems and we provide a detailed prompt to help avoid this situation.

II. Related work

A. Program Synthesis and Benchmarks for it

Program synthesis is the process of generating computer programs as solutions to given problem specifications based on input-output examples or natural language descriptions.A large body of prior research attempted to address by exploring methods like stochastic search techniques [14], [15] and deductive top-down search. [16], [17] However, the scalability of these approaches is still limited. Because of the well-learned conditional distribution and language understanding capacity about LLMs, most reasearch will train a model to achieve their idea.

At the area of code evaluation, AlphaCode[18] evaluates a set of generations on hidden test cases. CodeT [19] uses self-generated unit tests that are used to score generated function implementations. Self-Debugging[8] employs a debugging component that is used to improve existing implementations given feedback from a code execution environment. CodeRL [20] sets the problem in an RL framework using an actor-critic setup to debug programs given feedback from an execution environment.

In this paper ,we use GPT-3.5 model and its temperature is 0. In the process of reasearch,we just involve the Python code. To evaluate the code , we use the same pattern as humaneval [13] to achieve evaluation more easily,not like the original evaluation of MTPB using inputs and outputs.

B. Chain of Thought prompting

Traditional opinion on Large Language Models is proposed as scaling laws,which tried to explain the phenomenon that the performance improves somewhat predictably as one scales up either the amount of compute or the size of the network . [12] While the proposition of Chain of thought prompting not only confirms that the large language models can achieve some thinking abilities similar as human ,but also provide a new research guidelines.

Chain of Thought (CoT) is a mechanism where the user provides step-by-step prompts for the machine to answer. This approach allows for accurate responses without significantly increasing labor costs. One approach is the zero-shot method, where the machine automatically generates intermediate reasoning steps and sample answers based on learned examples. 'Let's think step by step 'is a relatively advantageous option without significantly reducing accuracy. [11] Another method is few CoT prompting , where the model learns from manually written few-shot demonstrations. These approaches have demonstrated the capability of LLMs to tackle complex reasoning tasks. [10]

In this paper, we use few shot prompting as the part of the methods . Specifically,we have used it in three places: Firstly ,in code loading from agent(GPT-3.5) ,to avoid some string explanation about the code that may appear some condition that people need to revise by themselves, we give him a example only with clear code as the standard outputs. Secondly, in auto decomposition ,in order to confirm that the agent just generates a multi-step. Finally ,we use it in code generation step.

III. Methods

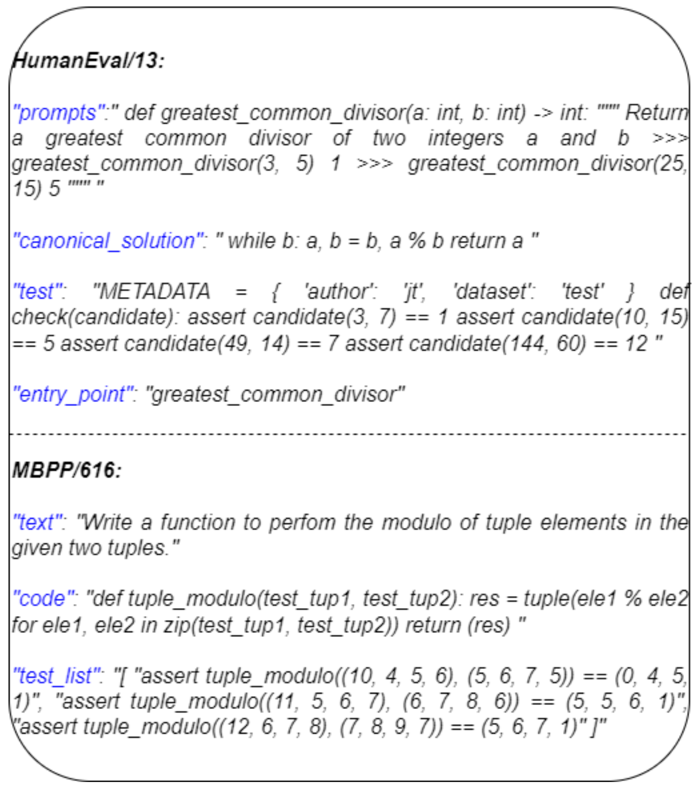


Fig. 1. The example of prompts from evaluation datasets :Humaneval and MBPP [13] [21]

Our idea rises from that recent research focuses on improving program synthesis accuracy through indirect approaches ,which has ignored the impact of prompts in program synthesis. As the picture seen in Fig.1, some prompts combine logical symbol and extra information to mask the actual accuracy. Meanwhile, if you just input a pure natural language text to the LLMs ,it may be difficult for it to generate code if some prompts combine potential multi objects to do without appearing in prompts. This paper aims to maximize the performance of input words to enhance the precision of output code.

A. The Construction of Enhanced datasets

The dataset Multi-Turn Programming Benchmark (MTPB) [1] is a code evaluation benchmark,which has 115 tests. Compared with other evaluation datasets, the main advantage of it is that it contains a mature manual mark about each step which can be considered as a target that has been decomposed.

```
MTPB/id1:

"prompts": [Assign the string \"{A}\" to a variable named \"my_string\".", "Lowercase the given string \"my_string\".", "Assign the distinct characters of the string to a variable named \"chars\".", "Sort these characters in alphabetical order.", "Print the resulting list of characters."]

"inputs":[{"A":"abcde"},{"A":"abcdecadeCADE"},{"A": "aaaaAAAAaaaa"},{"A": "Jerry jERRY JeRRRY"}, {"A": "ddddc"}]

"outputs": [["a", "b", "c", "d", "e"], ["a", "b", "c", "d", "e"], ["a"], [" ", "e", "j", "r", "y"], ["c", "d"]]

"max_gen_length": 128

"category": "string"

"name": "Sandwich string"

"description": "Append a string in the middle of another string.",
```

Fig. 2. The example of prompts from evaluation datasets: MTPB [1]

```
EMTPB/id1:

"summary_prompt":"Define a function to return a sorted list of distinct characters obtained from a given string, after converting it to lowercase."

"test": "def check(candidate):\n assert candidate(\"abcdecadeCADE\")==
[\"a\", \"b\", \"c\", \"d\", \"e\"] \n assert candidate(\"Jerry jERRY JeRRRY\")==
[\" \",\"e\",\"j\", \"r\", \"y\" ]\n",

"prompts":["Assign the string \"{A}\" to a variable named \"my_string\".", "Lowercase the given string \"my_string\".", "Assign the distinct characters of the string to a variable named \"chars\".", "Sort these characters in alphabetical order.", "Return the resulting list of characters."],

"mark":"True"

"example":"input \"abcdj\" \n output:
\"a\", \"b\", \"c\", \"d\", \"j\""
```

Fig. 3. The example of prompts from evaluation datasets: Enhanced MTPB(EMTPB)

In figure2,the original description cannot cover up prompts.Therefore,what we mainly need to do is to manually combine the all steps into a summary prompt with similar pattern as the pure natural language text in MBPP. Secondly, in order to test codes about whether the code generated by LLMs can standard for the prompts more easily ,we change the inputs and outputs into a test function which can be tested by ”exec()”.Thirdly, we change the token ”print” into ”return”in prompts to confirm the generation is a class or a function. Next, to help the LLMs have better understanding of the prompts,we manually add a example to EMTPB. Finally ,if some prompts cannot be summarized by human ,we mark ”False” and the others will be marked ”True”.

B. Auto Decomposition with Chain of Thought

The Auto Decomposition with Chain of Thought is mainly achieved by three prompts. As the flow path of figure 7, one in agent1 and two in agent2. In this paper ,we use GPT-3.5 LLMs model as the agents with different system prompts and user prompts with 0 temperature . Also, in the process of reasearch,we just involve the Python code.

Firstly,in order to confirm that the agent just generates a multi-step, we create a new example with same pattern as EMTPB. The challenge of it is how to design the prompts. In zero shot CoT prompting,”Let’s think step and step” has proved its performance [11] therefore, it is added as a head part. Like Figure 4, each steps are confirmed just move a small step away from the prior. In prompting, the inputs and the search space is very important. To help the code generation without problems of variables’ name changes,we use style of ”explanation – objective”, also we have written the situation about what the agent may face, hoping help the agent get an unsupervised reward or reflection from the prompts’ observation on situation.

```
system_prompts:
You are a smart linguist who only need decompose the the contents according to user requirements\

user_prompts:
Let's think step by step\
For example \
the question:"Given the lengths of the four sides of the quadrilateral in an array, judge whether the quadrilateral is Cyclic"\
the answer will be :\
"1.Assume the function is called "is_cyclic_quadrilateral"\
2.It takes one parameter, "sides", which is an array containing the lengths of the four sides of a quadrilateral.\
3.The purpose of the function is to determine whether the quadrilateral is cyclic or not.\
4.A cyclic quadrilateral is a quadrilateral that can be inscribed in a circle, meaning that all four vertices of the quadrilateral lie on the circumference of a circle.\
5.To determine if a quadrilateral is cyclic, we need to check if the sum of opposite angles is equal to 180 degrees.\
6.If the sum of opposite angles is equal to 180 degrees, then the quadrilateral is cyclic.\
7.If the sum of opposite angles is not equal to 180 degrees, then the quadrilateral is not cyclic.\
8.The function should return a boolean value, True if the quadrilateral is cyclic, and False if it is not."\
Now the question is :{summary_prompt}\
please give me the answer in the similar way\
```

Fig. 4. The prompt of Auto decomposition

Secondly, after getting the multi-step, we can combine it with summary prompt. Traditional CoT may use language style of ”one step ,one explanation, one outputs”. This pattern may help agent to react on true environments to get a more actual advice or reflection. Contrarily, like the figure

```
system_prompts:
You are now a smart programmer who needs to understand the
instructions and outputs the correct code in Python mode according to
user requirements\

user_prompts:
Let's think step by step and answer the question like the example:\
Q:Enter an integer array (list), and the function outputs the closest
integer to the average of the array (rounded down if equidistant).For
instance,inputs [1,2,3,4.1] outputs 3\
To help generate the code, the steps are given\
1.The function is called "find_closest_integer".\
2.It takes one parameter, "array", which is an integer array (list)\.
3.The purpose of the function is to find the closest integer to the
average of the array.\
4.To find the average of the array, we sum up all the elements and
divide it by the length of the array.\
5.We then find the closest integer to the average by comparing the
absolute difference between each element and the average.\
6.If there are multiple integers equidistant from the average, we
round down to the lower integer.\
7.The function should return the closest integer to the average.\
8.If the array is empty, the function should return None.\
A: "explanation":'Firstly,we can know that the value is a list with
variables. As we need to find the averages, we need to know the
function average=sum substract the number of variables\
We can use 'len(inputs)' to get the number of variables.To get the
sum ,we can use (for i in inputs)to get variables and save it with
s,like s=s+i\
After we get the average we can use '\\' to get the range and compare
the distance between average\\1 and average\\1+1 \
Finally , we can get the answer:\'
"code":' def closest_integer(value):\
       s=0\
       for i in value:\
           s=s+i\
       average=s/len(value)\
       if (average\\1-average)^2 > (average\\1+1-average)^2:\
            interger=average\\1+1\
       else:\
            interger=average\\1\
       return interger '\

I want you to write only  a function or a class with some functions
in python code and please follow this rule all the time\
the task is {summary_prompt}\
To help you have a better understanding we have the following
instance to understand how it works:{example}\
To help you generate the code, i will give you each step\
the steps are :{step}\
```

Fig. 5. The prompt of code generation with chain of thought and decomposition

5, we use language style of "summary,instance,complete steps,complete explanation and code" owing to the nature of CoT is LLMs Thinking and reasoning. The advantage is able to get a higher correlation natural language output as the prompts are logistic with huge information compared with small steps.

Thirdly,in order to get the code, we load the output as Json patterns to get the string. However, sometimes the agent may generates code involved pattern problems or inputs colon,which will bother the results.To avoid this ,we write the Fig.6. prompts. What you need to pay attention is that you must add "You mustn't copy the code about the question example".We will explain this reason in next chapter.

In executor mode, we has prepared five methods combination: Simple summary prompts, summary prompts with LLMs' decomposition ,summary prompts with manual decomposition, summary prompts with LLMs' decomposition and COT, summary prompts with manual decomposition and

```
system_prompts:
You are now a smart programmer who needs to
understand the instructions and outputs the correct
code in Python mode according to user requirements\

user_prompts:
Here is an example :\
Q:"code": "def sum(m,n)\n    k = m+n    return k"\
"explanation":"The code defines a function called
'sum' that takes m,n as input.  Finally, it returns
the sum of m+n"\
A:"def sum(m,n)\n    k = m+n    return k"\

You need to copy the content of code from dictionary
{code}\
You mustn't copy the code about:"def sum(m,n)\n    k
= m+n  return k"\
Make sure that the output is clear only with code
without any explanations\
```

Fig. 6. The prompt of Code extraction

COT to implement different control groups, mainly verifying three questions: 1. Whether to automatically decompose and combine input with potential information and a few problem words can improve code accuracy. 2. For input languages that no longer have potential information, HOW is the impact of decomposition and CoT . 3. Whether adding a chain of thought is effective.

IV. Experiment

A. The Verification of Method Feasibility


```
"id66": {
    "example": "input: {'A': [[1, 3], [2, 2]], 'K': 1}\n output: [[2, 2]]",
    "summary_prompt": ["Find the k nearest points to the origin."],
      "prompts": ["Assign the list of points {A} to a variable named my_points.",
"Assign the integer {K} to a variable named k.", "Implement a function that computes
the distance between a point and the origin (0,0).", "Implement a function that
computes the k closest points in an array to the origin and store as result.",
"Compute the k closest points in my_points and print them out."],
    "test": "def check(candidate):\n    assert candidate([[1, 3], [2, 2]], 1) == [[2, 2]]",
    "mark": "True"
  }
```


Fig. 7. The datasets of EMTPB: id66

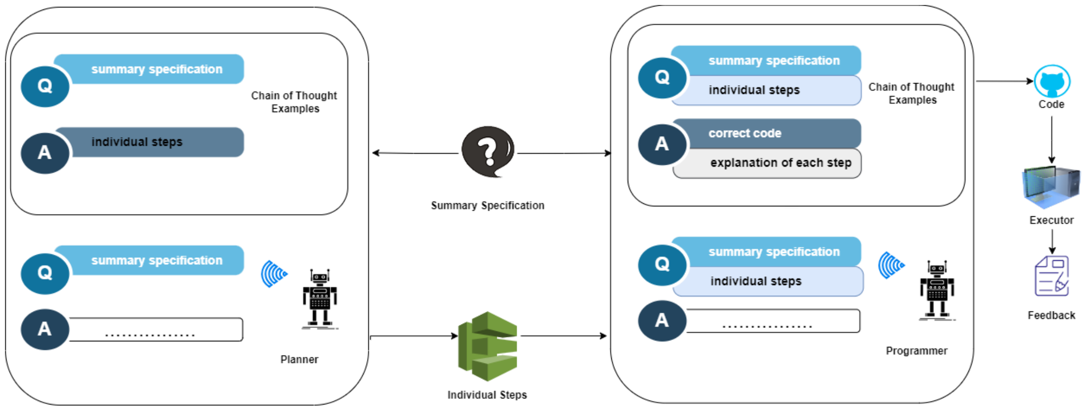


Fig. 8. The flow path of Auto-D with CoT

INPUTS:

system_prompts:
You are a smart linguist who only need decompose the the contents according to user requirements\
user_prompts:
Let's think step by step\
For example \
the question:"Given the lengths of the four sides of the quadrilateral in an array, judge whether the quadrilateral is Cyclic"\
the answer will be :\
"1.Assume the function is called "is_cyclic_quadrilateral"\
2.It takes one parameter, "sides", which is an array containing the lengths of the four sides of a quadrilateral.\
3.The purpose of the function is to determine whether the quadrilateral is cyclic or not.\
4.A cyclic quadrilateral is a quadrilateral that can be inscribed in a circle, meaning that all four vertices of the quadrilateral lie on the circumference of a circle.\
5.To determine if a quadrilateral is cyclic, we need to check if the sum of opposite angles is equal to 180 degrees.\
6.If the sum of opposite angles is equal to 180 degrees, then the quadrilateral is cyclic.\
7.If the sum of opposite angles is not equal to 180 degrees, then the quadrilateral is not cyclic.\
8.The function should return a boolean value, True if the quadrilateral is cyclic, and False if it is not."\
Now the question is :{summary_prompt}\
please give me the answer in the similar way\

OUTPUTS:

multi_step=['####', ' Here is the answer to the question "Find the k nearest points to the origin":', '', ' 1. The function is called "find_k_nearest_points".', ' 2. It takes two parameters, "points" and "k". "points" is an array containing the coordinates of multiple points in the form (x, y), and "k" is the number of nearest points to find.', ' 3. The purpose of the function is to find the k nearest points to the origin (0, 0).', ' 4. To find the distance between each point and the origin, we can use the Euclidean distance formula: distance = sqrt(x^2 + y^2).', ' 5. We can calculate the distance for each point and store it in a separate array.', ' 6. Then, we can sort the array of distances in ascending order.', ' 7. Finally, we can return the first k points from the sorted array as the k nearest points to the origin.', ' 8. The function should return an array containing the k nearest points.', '', ' Example usage:', ' points = [(1, 2), (3, 4), (5, 6), (7, 8)]', ' k = 2', ' nearest_points = find_k_nearest_points(points, k)', ' print(nearest_points) # Output: [(1, 2), (3, 4)]']

INPUTS:

system_prompts:
You are a smart languager who only need decompose the the contents according to user requirements\
user_prompts:
I want you to write a python code about define a function or class \
The task is {summary_prompt}\
To help you have a better understanding we have the following instance to understand how it works:{example}\

Auto-D:To help you generate the code, i will give you each step\
The steps are :{multi_step}\

God-D:To help you generate the code, i will give you each step\
The steps are :{prompts}\

OUTPUT:

Baseline:

code_json={"code": "import math\n\n def find_k_nearest_points(points, k):\n distances = []\n for point in points:\n distance = math.sqrt(point[0]**2 + point[1]**2)\n distances.append(distance)\n sorted_points = [point for _, point in sorted(zip(distances, points))]\n return sorted_points[:k]",
"explanation": "Define a function find_k_nearest_points that takes a list of points and an integer k as input. Calculate the distance of each point from the origin using the Euclidean formula. Sort the points based on their distances. Return the k nearest points."}

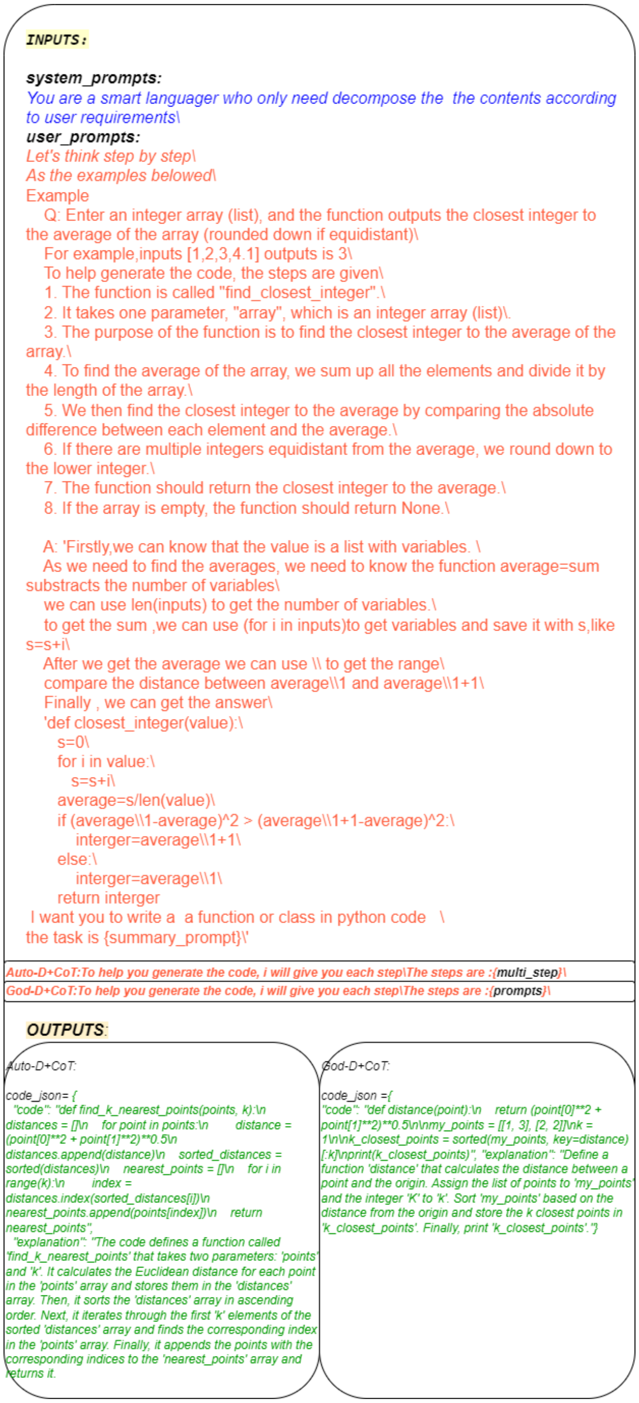
God-D:

code_json= {
"code": "def distance(point):\n return (point[0]**2 + point[1]**2)**0.5\n\n\ndef k_closest_points(points, k):\n points.sort(key=distance)\n return points[:k]\n\n\nmy_points = [[1, 3], [2, 2]]\nk = 1\n\n
result = k_closest_points(my_points, k)\nprint(result)",
"explanation": "Define a function 'distance' that calculates the distance between a point and the origin using the Euclidean formula. Define another function 'k_closest_points' that sorts the points based on their distance from the origin and returns the k closest points. Assign the list of points to 'my_points' and the value of k to 'k'. Compute the k closest points using 'k_closest_points' and store the result. Finally, print the result."

Auto-D:

code_json={{"code": "def find_k_nearest_points(points, k):\n distances = []\n for point in points:\n distance = (point[0]**2 + point[1]**2)**0.5\n distances.append(distance)\n sorted_distances = sorted(distances)\n nearest_points = []\n for i in range(k):\n index = distances.index(sorted_distances[i])\n nearest_points.append(points[index])\n return nearest_points",
"explanation": "Here is the answer to the question 'Find the k nearest points to the origin':\n\n1. The function is called 'find_k_nearest_points'.\n2. It takes two parameters, 'points' and 'k'. 'points' is an array containing the coordinates of multiple points in the form (x, y), and 'k' is the number of nearest points to find.\n3. The purpose of the function is to find the k nearest points to the origin (0, 0).\n4. To find the distance between each point and the origin, we can use the Euclidean distance formula: distance = sqrt(x^2 + y^2).\n5. We can calculate the distance for each point and store it in a separate array.\n6. Then, we can sort the array of distances in ascending order.\n7. Finally, we can return the first k points from the sorted array as the k nearest points to the origin.\n8. The function should return an array containing the k nearest points.\n\nExample usage:\npoints = [(1, 2), (3, 4), (5, 6), (7, 8)]\nk = 2\nnearest_points = find_k_nearest_points(points, k)\nprint(nearest_points) # Output: [(1, 2), (3, 4)]"}}

The Figure 7 introduce the basic information of EMTPB: id66. And these two figures show the process of string generation to prove that different prompts can truly change the outputs and achieve the style we expected to get. In output,we will get a string of json format to reach a better classification betwwen code and explantion.



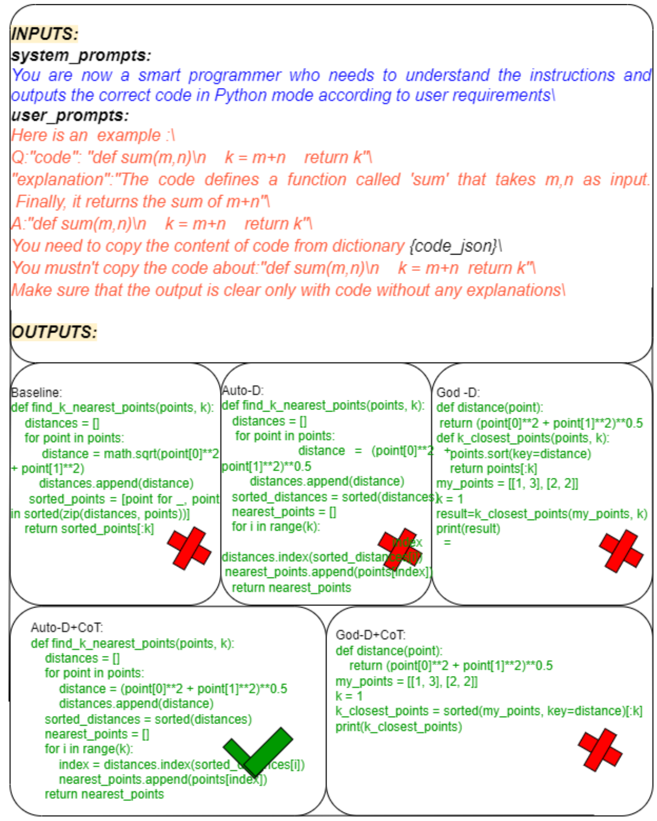
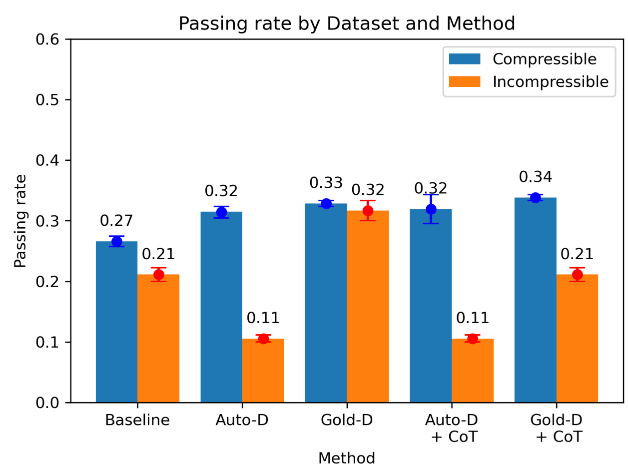


Fig. 9. The five results of EMTPB:id66

B. The Results on EMTPB



This figure is CoT using in Auto-D and God-D . We can get code by tradition code like ”json.load()”. However, sometimes we may face problems if the string combines lots of signals that cannot be recognized. The Figure 9 shows its function which can get the code in string pattern. It is worth noting that this prompt does not generate new code through second iteration to avoid the compassion on five ways and reduce api token using. The pictures shown prove this view. From the result ,we can see the Auto-D+CoT truly can solve some problems that others cannot solve.

This figure shows the accuracy Passing rate of different methods based on the EMTPB. The Baseline, Auto-D, God-D, Auto-D+CoT and God-D+CoT respectively represents simple summary prompts, summary prompts with LLMs' decomposition , summary prompts with manual decomposition, summary prompts with LLMs' decomposition and COT, summary prompts with manual decomposition and COT.

The experiment is done by one epoch with LLMs GPT-3.5 at temperature 0. From the figure in elements of mark:True EMTPB ,we can ensure that 1 the COT using in Program analysis can truly improve the accuracy,2 compared with manual notation,the introduction of auto decomposition can reach nearly similar performance and prior to the baseline.

It is worth notating that the experiment need do more times in different temperature as the LLMs code generation exists randomness . Also ,we haven't overcame the problem if the prompts cannot sufficiently summarized by human.

C. The Additional Findings

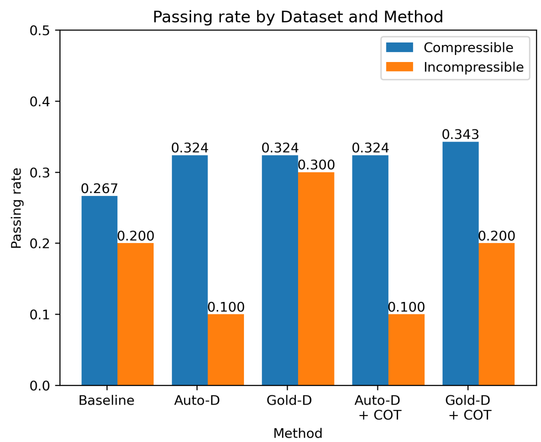


Fig. 10. The accuracy Pass@1 of five methods based on the EMTPB without the prompt"You must not copy the code about the question example"

In this part ,we will explain the measure in III.B about why you must add "You mustn't copy the code about the question example". The following figure is about the code in the experiment of one epoch with prompt added,and we find that only id23 and id87 appear the phenomena of coping the example in Fig.6 Code extraction. Without using it ,the accuracy is shown in the following figure. We haven't find the reason about some doesn't appear this phenomena and the key factors,but by this way you can reduce this condition. Our findings show in some condition,the LLMs prefer to just copy the example without thinking and reasoning.

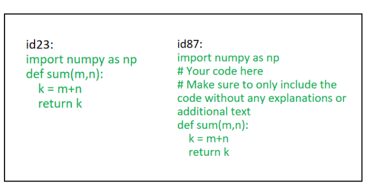


Fig. 11. The content about what code the agent has copied in id23 and id87

V. CONCLUSIONS

Through this research, we propose a multi-turn program synthesis method that directly helps the model understand user intentions. Our approach not only improves the accuracy of the model but also provides convenience for interpreting and troubleshooting model behavior. We believe that this approach has broad prospects for further research and application. However,we admit that the prompt we provide has many details to revise in order to achieve higher accuracy in MTPB. Also,we haven't tested and verified that our idea has a better performance on humaneval.For future thinking ,we propose questions why the code answer is only one and whether the large language model can generate multiple codes which are all corresponding to the prompt. We speculate that this model may have more thinking and reasoning abilities.

REFERENCES

[1] Nijkamp E, Pang B, Hayashi H, et al. Codegen: An open large language model for code with multi-turn program synthesis[J]. arXiv preprint arXiv:2203.13474, 2022.
[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, L ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Advances in Neural Information Processing Systems, volume 30, 2017.
[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.
[4] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 7871–7880, 2020.
[5] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In ICLR, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.
[6] Gunasekar S, Zhang Y, Aneja J, et al. Textbooks Are All You Need[J]. arXiv preprint arXiv:2306.11644, 2023.
[7] Shinn N, Labash B, Gopinath A. Reflexion: an autonomous agent with dynamic memory and self-reflection[J]. arXiv preprint arXiv:2303.11366, 2023.
[8] Chen X, Lin M, Schärli N, et al. Teaching large language models to self-debug[J]. arXiv preprint arXiv:2304.05128, 2023.
[9] Athiwaratkun B, Gouda S K, Wang Z, et al. Multi-lingual evaluation of code generation models[J]. arXiv preprint arXiv:2210.14868, 2022.
[10] Wei J, Wang X, Schuurmans D, et al. Chain-of-thought prompting elicits reasoning in large language models[J]. Advances in Neural Information Processing Systems, 2022, 35: 24824-24837.
[11] Kojima T, Gu S S, Reid M, et al. Large language models are zero-shot reasoners[J]. Advances in neural information processing systems, 2022, 35: 22199-22213.
[12] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.
[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.
[14] Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis. In ICLR (Poster), 2017. URL https://openreview.net/forum?id=rJ0JwFcex.
[15] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. ACM SIGARCH Computer Architecture News, 41(1):305–316, 2013.
[16] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. ACM Sigplan Notices, 46(1):317–330, 2011.
[17] Oleksandr Polozov and Sumit Gulwani. Flashmeta: A framework for inductive program synthesis. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 107–126, 2015.

[18] Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling,J., Gimeno, F., Dal Lago, A., et al. (2022). Competition-level code generation with alphacode. Science, 378(6624):1092–1097.

[19] Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.-G., and Chen, W. (2022). Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397.

[20] Le, H., Wang, Y., Gotmare, A. D., Savarese, S., and Hoi, S. C. H. (2022). Coderl: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35:21314–21328.

[21] Austin J, Odena A, Nye M, et al. Program synthesis with large language models[J]. arXiv preprint arXiv:2108.07732, 2021.

Appendix

Firstly, We 'd like to thanks for the instruction of Chunchuan lyu. For nearly three months research , we try to contributes something in this area, though the idea is just a physical combination without some huge innovation.I'm not sure whether other people can wittiness this report or will continue to do some research in this area, we hope this report can do some what we can do with limited time and costs. Also, thanks for the company Deeplearning.Ai to open the free source of Openai Gpt-3.5 api use .Code: https://github.com/nakupenda-c/Auto-Prompt-Decomposition-with-Chain-of-Thought-in-Large-Language-Models-for-Program-Synthesis