ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #44863

A Large Language Model Pipeline to Automate the Solution of Competitive Programming Problems

Mr. Devang Jayachandran, Pennsylvania State University, Harrisburg, The Capital College

Devang Jayachandran is currently a graduate student pursuing a Masters of Science in Computer Science at the Mathematics and Computer Science department in Penn State Harrisburg. Devang received his Bachelor's of Engineering in Information Science from the National Institute of Engineering, Mysuru, India and then worked at JP Morgan Chase & Co, Bengaluru, India in the field of Natural Language Processing and Document Extraction.

Dr. Jeremy Joseph Blum, Pennsylvania State University, Harrisburg, The Capital College

Dr. Jeremy Blum is an associate professor of Computer Science at the Pennsylvania State University, Harrisburg, PA, USA. Prior to joining Penn State Harrisburg, Dr. Blum worked as a research scientist at the Center for Intelligent Systems Research at the George Washington University. Dr. Blum received a D.Sc. in Computer Science and an M.S. in Computational Sciences, both from the George Washington University, as well as a B.A. in Economics from Washington University. His research interests include computer science education and transportation safety.

©American Society for Engineering Education, 2024

A Large Language Model Pipeline to Automate the Solution of Competitive Programming Problems

Devang Jayachandran and Jeremy J. Blum
The Pennsylvania State University, Harrisburg

Abstract

The recent rapid introduction of large language models has enabled new black box approaches to optimize the application of these models for various scenarios. GPT-4 is a multimodal large language model introduced by OpenAI which can answer complex questions, analyze nuanced data, and solve complicated programming problems. The performance of GPT is dependent upon the provided prompt and hyperparameters. This paper explores the effect of minor variations in system prompt and parameters including temperature and top-p for code generation and code accuracy for competitive programming tasks. Temperature controls the amount of randomness in the response, with a temperature of zero producing deterministic output. Top-p controls the cumulative probability distribution for tokens considered for the next output token. Based on the results, we propose approaches to optimize system prompts for code-generation and parameter values to improve the correctness of code. In addition, we propose a pipeline that utilizes these enhancements to effectively solve algorithmic puzzles common in computer science education, in addition to complex contest programming problems.

Keywords

Generative Artificial Intelligence, Code Generation Pipelines, Contest Programming.

Introduction

Programming contests are competitions in which participants attempt to write computer programs that solve algorithmic puzzles. Past studies have identified a range of pedagogical benefits for student participation in these contests, including enhancing learning outcomes by deepening conceptual comprehension and fostering team collaboration, along with equipping students for technical job interviews [1-2]. These benefits, notwithstanding, a number of hurdles exist to expanding participation in these contests [3].

Artificial Intelligence (AI) tools, like ChatGPT, have been found to lower barriers to participation in contest programming [4]. Generative AI tools can provide scaffolding for participants that allow participants to succeed, even if they have limited computer science background and familiarity with contest programming problems. This scaffolding includes the ability of these systems to provide template code that partially solves a problem, which participants can use as a basis for a solution. These systems also provide the ability to help participants debug their programs when they contain errors. Despite these potential benefits, there is limited guidance on how participants can leverage these systems effectively to solve contest problems.

© American Society for Engineering Education, 2024

This work analyzes prompt engineering approaches that can improve GPT-4's code-generation accuracy and randomness with changes to the model's temperature and system prompts. We assess the optimal temperature for GPT-4 to better enable its code-generation capabilities and observe the change in code correctness based on the personality provided to GPT-4. We propose a pipeline built upon GPT-4 that can generate accurate python code for complex questions in competitive programming contests. The result is a robust framework that utilizes dynamic system prompts to generate code and modify the solution. It includes a feedback loop with GPT-4 that will iteratively remove syntactical errors present in the generated code. In addition, we explore its capabilities in generating sample test cases to check the validity of the generated output code.

The pipeline was evaluated against various high level programming problems. In addition, we compared the code generated by the pipeline against the code generated in a naive approach without the use of system prompts. The pipeline performs significantly better than available approaches. An analysis of the effect of temperature and system prompt in code correctness was used to arrive at optimal values for these parameters.

This exploration of the capabilities and limitations of GPT-4 in the area of code generation and problem processing provides insight into the role that generative AI plays in programming education and competitions. This pipeline helps to provide an accessible entry point to competitive programming contests for students that are inexperienced in the field of coding in two key ways. First, a similar pipeline could be integrated into programming classes. Moreover, the strategies utilized by the pipeline could be taught to students to help them interact more effectively with these tools.

Literature Review

Chatbots have seen major advances in the last few years. They are now able to mimic human agents during conversation due, in large part, to advances in large language models [5].

GPT is an example of such a large language model, created by OpenAI. OpenAI claims that it can "answer follow up questions, admit its mistakes, challenge incorrect premises, and reject inappropriate requests" [6]. GPT-4, the latest iteration of GPT "outperforms both previous large language models and most state-of-the-art systems in a variety of NLP and MMLU benchmarks" [7]. Independent evaluations of ChatGPT support its ability to generate high quality output over a wide range of tasks [8].

There are other solutions like Codex, which is designed to work as a programming companion [9]. The authors conclude that "Copilot is able to generate correct and optimal solutions for some fundamental problems in algorithm design. However, the quality of the generated code depends greatly on the conciseness and depth of the prompt that is provided by the developer."

ChatGPT's performance can be significantly affected by the framework's design. Prompt patterns can be used to significantly increase its capabilities [10]. Previous frameworks have used GPT-4 for tasks as wide ranging as medical text de-identification [11] and SQL query generation [12].

© American Society for Engineering Education, 2024

Even within the context of direct interaction with the generative AI, the performance of the system is affected by a range of prompting strategies and system parameters. Providing test cases with the problem description has been shown to improve generative AI correctness when applying these systems to coding tasks [13]. In addition, summarizing the task to provide a more concise prompt has been shown to improve system output in business management applications [14]. Asserting a personality or goals for the system can also have a dramatic effect on the quality of system output [15].

In terms of system parameters, these models have multiple settings that control the amount of randomness in their output [16]. This randomness is further compounded with the inherent non-determinism of the GPT-4 API [17]. Higher settings create a larger variety of possible responses for a single prompt, which has been characterized as increasing the "creativity" of these models. On the other hand, the increasing randomness can reduce the likelihood of a correct system response.

Methodology

Multiple pipelines have been created to generate code for various use cases since OpenAI exposed the GPT model and API. It is also evident that standalone ChatGPT needs additional clarification to arrive at optimal solutions. In this paper we aim to create a pipeline that uses custom prompts and dynamic prompt generation for the tailored use case of generating python code for contest programming problems.

We also analyze the effect of GPT-4 model parameters including temperature, top-p, and personality with respect to accuracy of the generated code. Temperature controls the randomness present in the model. Lowering this value allows the model to become deterministic while increasing it allows for more creativity. For code generation we hypothesize the use of a lower value since we usually want the same code to be generated for every iteration. The GPT model can use a system prompt containing personalities to enhance its capabilities for specific use cases Additionally, we try to enforce consistent parse-able output formatting via few-shot learning. Providing personalities with expertise in sub-fields along with examples should allow GPT to generate better outputs for specific inputs.

Pipeline Architecture

The pipeline proposed in Figure 1 is designed to accept the text input of a contest programming problem and respond with an easy-to-understand explanation of the problem and a working python code for solving the problem. Each block takes text data as input and transforms into an output based on its specified system and user prompt. All blocks in the pipeline use the "gpt-4-1106-preview" model by OpenAI to generate the outputs. As shown in the figure, the pipeline contains five main blocks.

© American Society for Engineering Education, 2024

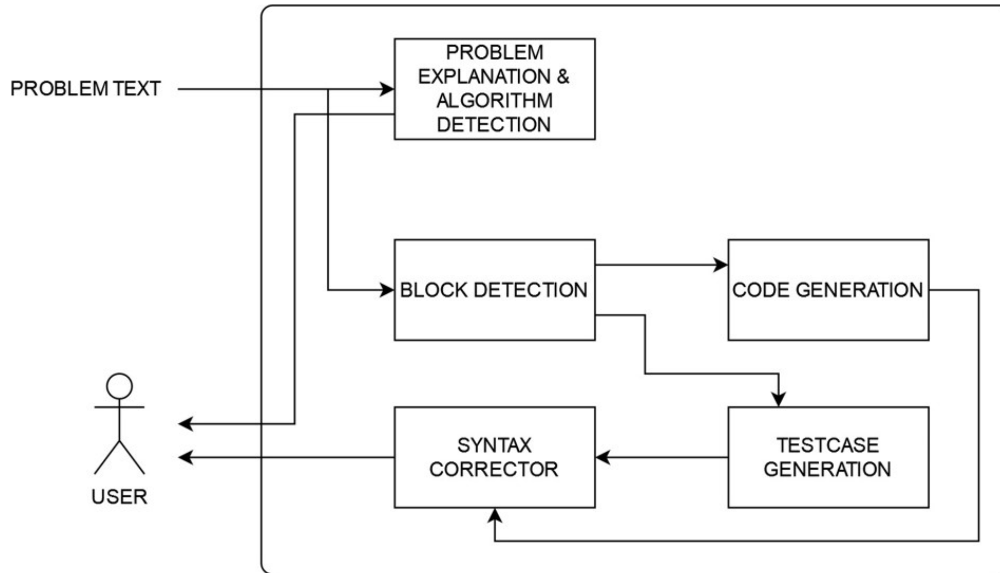

Figure 1: Pipeline Architecture

The *Block Detection* component detects the required blocks present in the problem. As in the example in Figure 2, these blocks typically include "question", "input format", "output format", and "constraints." This block then returns a Javascript Object Notation (JSON) formatted string which encodes these sections.

Write a program to respond to the different prompts in the Penn State "WE ARE" call and response.

Input Format

The first line of input will contain an integer *n*.

The next *n* lines will contain a call from the "WE ARE" call and response.

Constraints

$1 \leq n \leq 4$

Output Format

Output *n* lines with the appropriate response for each of the *n* lines of input. Note that the response should consist only of uppercase letters and spaces. You should omit any punctuation.

Figure 2: A sample problem with blocks identified by the Block Detection Component

The *Problem Explanation & Algorithm Detection* component provides the problem description to the large language model to generate an easy-to-understand explanation of the problem to pair with the code generated by the pipeline. The prompt that is provided to create this explanation gives special emphasis in the prompt on getting a response with algorithms that could improve the code in terms of time complexity. Additionally, we specify in prompt not to generate a code solution.

© American Society for Engineering Education, 2024

The *Code Generation* component provides the large language model the encoded question, its input and output format, and the required constraints to generate a python program whose goal is to solve the problem. The behavior of this component changes according to the system prompt that it is supplied with and the temperature and top-p specified for the model.

The *Testcase Generator* component uses the language model to create additional testcases using the input and output formats that were extracted from the problem. Currently these testcases are syntactically viable for the given problem and need not be logically viable.

Finally, the *Syntax Corrector* component receives the generated code from the previous block and iteratively removes any syntactical errors present in the code that stop its execution. It uses sample test cases that conform to the specified input and output format to ensure that the code follows the required formatting. Syntax correction lets us remove the syntax errors that might be caused by model hallucinations or incorrect formatting. The corrected code is then provided to the user.

System Prompts

There are two distinct types of prompts provided in the GPT API. The first is a system prompt that provides a background for the task. The second is a user prompt with the specific problem to be solved.

We created a set of system prompts used for code generation to test the importance of subtle changes in system prompts for code generation. The prompt is based on the Wolverine system that uses GPT-4 to debug Python scripts [18]. Variations in the prompt included the personality for the prompt, whether language was specified, and whether example input and output was provided.

Three distinct personalities were tested, amateur, default, and expert. These personalities were generated based on data repositories [19-20] that curated prompts for various scenarios. For the amateur personality, the prompt began with the instruction: `You are part of an automated program code generator`. The default personality used the description: `You are part of a personal AI code generator`. The expert personality used the description: *`You are part of an elite automated program generating team.`*

The target language was specified in some tests. To do this, the language "Python" was inserted into the first sentence of the instruction. For example, the expert personality with the language specified would be: *`You are part of an elite automated`* ***`python`*** *`program generating team.`*

The pipeline required a predictable output format for the code generation block. We tested the ability of the large language model to use one-shot learning by providing, as part of the prompt, an example of the input and output that would be provided as part of the input prompt. For example, if formatting was specified, the prompt would include the following text:

© American Society for Engineering Education, 2024

```
An example input is
[
    {"question": "In this challenge you must find the sum of two integers."},
    {"input_format": "two space separated integers"},
    {"output_format": "single integer"},
    {"constraints": ["1<=a<=100","1<=b<=100"]}
]
An example output is
{
    "code":"a, b = map(int, input().split())\nprint(a + b)"
}
```

Results

Pipeline Parameterization

A range of instantiations of this architecture were evaluated to explore the impact of system prompts and parameters on the pipeline effectiveness. A total of 25 different combinations of system prompts and temperature were run on questions from the IEEEXtreme 16 and IEEEXtreme 17 dataset.

As shown in Figure 3, system prompts containing the expert personality performed better than the other ones. Simply by instructing the model that it is an expert in python programming provided a boost of 5.3% in the number of testcases passed than the next best personality.

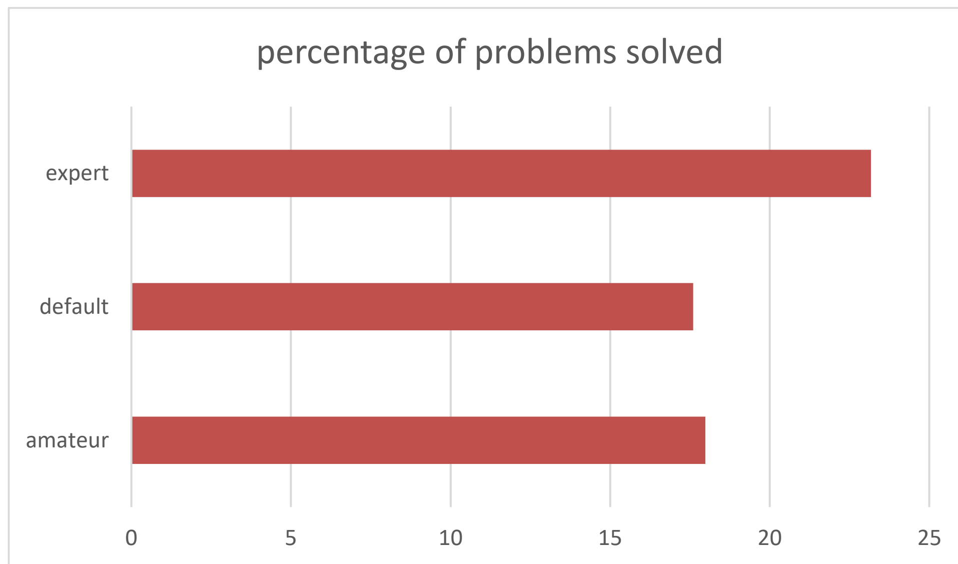

Figure 3: Average percentage of testcases solved as a function of personality

Temperature controls the randomness in the response of the large language model. High temperatures provide more variety and “creativity” in the response, while low values provide more consistent output. As shown in Figure 4, reducing the temperature of the model to 0.4 from

© American Society for Engineering Education, 2024

a default value of 1.0 helps in preserving code integrity and correctness across multiple runs and increases the percentage of testcases solved by 1.47%.

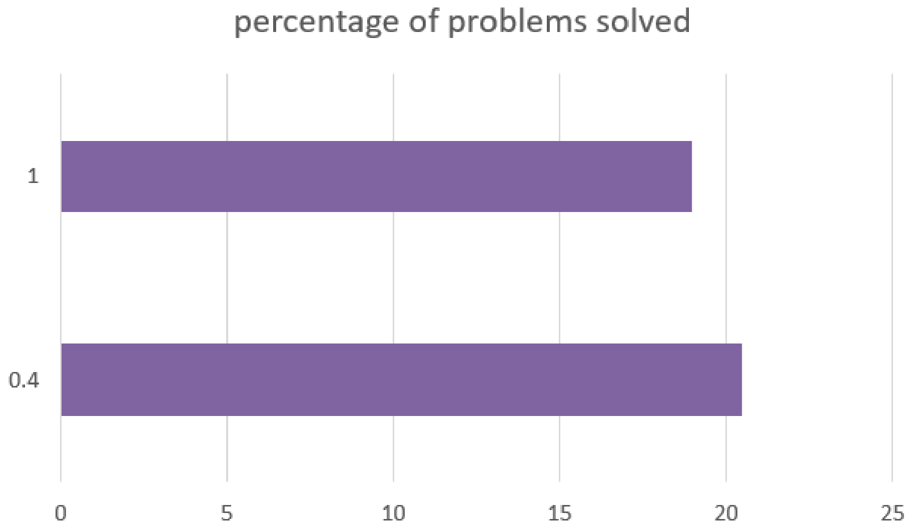

Figure 4: Average percentage of testcases solved as a function of temperature

Additionally, Figure 5 shows that providing the language to solve the problem in, namely python, results in the pipeline solving 2.5% more of the testcases. For example, explicitly mentioning the language, as below, provided an improvement over simply omitting it.

```
You are part of an elite automated python program generating team.
```

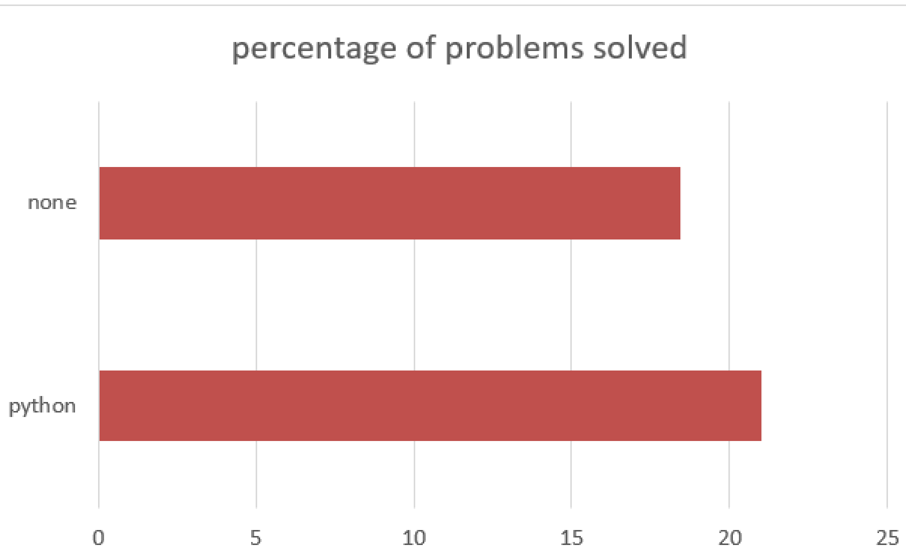

Figure 5: Average percentage of testcases solved when target language is not specified or when it is specified as Python

© American Society for Engineering Education, 2024

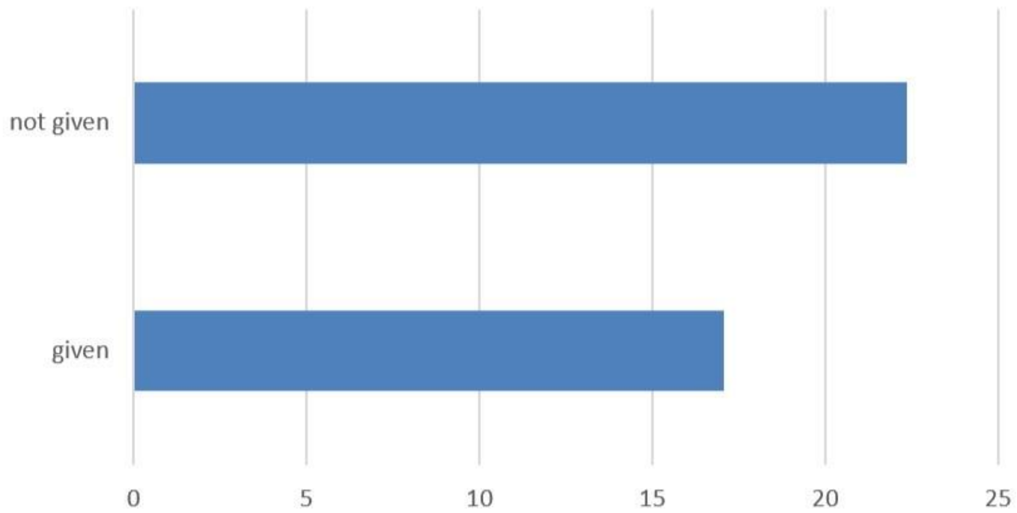

Figure 6: Average percentage of testcases solved when target format examples are given or not

Not all the system prompt options were found to improve pipeline performance as shown in Figure 6. The use of few-shot learning in the form of an input and output example in the system prompts did help in standardizing the received output. However, it also led to a decrease in the number of testcases solved by 5.3%. The pipeline performed better when no examples were given with respect to formatting.

© American Society for Engineering Education, 2024

Pipeline Performance

After finalizing the pipeline design by choosing the best version of the system prompts, the pipeline was evaluated against ChatGPT using the GPT 3.5 large language model and GPT 4.0 API. These systems were asked to solve twenty-five problems from IEEEXtreme 16 and 17 problem set as well as the questions from a generative AI-assisted competition held locally. These competitions were chosen to ensure that the test set included problems that the language model would not have been exposed to during the model's training.

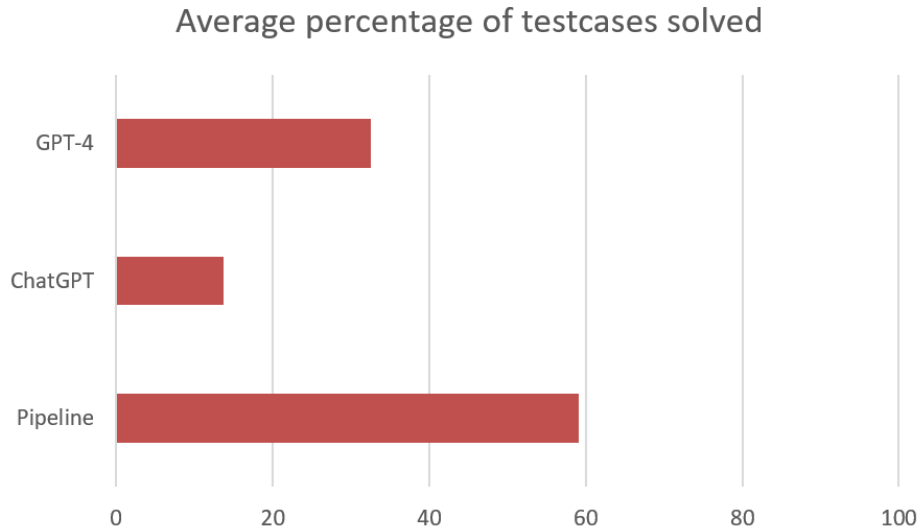

Figure 7: Average percentage of testcases solved for the pipeline versus baseline models

As shown in Figure 7, the pipeline performs significantly better than both baseline models. The pipeline solved close to 60% of the testcases on average and is even able to solve 100% of the testcases for some problems that were not able to be solved by other models [1]. The pipeline had an overall testcase resolution of 59.11% as compared to 13.72% of testcases resolved by ChatGPT and 32.53% of testcases resolved by GPT-4. Additionally, the pipeline solved three of the problems with no failed testcases and one problem with a single failed testcase whereas both GPT-4 and ChatGPT were only able to solve one problem with no failed testcases. There were three problems where both ChatGPT and GPT-4 were unable to provide a viable solution, but the pipeline was able to solve these problems partially.

Conclusions and Future Work

GPT models and their implementations have been rising in popularity and there has been an increase in their various implementations. We designed one such pipeline that solves complex contest programming problems in python. We also analyzed the effect of change in temperature and system prompts in ensuring correct and consistent outputs. We further compared the accuracy of the pipeline versus other available approaches. The pipeline performs significantly better than existing approaches, namely, directly using ChatGPT or the GPT-4 API.

© American Society for Engineering Education, 2024

Overall, the proposed pipeline reduces the interaction required with GPT to arrive at a workable solution. The pipeline is also able to remove hallucinations generated by GPT and always provides the required input and output formatting.

In future work, we intend to analyze the ability of this pipeline helps to provide an accessible entry point to competitive programming contests. We plan to explore the use of a similar pipeline that could be integrated into programming classes. Moreover, we plan to introduce students to the prompting strategies and parameterization utilized by the pipeline and evaluate whether this background can improve their effectiveness in using generative AI for coding tasks.

Acknowledgments

To be included in the de-anonymized version of the paper.

References

1 A. Bloomfield, B. Sotomayor. 2016. A programming contest strategy guide. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE '16). Association for Computing Machinery, New York, NY, USA, 609–614. DOI: https://doi.org/10.1145/2839509.2844632

2 M. Armoni 2011. CS contests for students: why and how? ACM Inroads 2, 2 (June 2011), 22–23. DOI: https://doi.org/10.1145/1963533.1963540

3 J.J. Blum. 2023. Competitive Programming Participation Rates: An Examination of Trends in U.S. ICPC Regional Contests. Discover Education, 2(11), 12.

4 D. Jayachandran, P. Maldikar, T. S. Love, J.J. Blum. 2024. “Leveraging Generative Artificial Intelligence to Broaden Participation in Computer Science,” to appear at the Association for the Advancement of Artificial Intelligence Symposium on Increasing Diversity in AI Education and Research, March 25-27, Stanford University, 1-6.

5 E. Adamopoulou and L. Moussiades. 2020. “Chatbots: History, technology, and applications,” Mach. Learn. Appl., vol. 2, p. 100006, Dec. 2020, doi: 10.1016/j.mlwa.2020.100006.

6 “Introducing ChatGPT.” Accessed: Jan. 12, 2024. [Online]. Available: https://openai.com/blog/chatgpt

7 OpenAI et al. 2023. “GPT-4 Technical Report.” arXiv, Dec. 18, 2023. doi: 10.48550/arXiv.2303.08774.

8 J. Kocoń et al., 2023. “ChatGPT: Jack of all trades, master of none,” Inf. Fusion, vol. 99, p. 101861, Nov. 2023, doi: 10.1016/j.inffus.2023.101861.

9 A. M. Dakhel et al., 2023. “GitHub Copilot AI pair programmer: Asset or Liability?” arXiv, Apr. 14, 2023. doi: 10.48550/arXiv.2206.15331.

10 J. White et al., 2023. “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT.” arXiv, Feb. 21, 2023. Accessed: Oct. 03, 2023. [Online]. Available: http://arxiv.org/abs/2302.11382

11 Z. Liu et al. 2023. “DeID-GPT: Zero-shot Medical Text De-Identification by GPT-4.” arXiv, Mar. 20, 2023. Accessed: Sep. 01, 2023. [Online]. Available: http://arxiv.org/abs/2303.11032

12 I. Trummer. 2023. Demonstrating GPT-DB: Generating Query-Specific and Customizable Code for SQL Processing with GPT-4. Proc. VLDB Endow. 16, 12 (August 2023), 4098–4101. https://doi.org/10.14778/3611540.3611630

13 L. Murr, M. Grainger, and D. Gao. 2023. Testing LLMs on Code Generation with Varying Levels of Prompt Specificity. arXiv preprint arXiv:2311.07599.

14 K. Busch, A. Rochlitzer, D. Sola, and H. Leopold. 2023. Just tell me: Prompt engineering in business process management. In Proceedings of the In-ternational Conference on Business Process Modeling, Development and Support (pp. 3-11). Cham: Springer Nature Switzerland. doi.org/10.1007/978-3-031-34241-7_1

15 G. Marvin, N. Hellen, D. Jjingo, and J. Nakatumba-Nabende. 2024. Prompt Engineering in Large Language Models. In: Jacob, I.J., Piramuthu, S., Falkow-ski-Gilski, P. (eds) Data Intelligence and Cognitive Informatics. ICDICI 2023. Algorithms for Intelligent Systems. Springer, Singapore. doi.org/10.1007/978-981-99-7962-2_30

© American Society for Engineering Education, 2024

16 S. Ouyang, J.M. Zhang, M. Harman, and M. Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. arXiv preprint arXiv:2308.02828.
17 "Non-determinism in GPT-4 is caused by Sparse MoE," 152334H. https://152334H.github.io/blog/non-determinism-in-gpt-4/. Accessed Nov 07 2023
18 GitHub (2023) GitHub–biobootloader/wolverine. GitHub. https://github.com/biobootloader/wolverine. Accessed 23 April 2023.
19 GitHub (2023) GitHub– formulahendry/awesome-gpt. GitHub. https://github.com/formulahendry/awesome-gpt. Accessed 23 April 2023
20 GitHub (2023) GitHub– xubujie/llm-application-prompts. GitHub. https://github.com/xubujie/llm-application-prompts. Accessed 23 April 2023

© American Society for Engineering Education, 2024