Improved Maths Solving Abilities for Transformers Using Flow Engineering

Nicolae-Robert Lincă, Traian Rebedea
National University of Science and Technology Politehnica of Bucharest
linca_robert@yahoo.com, traian.rebedea@upb.ro

ABSTRACT

In this paper, we propose a flow engineering-based approach for improving the generation of solutions for math problems in the case of general content generative Natural Language Processing (NLP) models such as OpenAI GPT-4 and Google Gemini. Flow engineering makes use of a mix of prompt engineering techniques and has been recently proposed as a solution to improve code generation for transformer models. To test the performance of our proposed approach we use a subset of one of the most challenging datasets for mathematical solving, MATH, for which top transformer models exhibit accuracy up to 70%. Additionally, we aim to create a framework for generating highly accurate solutions and automating the correctness testing process of solutions. Advancing AI research by improving GPT models in mathematical problem-solving can increase the robustness and versatility of AI. This advancement facilitates interdisciplinary research and promotes collaboration across different fields including the design of efficient chat interfaces, unlocking limitless potential applications.

Author Keywords
Natural Language Processing; Chat Interfaces; Mathematical Reasoning; Transformers; Flow Engineering.

ACM Classification Keywords
I.2.7 Natural Language Processing: Text analysis.
DOI: 10.37789/ icusi.2024.28

INTRODUCTION

In recent years, Generative Pre-trained Transformer (GPT) models have revolutionized natural language processing (NLP). Despite their impressive advancements, their capabilities in solving mathematical problems remain relatively limited. As the popularity and usage of artificial intelligence (AI) continues to grow among both specialists and general users, improving the mathematical capabilities of GPT models holds significant potential.

This study aims to explore various techniques to enhance GPT models' performance in solving mathematical problems by developing a framework based on existing approaches and research in AI interaction. The primary objective of this project is to improve GPT transformers by creating a general process to enhance the accuracy of solving mathematical problems. Considering the attention on improving AI models through specialized training, new datasets, and innovative techniques, this is an opportune moment for delving into new solutions. Another significant aspect is the high cost associated with training or retraining new and existing GPT models for specific purposes. In contrast, significant results can be achieved with much lower costs through prompt engineering.
Improving GPT-powered chat interfaces for mathematical solving is an important feature for providing an enhanced usability and trust in these systems. Moreover, the proposed method using flow engineering has several steps that are similar to human solving and thus provide better explainability – which is another important attribute for improving usability and allowing such solutions to be used by novices as well as proficient maths users.

The main goal is to develop an automated process for generating solutions to various categories of mathematical problems and to measure the accuracy of the provided solutions. This involves three key components:

1. **Dataset**: Selecting a subset of problems from the MATH dataset created by Hendrycks et al. [1], ensuring an equal number of problems from each category and difficulty level.
2. **Correctness Testing System**: Developing a solution to verify the correctness of generated answers based on metrics and GPT models.
3. **Solution Generation Flow**: Combining techniques like prompt engineering, chain-of-thought, and enhanced reasoning into a single flow to improve GPT models' performance and accuracy in solving mathematical problems. This method, named Flow Engineering for Mathematical Problems (FEMP), has been inspired by latest research in code generation with pre-trained transformers, and we have adapted it for mathematical problems.

RELATED WORK

In this section, we provide a summary of several major prompting methods and concepts on which our study is based.

Flow Engineering, introduced by Ridnik et al. [2], is an iterative approach to prompt engineering designed to generate accurate code solutions. It involves a multi-step process where partially correct solutions are iteratively refined through error correction and validation against extensive test sets. Key stages include understanding the problem, generating potential solutions, ranking them, and creating additional AI-generated tests.

This method outlines several prompt design concepts to improve the performance of prompt engineering in the Flow Engineering process. These include structured semantic argumentation, which organizes responses into bullet points for clarity; double validation, where models regenerate results to correct errors; avoiding direct questions to prevent hallucinations and promote deep thinking and reasoning. These strategies enhance the logical structuring, accuracy, and robustness of the generated solutions.

The Chain-of-Thought (CoT) technique, introduced by Wei et al. [3], improves the reasoning capabilities of LLMs by breaking down complex problems into manageable intermediate steps. This method enhances interpretability by providing insights into the model's reasoning, facilitates error correction, and is versatile across various tasks such as mathematics and symbolic manipulation. CoT is easily implemented by including reasoning step examples in prompts, significantly enhancing performance on complex tasks without the need for retraining.

Self-Discovering Reasoning [4] is a two-stage approach that mirrors human problem-solving by leveraging previously accumulated knowledge. The first stage, discovering specific problem structures, involves selecting, adapting, and implementing modules of logical reasoning. The second stage uses these structures to address the problem through guided prompts. This method aims to utilize reflective or critical thinking modules to adapt and apply them effectively for solving specific tasks.

Multi-Agent System for Condition Mining (MACM) proposed by Lei et al. [5] shifts traditional prompt engineering to a condition-based problem-solving framework. It employs three agents - Thinker, Judge, and Executor - who collaborate to identify conditions and solve problems. The Thinker generates and plans the solution, the Judge verifies conditions, and the Executor performs calculations. MACM standardizes responses and enhances performance, although it increases computational calls and operational costs. This approach is notable for its ability to generalize prompts across different problem types.

PROPOSED SOLUTION

As mentioned before, the primary goal of this study was to develop an adaptive framework for solving mathematical problems. Based on the presented methods, we aim to create and adapt Flow Engineering to a specialized form to boost the maths solving abilities of GPTs such as OpenAI GPT-4o and Google Gemini 1.5 Pro.

The main challenge in implementing Flow Engineering, inspired by the one designed for problem-solving in computer science, was transitioning from modular code writing and thinking to approaching mathematical problems similarly. The solution draws from how humans tackle these problems: understanding the problem statement, extracting relevant observations, using these to derive new knowledge, and formulating a solution once sufficient details are gathered, an idea echoed by Lei et al. [5].

Based on all this information, we devised the interactive prompting system presented in Figure 1, which is an adaptation of Flow Engineering for Mathematical Problems (FEMP).

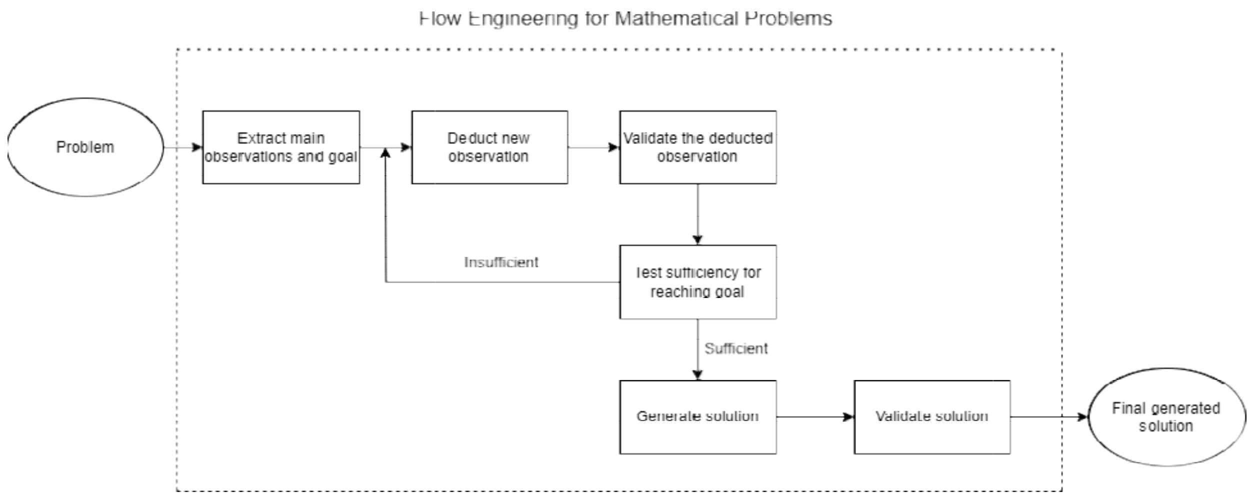

Flow Engineering for Mathematical Problems
Problem
Extract main observations and goal
Deduct new observation
Validate the deducted observation
Insufficient
Test sufficiency for reaching goal
Sufficient
Generate solution
Validate solution
Final generated solution

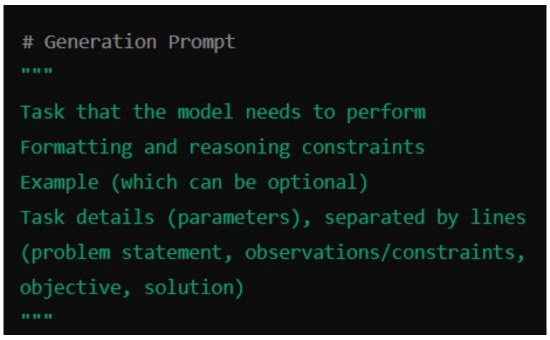
Figure 1. Flow Engineering steps involved in solving mathematical problems.

The flow ensures a systematic, iterative approach to solving mathematical problems, leveraging structured observation, validation, and incremental knowledge accumulation to produce accurate and robust solutions. These are steps of the logical structure for the flow, based on which the prompts are designed:

1. **Extracting Observations and Problem Objective**: This step involves the SELECT phase from the Self-Discovering Reasoning technique, guiding the model to focus on mathematical task-solving, functioning as a Thinker from MACM.
2. **Deducing a New Observation:** This step involves ADAPT phase from Self-Discovering Reasoning, the model formulates a single observation based on previous knowledge and observations, aligning with suggestion of Ridnik et al. [2] for gradual information accumulation.
3. **Validating the Observation**: This step includes two stages: testing an observation's correctness and attempting its correction. The first stage involves double validation of the model's decisions through an AI quorum to limit bias. If deemed relevant, the observation is added to a list of valid observations. If irrelevant, it proceeds to correction attempts. The quorum functions similarly to a Judge in MACM.
4. **Testing Sufficiency of Observations for Objective Achievement**: To stop the loop from generating observations, the AI quorum checks if the observations are sufficient to formulate a solution after generating a new one, akin to the Judge's role in MACM.
5. **Generating a Solution**: Once enough observations are collected, the model formulates a detailed step-by-step solution, performs necessary calculations, and provides a final answer tagged \boxed{answer}. Unlike MACM, here the Executor's role combines with the Thinker to reduce inference calls and operational costs.
6. **Reverifying the Solution**: Based on the double-checking principle introduced by Ridnik et al. [2], this step involves correcting any potential errors in the final solution by rechecking it against the overall problem, observations, and objective.

To design effective prompts for Flow Engineering, each step of the process was assigned a specialized prompt, tailored to the specific needs of that phase. These prompts were categorized into two main types: *generation prompts* and *verification prompts*.

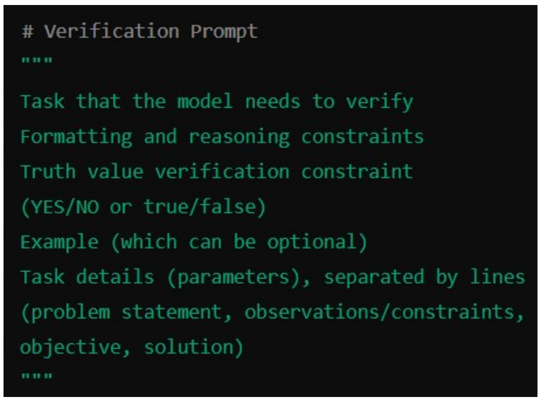
Generation prompts guide the model to produce a response by specifying the task, formatting and reasoning constraints, and including optional examples to demonstrate the expected format. The structure of such prompts is shown in Figure 2.



```
# Generation Prompt
"""
Task that the model needs to perform
Formatting and reasoning constraints
Example (which can be optional)
Task details (parameters), separated by lines
(problem statement, observations/constraints,
objective, solution)
"""
```

Figure 2. Generation prompt structure

Verification prompts, outlined in Figure 3, focus on validating the responses by prompting the model to analyse truth values, guiding it to answer with YES/NO or true/false. Both types of prompts are formatted using Python's triple-quotes to allow for the dynamic insertion of relevant problem details, ensuring clarity and consistency throughout the Flow Engineering process.



```
# Verification Prompt
"""
Task that the model needs to verify
Formatting and reasoning constraints
Truth value verification constraint
(YES/NO or true/false)
Example (which can be optional)
Task details (parameters), separated by lines
(problem statement, observations/constraints,
objective, solution)
"""
```

Figure 3. Verification prompt structure

Both types of prompts are formatted using Python's triple-quotes to allow for the dynamic insertion of relevant problem details, ensuring clarity and consistency throughout the Flow Engineering process.

DIRECT PROMPTING

To establish a baseline for comparison, each model was also run using Direct Prompting, where the model is directly asked to provide a solution for each problem in the dataset. For this implementation, a simple prompt was used, containing minimal context specifying the problem, its category, and difficulty level. The prompt also instructed the model to explain the solution step-by-step, ensuring that intermediate operations were performed in a logical sequence.

The prompt is composed of two parts: role and question. The role prompt provides context for the conversation, specifying the problem category and difficulty, and explaining how the problem should be approached, followed by the actual request for the solution.

TESTING CORECTNESS

To establish an automated method for testing the correctness of solutions generated by GPT models, we considered the following three strategies.

1. Result Testing

This strategy leverages the fact that the MATH dataset problems include the final answer marked with \boxed{answer}. All models were prompted to encapsulate their final answer in this format. Using regular expressions, the final answer can be extracted and verified against the provided solution. This verification serves as an accuracy metric for the tested model. By ensuring the format consistency, we can easily compare the expected and generated answers to gauge correctness.

2. Similarity Degree via Levenshtein Distance

The Levenshtein distance was chosen for evaluating the correctness of model-generated solutions due to its ability to quantify the similarity between two texts by measuring the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. This metric provides an intuitive and comprehensive measure of textual similarity, making it well-suited for comparing the accuracy of generated solutions against correct ones. Additionally, the Levenshtein distance can be normalized to offer standardized and easily interpretable scores, and it can be combined with other similarity measures like the Longest Common Subsequence (LCS) for a more nuanced evaluation. We decided to test this approach by computing and checking the following similarity degrees proposed by Zhang, Hu and Bian [6] for two string S and T:

- $Sim1(S,T) = 1 - \text{ld}(S,T) / \max(m,n)$; where $m=len(S)$, $n=len(T)$ and ld(S,T)=Levenshtein distance of S and T
- $Sim2(S,T) = 1 - ld(S,T) / (ld(S,T) + lcs(S,T))$; where lcs(S,T)=lowest common subsequence of S and T

3. Correctness Testing via AI Quorum

Inspired by the approaches of Zhou et al. [4] and Lei Bin in MACM [5], this strategy takes advantage of the strong textual and contextual analysis capabilities of contemporary AI models. To minimize false positives when two solutions present the same idea overall, we assembled a quorum comprising the following three models:

- Llama3-70b: With 70 billion parameters, Llama3-70b offers deep understanding and high accuracy, reducing the likelihood of false positives due to its robust analytical capabilities.
- GPT-3.5-turbo: Known for its demonstrated ability and performance in natural language processing and superior contextual generation capabilities, this model ensures consistency and avoids interpretation errors. It also offers a lower usage cost compared to more advanced models from OpenAI.
- Gemini 1.5 Pro: Featuring recent innovations and adaptability in context understanding, this model excels in handling complex scenarios due to its specialization in specific tasks.

RESULTS

Dataset

To evaluate the performance of the solution, 105 math problems from the MATH dataset [1] were selected, covering seven categories: Geometry, Algebra, Probability, Prealgebra, Precalculus, Intermediate Algebra, and Number Theory. The MATH dataset classifies problems into five difficulty levels. To achieve significant results, a greater number of higher difficulty problems were chosen. For each category, 15 problems were selected, distributed according to their difficulty levels as shown in Figure 4. The main reason for selecting a smaller subset of the MATH dataset was both for ensuring a good coverage and also for reducing the costs of the evaluation.

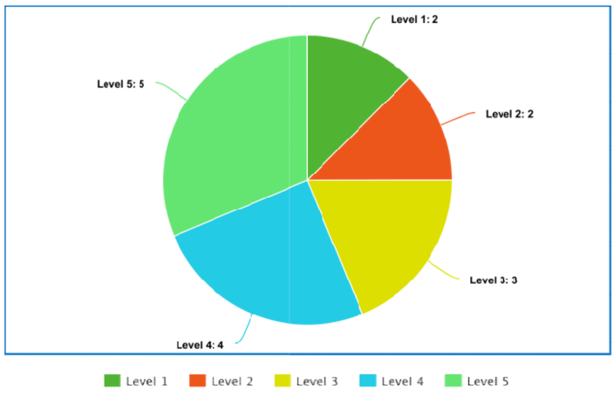


Figure 4. Distribution of problems by difficulty level per category of the dataset

Evaluation of Correctness Methods Performance

In this section, we will present why we considered the best metric for verifying model accuracy to be the comparison of the generated solution with the correct one through an AI quorum.

First, as observed in Table 1, which presents the average similarity scores for both formulas based of Levenshtein Distance, it is evident that regardless of the model or strategy used (Direct Prompting or Flow Engineering), the average similarity scores are too low to be relevant. This indicates that this method is inefficient for verifying the correctness of solutions.

Model \ Strategy	Direct Prompting	Flow Engineering
Similitude score 1		
Gemini-1.5 Pro	0,216	0,224
GPT-4o	0,232	0,231
Similitude score 2		
Gemini-1.5 Pro	0,195	0,264
GPT-4o	0,173	0,236

Table 1. Average similitude scores based on Levenshtein Distance by Model per Strategy

Secondly, regarding the final answer verification extracted through regular expressions from the Flow Engineering responses executed on each model, Figure 5 shows a strong correlation between the AI quorum's truth value and the actual response. This correlation confirms the high performance of the quorum in identifying whether a solution is indeed correct. The presence of false positives indicates that GPT models are not always able to formulate an answer identical to the official solution due to interpretation or presentation aspects, while the quorum detects these discrepancies.

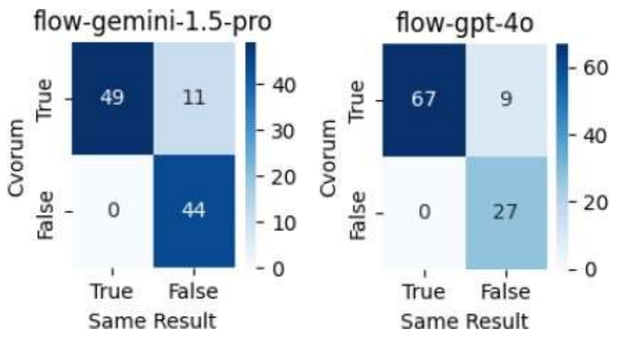


Figure 5. Heatmap to highlight correlation between AI Quorum Correctness and Final Answer for Math Problems

A good example extracted from the model responses in the case of false positives is when the solution used the infinity sign (∞) encoded in UTF-8, while the official solution wrote "infinity," making it impossible to detect only through regular expressions. Another commonly encountered example was the difference between presenting fractions as they are or in decimal form.

An important aspect is the absence of false negatives, which reinforces the assertion that the AI quorum is the most capable of detecting the correctness of a solution.

Solution Performance

In evaluating the performance of the models based on the chosen strategies, Table 2 presents the accuracy (percentage of problems solved) of the two models based on the strategy used.

The GPT-4o model has shown a raise of 4,2% which is considerable when the best obtained on MATH dataset provided a raise of around 20% in MACM [5]. On the contrary, the Gemini-1.5 Pro presented a decrease in performance when using the Flow Engineering, this will be elaborated further when analysing the distribution of solved problems for each model.

As shown in Figure 6, when moving from the Direct Prompting to Flow Engineering, the Gemini-1.5 Pro presents inconsistency by being unable to maintain the problem solved with the simpler approach. This is caused by the extensive number of prompts and elaborate thinking process implied by the interactive system of FEMP. Another cause may also be the fact that this study was based on many solutions and approaches tailored for the GPT-4 from OpenAI.

Model \ Strategy	Direct Prompting	Flow Engineering
Gemini-1.5 Pro	63,8%	57,6%
GPT-4o	69,5%	73,7%

Table 2. Accuracy (%) comparison by Model per Strategy on the test dataset

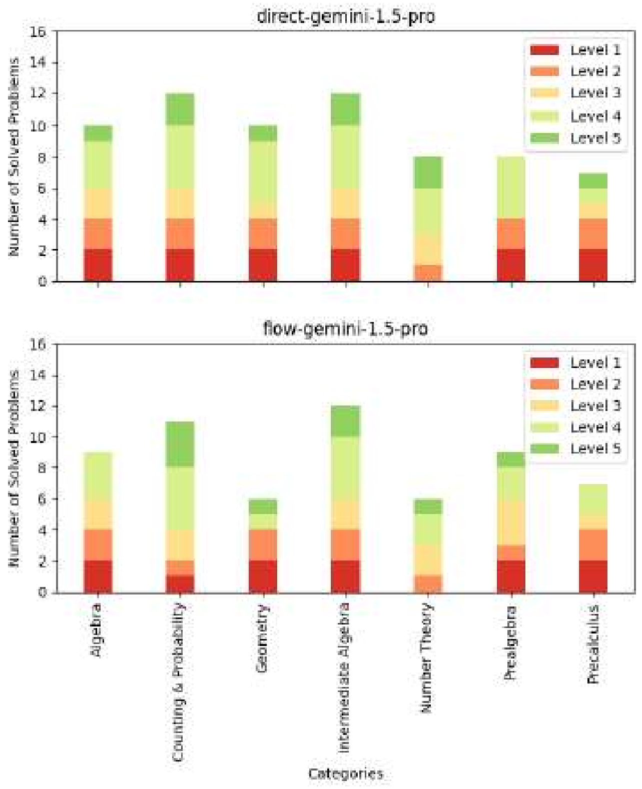


Figure 6. Distribution of solved problems by Gemini-1.5 Pro

In contrast with Gemini, GPT-4o has shown the potential of FEMP having the increase in accuracy and the consistency when passing between the two strategies, as presented. The raise of performance is the result of the increased number of difficult problems of at least level 3. This is an important aspect for the Flow Engineering which proves the benefits it could add to improving the abilities of solving mathematical problems for GPTs through prompt engineering.

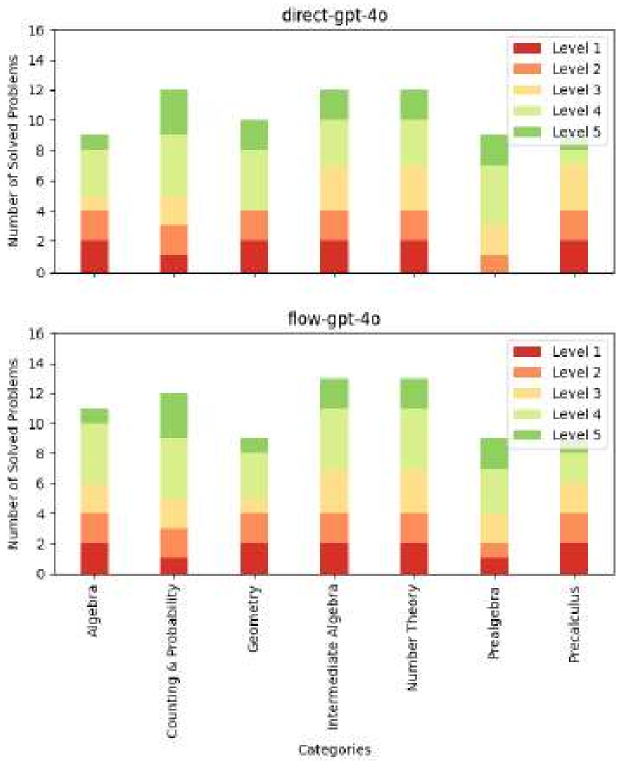


Figure 7: Distribution of solved problems by GPT-4o

CONCLUSION

The implementation of Flow Engineering has demonstrated its potential to enhance the performance of GPT models in solving complex mathematical problems. Notably, Flow Engineering has been particularly effective in addressing problems with difficulty levels above 3, especially in categories such as Algebra, Number Theory, and Probability.

Flow Engineering prompts were optimized based on GPT- 4's capabilities, which may have contributed to performance inconsistencies in other models. GPT-4o remained relatively consistent across both Direct Prompting and Flow Engineering due to these optimizations. However, the small and randomly selected dataset introduced variability in results, indicating a need for a more selective and extensive dataset for stable and deterministic outcomes.

FUTURE RESEARCH DIRECTIONS

To further enhance the proposed solution, several avenues for future research and improvements have been identified. Extending Flow Engineering by exploring new approaches that leverage the strengths of each model and testing and integrating new models and techniques, such as Tree-of-Thought (TOT) [7] and Graph-of-Thought (GOT) [8], could lead to significant advancements. Improving the dataset and testing methods is another critical area. Expanding the dataset to provide a comprehensive overview of Flow Engineering's improvements and developing a more qualitative selection of problem distributions would result in more reliable outcomes. Optimizing execution methods by replacing static synchronization with automated detection to handle rate limits more efficiently is also crucial.

REFERENCES

1. D. Hendrycks *et al.*, 'Measuring mathematical problem solving with the math dataset', *arXiv preprint arXiv:2103.03874*, 2021.
2. T. Ridnik, D. Kredo, and I. Friedman, 'Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering', *arXiv preprint arXiv:2401.08500*, 2024.
3. J. Wei *et al.*, 'Chain-of-thought prompting elicits reasoning in large language models', *Adv Neural Inf Process Syst*, vol. 35, pp. 24824–24837, 2022.
4. P. Zhou *et al.*, 'Self-discover: Large language models self-compose reasoning structures', *arXiv preprint arXiv:2402.03620*, 2024.
5. B. Lei, 'MACM: Utilizing a Multi-Agent System for Condition Mining in Solving Complex Mathematical Problems', *arXiv preprint arXiv:2404.04735*, 2024.
6. S. Zhang, Y. Hu, and G. Bian, 'Research on string similarity algorithm based on Levenshtein Distance', in *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, IEEE, 2017, pp. 2247–2251.
7. S. Yao *et al.*, 'Tree of thoughts: Deliberate problem solving with large language models', *Adv Neural Inf Process Syst*, vol. 36, 2024.
8. B. Lei, C. Liao, and C. Ding, 'Boosting logical reasoning in large language models through a new framework: The graph of thought', *arXiv preprint arXiv:2308.08614*, 2023.