Comparison of Python code generated by ChatGPT and AlphaCodium

Rareş Mihai Dinu
University of Craiova
raresmihai18@gmail.com

Marian Cristian Mihăescu
University of Craiova
cristian.mihaescu@edu.ucv.ro

Traian Eugen Rebedea
University Politehnica of Bucharest
traian.rebedea@cs.pub.ro

ABSTRACT
This paper presents a comparative analysis of the performance of ChatGPT and AlphaCodium in generating Python code for algorithmic problem-solving, using tasks sourced from the competitive programming platform Codeforces. The study evaluates how effectively each model interprets problem statements, generates correct and efficient solutions, and handles edge cases and complex scenarios under the platform's strict time and memory constraints. Both models were assessed by executing their generated solutions, which were then evaluated by Codeforces' judging system. Unlike previous studies that often rely on closed or controlled environments, this research leverages an open, reproducible, and objective testing framework provided by Codeforces to deliver quantifiable and verifiable results. The findings contribute to a deeper understanding of the practical capabilities and limitations of AI-based code generation models in competitive programming contexts, offering valuable insights for both academic research and industry applications.

Author Keywords
AI Code Generation, ChatGPT, AlphaCodium, Competitive Programming, Python, Codeforces, Algorithmic Problem Solving, Natural Language Processing, Program Synthesis, Model Evaluation

ACM Classification Keywords
Computing methodologies - Artificial intelligence - Natural language processing (I.2.7)

DOI: 10.37789/icusi.2025.14

INTRODUCTION

In recent years, the rapid evolution of artificial intelligence models has significantly impacted the field of software development, particularly in the domain of algorithmic problem-solving and competitive programming. Platforms such as Codeforces have become key benchmarks for assessing the capability of both human programmers and AI-based code generation tools. Through its automated testing system and well-structured problem classifications, Codeforces offers an objective, transparent, and reproducible environment for evaluating solutions under strict time and memory constraints.

ChatGPT [1], developed by OpenAI, and AlphaCode, developed by DeepMind, along with its derivative, AlphaCodium [2] have emerged as leading AI models capable of tackling complex algorithmic challenges. AlphaCodium's performance, demonstrated by its ranking in the top 54% of human competitors in a controlled Codeforces-based evaluation, highlights its potential in this field. Similarly, ChatGPT, especially in its GPT-4-based variants, has shown remarkable improvements in algorithmic reasoning, code generation, and problem-solving abilities, becoming a widely used tool among programmers for tasks such as brainstorming, debugging, and solution development.

Despite their promising capabilities, direct, objective, and openly verifiable comparisons between these models remain limited. Existing evaluations often rely on closed testing environments or provide primarily qualitative assessments, which can obscure the practical applicability of such models in real-world scenarios. This paper addresses this gap by proposing a comparative study based on the Codeforces platform, using its public, standardised, and automated evaluation system to ensure accuracy, fairness, and reproducibility.

The primary objective of this research is to analyse and compare the performance of ChatGPT and AlphaCodium in generating Python code solutions for a diverse set of real algorithmic problems with varying difficulty levels. By emphasising this two-step acceptance metric, the study offers a focused and practical perspective on the immediate and final accuracy of the generated solutions, providing clear, measurable insights into the models' performance. The selected problems span various levels of difficulty, ensuring a comprehensive assessment of the models' capabilities across a diverse problem set.

By adopting this methodology, the paper aims to provide practical, quantifiable insights into the strengths and limitations of these models in competitive programming contexts. Ultimately, the research contributes to a better understanding of how artificial intelligence can complement human efforts in algorithmic problem-solving and software development, supporting a responsible and informed integration of these technologies in both educational and industrial settings.

The project utilises a dataset of algorithmic problems sourced from the Codeforces platform, carefully selected to cover a wide range of difficulty levels. This approach ensures a comprehensive evaluation of both models,

providing meaningful insights into their respective strengths, weaknesses, and optimal use cases across different problem complexities.

Alongside the results obtained, this paper covers the data preparation stage for each model. Within the same concept, the problem data is written in a manner that is accessible for both approaches, resulting in minimal to no alterations to the testing environment.

RELATED WORK

The domain of automated code generation using large language models has advanced significantly in recent years, driven by the increasing capabilities of transformer-based architectures and the availability of large-scale code datasets. Numerous studies have explored the application of these models in both general-purpose programming and competitive programming contexts, with a particular focus on assessing their problem-solving efficiency, accuracy, and semantic understanding of code. This section reviews a selection of relevant works that provide essential background for understanding the performance and evaluation of models such as ChatGPT and AlphaCodium in the context of competitive programming tasks. The selected literature highlights key methodologies, evaluation frameworks, and model-specific advantages or limitations that collectively contribute to the current state of research in automated programming. By situating the present study within this body of work, a clearer perspective emerges regarding the comparative strengths, weaknesses, and practical implications of deploying such models in real-world coding environments.

Li et al. [3] present the performance of AlphaCode, a language model developed explicitly for solving competition-level programming problems. The system is evaluated on the Codeforces platform, which, at the time of writing, is one of the most widely used competitive programming environments. Using a combination of transformer-based architectures and sampling strategies, AlphaCode achieved a performance ranking within the top 54% of Codeforces users. The paper emphasises that AlphaCode's success relies on generating a large set of candidate solutions followed by a filtering mechanism based on test cases. Despite the promising results, the authors acknowledge the scalability limitations of the approach due to computational costs. Moreover, the underlying methodology is highly dependent on the diversity of the sampled solutions, which remains a challenge in problems with larger search spaces.

Liu et al. [4] extensively evaluate the correctness of code generated by large language models, including ChatGPT, through the introduction of the EvalPlus framework. The core of their work lies in improving the traditional evaluation settings by supplementing existing public test cases with additional, more challenging ones, thus mitigating the incompleteness of the HumanEval dataset. By leveraging extensive mutation-based test generation, the authors demonstrate that the pass@k metric, commonly used in code generation evaluation, is significantly overestimated under previous benchmarks. Despite achieving faster computational times, ChatGPT's performance deteriorates when subjected to the enhanced test suites proposed by EvalPlus. Liu et al. highlight that the superficial validation used in prior work might have overrepresented the models' true problem-solving capabilities.

Zhou et al. [5] introduce CodeBERTScore, a novel metric designed for evaluating code generation through the semantic alignment between predicted and reference code. The method builds upon the CodeBERT model, specifically pre-trained on source code, which ensures the semantic granularity necessary for cross-language code comparisons. Unlike traditional string-matching metrics, CodeBERTScore emphasises the importance of capturing functional equivalence rather than exact syntactic matches. By conducting experiments across several datasets, including HumanEval and MBPP, the authors demonstrate that their metric correlates more strongly with functional correctness and human preference than previous metrics such as BLEU or exact match. Despite this improvement, Zhou et al. note that the metric's reliability is contingent on the representational quality of the pre-trained code models, which might vary across programming languages.

Tong and Zhang [6] develop CodeJudge, a framework designed to evaluate code generated by large language models without requiring traditional test cases. The key innovation in their work is the semantic similarity evaluation pipeline, which leverages pre-trained language models to compare the model-generated solution against a reference at a deeper functional level. By using both natural language and code-based embeddings, the authors provide an alternative to functional testing that is particularly valuable in scenarios where test cases are incomplete or unavailable. The study demonstrates that CodeJudge can predict functional correctness with a high degree of correlation to traditional testing. Still, the authors also acknowledge the sensitivity of their framework to the variability in code representations and the inherent limitations of relying on pretrained embedding spaces.

Siam et al. [7] conduct a comparative study evaluating multiple AI programming assistants, including ChatGPT, AlphaCode, Gemini, and GitHub Copilot. The study systematically benchmarks these models across widely used datasets such as HumanEval and APPS, focusing on their ability to generate syntactically correct and functionally accurate code under standard conditions. According to the reported results, AlphaCode consistently outperforms ChatGPT and Copilot in terms of solution accuracy, particularly in complex competitive programming scenarios. The paper emphasises the trade-offs between model responsiveness and solution correctness, with ChatGPT providing faster responses at the cost of lower

accuracy. Siam et al. highlight the complementary nature of the models, suggesting that the choice of tool should depend on whether rapid prototyping or high-stakes competitive problem solving is the primary goal.

Yetiştiren et al. [8] conduct an empirical study evaluating the code quality of AI-assisted programming tools, specifically GitHub Copilot, Amazon CodeWhisperer, and ChatGPT. Their analysis, based on the HumanEval dataset, assesses multiple dimensions of generated code, including correctness, reliability, maintainability, and security. The results highlight that while all models exhibit potential as programming assistants, significant differences exist in the trade-offs between solution accuracy and code quality. This study is particularly relevant as it broadens the comparative scope beyond pairwise evaluations, offering a multi-tool perspective that situates ChatGPT's performance within the broader ecosystem of AI coding assistants.

Yan et al. [9] focus on understanding how ChatGPT's code generation abilities vary across tasks of different difficulty levels. By systematically categorising problems into low, medium, and high complexity tiers, the authors demonstrate that ChatGPT performs considerably better on simpler tasks, while its accuracy and reliability decline as problem difficulty increases. This observation closely aligns with the findings of the present study, reinforcing the importance of difficulty-based analysis when evaluating AI code generation models. Moreover, their work provides additional evidence that task complexity is a critical factor in determining the practical applicability of such models in real-world problem-solving contexts.

PROPOSED APPROACH

The primary objective of the present system is to perform a comparative evaluation of the results obtained by AlphaCodium and ChatGPT. The study aims to draw precise, quantifiable conclusions regarding the models' ability to solve algorithmic problems and their respective acceptance rates within the Codeforces competitive programming platform.

The core goal of this project is straightforward: to determine the most effective approach for solving competitive programming problems. The system follows an intuitive and user-guided process. The user selects a problem from Codeforces and provides its link to the system. The problem information is then automatically converted into a format compatible with both approaches, AlphaCodium and ChatGPT. The user chooses one of the two models, which subsequently generates a solution for the selected problem. To validate the solution, the user manually submits it to the Codeforces judging system, where the solution's correctness is objectively assessed.

The Dataset

The dataset employed in this study is a curated collection of 399 competitive programming problems sourced from the Codeforces platform, a widely recognised online judge specialising in algorithmic problem-solving and competitive programming contests. The problems selected cover a broad spectrum of difficulty levels, ranging from 800 to 2500, which aligns with the Codeforces rating system, where lower values denote introductory tasks and higher values correspond to highly challenging problems.

This dataset represents a subset extracted from the publicly available "Codeforces Competitive Programming Dataset[1]" hosted on Kaggle. The selection ensures a representative sample that encompasses a diverse array of problem types and difficulty distributions commonly encountered in competitive programming environments.

The dataset was systematically classified based on the Codeforces difficulty rating, which allowed for the segmentation of the problems into three distinct categories, each characterised by specific algorithmic and cognitive requirements. With that low (with scores between 800 to 1100), medium (1200-1700) and high (1800-2500) difficulty problems were assigned.

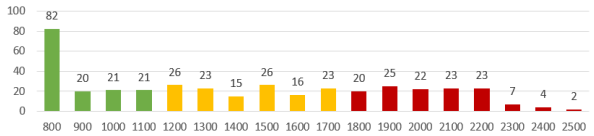

Figure 1. Problem difficulty distribution

The dataset exhibits a higher concentration of problems at lower difficulty levels, which reflects the general composition of publicly available competitive programming repositories such as Codeforces. Lower-rated problems are more abundant, as they serve as introductory exercises designed to engage a wider pool of participants, including beginners. Conversely, high-difficulty problems are deliberately fewer, since they target advanced competitors and are intended to appear less frequently in contests. This natural imbalance ensures that the dataset remains representative of the problem distribution encountered in real-world competitive programming environments, while also facilitating a more comprehensive evaluation of model performance across accessible tasks.

The dataset comprises 399 JSON files, each encapsulating the complete information required for the accurate understanding and resolution of a specific problem. The dataset's directory structure is systematic, with each file named using the pattern "*my_problems_X.json*", where X denotes the sequential index of the problem, ranging from 1 to 399.

[1] Codeforces Competitive Programming Dataset, https://www.kaggle.com/datasets/dinuiongeorge/codeforces-competitive-programming-dataset, last accessed on 29.06.2025

Each JSON file adheres to a standardised structure, consisting of the following key fields:

- *name*: Specifies the official problem title as published on Codeforces.
- *description*: Provides a comprehensive textual specification of the problem.
- *public_tests*: Contains an array of publicly available test cases intended for preliminary validation. Each element in this array includes:
  - *input*: The input data for the test.
  - *is_valid_test*: A flag indicating the validity of the test case.
  - *output*: The expected output for the given input.

The problems were systematically collected and processed to ensure compatibility with the AlphaCodium system's input format. An additional step in the data preparation process involved web scraping techniques, where relevant problem statements and associated data were extracted directly from the Codeforces platform by parsing HTML elements.

One notable challenge encountered during the dataset preparation involved the representation of mathematical expressions, which are frequently embedded in the original problem statements. These expressions can range from simple inequalities (e.g., $x \leq y$) to complex summations or combinatorial formulas (e.g., $\sum_{i=1} x^i a^{n-i}$).

Given the limitations of the JSON format in representing complex mathematical notation, the adoption of LaTeX syntax was selected as an appropriate solution. LaTeX provides a standardised, machine-readable format for mathematical expressions, ensuring that essential mathematical relationships are preserved during data processing.

User Interaction

The user interaction within this system was primarily indirect, being designed around the development of an automated pipeline that required minimal human intervention during the data acquisition phase. The core of the process was the structured parsing and systematic processing of competitive programming problems sourced from the Codeforces platform.

The acquisition began with an initial dataset composed of direct links to individual problem pages. These URLs served as critical access points for retrieving detailed problem content from the web. To perform the extraction, advanced web scraping techniques were applied, combining the functionality of the BeautifulSoup[2] library with the *curl_cffi*.requests[3] module. This configuration allowed efficient handling of HTTP requests and ensured compatibility with modern, potentially dynamic web content.

After retrieving the HTML content of each problem, the data were carefully parsed to extract the essential components required for subsequent processing. The parsing process isolated key elements such as the problem title, full statement, input and output specifications, illustrative examples, and supplementary explanatory notes that could aid comprehension. Particular attention was given to preserving the integrity of the extracted information, ensuring that all structural elements and semantic details remained intact and unaltered during the conversion.

Following extraction, the collected data were first stored in a Google Sheets document and exported in CSV format to enable organised visualisation and facilitate manual validation when necessary. Each row was structured to include the problem identification number, source link, title, detailed statement, input and output requirements, illustrative examples, explanatory notes, and assigned difficulty rating. Once validated, the dataset was programmatically converted into JSON files strictly conforming to the schema required by the AlphaCodium system. These files encapsulated problem metadata, segmented descriptions of statements and specifications, and corresponding public test cases, thereby ensuring structural consistency and compatibility for downstream processing.

AlphaCodium

For the development environment, Google Colab was selected due to its support for the latest Python-based technologies and its seamless integration with Google Drive, which facilitated efficient storage and management of project data and results.

Following the environment setup, the required Python packages were installed based on the provided requirements file, which largely specified appropriate version constraints.

A critical component of the AlphaCodium system was the integration of an API key for accessing ChatGPT models within the workflow. The acquisition and configuration of this API key were straightforward and did not present significant technical obstacles.

AlphaCodium employs a multi-stage workflow that emulates human problem-solving strategies while leveraging the efficiency of artificial intelligence. The process begins with a self-reflection phase, where the system analyses the problem, identifies requirements, and

[2] BeautifulSoup, https://beautiful-soup-4.readthedocs.io, last accessed on 29.06.2025

[3] curl_cffi, https://github.com/lexiforest/curl_cffi, last accessed on 29.06.2025

considers algorithmic paradigms and constraints. This reasoning step establishes a foundation before code generation, ensuring that comprehension precedes implementation. A subsequent validation phase then checks the logical consistency of the analysis before advancing to solution development.

Once validated, AlphaCodium generates several candidate implementations and evaluates them based on correctness, clarity, and likelihood of success. The most promising solution is selected and further tested with automatically generated edge cases derived from structural analysis. This approach increases robustness by revealing hidden flaws and supports iterative refinement, approximating the adaptive reasoning employed by human programmers. Once the solution is generated, it is evaluated against the public (user-provided) test cases. In the event of test failures, AlphaCodium autonomously initiates a repair cycle, which involves re-analysing the problem, adjusting the solution, and re-executing the relevant tests.

Even though AlphaCodium may not consistently achieve full test pass rates, the process highlights essential strategies for enhancing AI-assisted programming systems. These include repeated iterative refinement, continuous self-evaluation, feedback integration from test executions, and adaptive solution regeneration.

ChatGPT

ChatGPT 4o-mini model was employed as the primary language model for data processing and code generation. The selection of this specific model was strategically motivated by its optimal balance between computational efficiency and generative capability. Empirical observations confirmed that 4o-mini is capable of producing precise, contextually appropriate solutions tailored to the specifications of competitive programming tasks.

The interaction with the model was systematically structured. Each task was submitted via a standardised prompt format, phrased as:

"*Solve the following problem. Only write the Python code.*"

This prompt was dynamically completed with task-specific content extracted from the corresponding JSON files. The decision to employ a concise and directive prompt structure was critical in constraining the model's output exclusively to Python code, deliberately excluding explanatory text, comments, or non-executable content. This controlled interaction paradigm ensured syntactic purity and eliminated ambiguity in the model's response, facilitating seamless downstream processing.

Upon receiving the AI-generated response, only the Python code was extracted and stored for subsequent stages, including automated testing or additional analysis, as required by the evaluation framework. The workflow was fully automated (from prompt construction to solution storage and validation), enabling high-throughput processing, reducing manual intervention, and ensuring consistent handling across tasks. This integration, supported by structured data extraction from JSON files and uniform prompt templates, established efficient coupling between the generation engine and the processing infrastructure, thereby ensuring operational consistency, repeatability, and reliability throughout the entire pipeline.

The 4o-mini model also contributed significantly to the efficiency and accuracy of the workflow. Despite being a compact version within the GPT-4 model family, 4o-mini consistently generated functional, context-relevant code, particularly suited for educational and competitive programming scenarios.

By enforcing the exclusion of non-code elements from the model's output, the approach substantially reduced the risk of parsing errors and interpretative inconsistencies. This precision was particularly advantageous for compatibility with automated testing pipelines, where input streams are expected to consist solely of executable code segments.

EXPERIMENTAL RESULTS

This section will cover both the methods used to test the two generation methods and the obtained results.

Testing

The testing phase was designed to rigorously evaluate the solutions generated by ChatGPT and AlphaCodium, both selected for their demonstrated ability to interpret natural language and autonomously produce functional code for competitive programming tasks. After solution generation, each code fragment was executed within a controlled local environment using the official test cases extracted from the corresponding JSON files. This process ensured a systematic assessment of functional correctness by comparing the program outputs with the expected results defined in the problem specifications. Outcomes were classified into four categories, with solutions passing all test cases labelled as correct and those failing categorised as incorrect. Correct local solutions underwent an additional verification step, where they were manually submitted to the Codeforces platform. This procedure, though time-intensive, confirmed robustness against hidden test cases and provided a reliable benchmark of model accuracy under real-world competitive programming conditions.

Throughout this pipeline, several technical challenges were encountered that influenced performance. A primary limitation for AlphaCodium was sensitivity to token constraints, which often caused incomplete outputs for complex problems with lengthy statements. Both models also exhibited formatting inconsistencies, such as incorrect indentation, unsupported characters, or missing structural components, occasionally producing outputs that mixed code with natural language explanations. These issues frequently rendered the code non-executable, though rerunning the models on the same input sometimes yielded

valid outputs, highlighting the variability inherent in their generative processes.

Results – AlphaCodium

The results obtained from AlphaCodium's performance across the dataset of 399 competitive programming problems reveal several critical trends and limitations. Out of the total problems, AlphaCodium failed to generate any solution for 56 instances, effectively reducing the number of attempted problems to 343. The inability to produce output for a significant portion of the dataset is a direct consequence of the token limitations imposed by the model's API, as previously discussed. This constraint was particularly evident for problems of higher complexity, where the length of the problem statements and the accompanying explanations often exceeded the model's token processing capacity, leading to systematic failures in code generation.

The evaluation of AlphaCodium considered two performance thresholds: passing the initial public test case and achieving full correctness by passing all official Codeforces tests. The first threshold served as an early indicator of the model's capacity to capture the fundamental structure of a problem, while the second represented complete validation of the solution. AlphaCodium achieved its strongest results on low-difficulty problems, with pass rates on the first public test ranging from roughly 50% to 25%, highest at the lower end of the spectrum. For medium-difficulty tasks, performance stabilised around 25%, suggesting that the model was able to generate partial but viable solutions with a degree of consistency across this range.

At higher difficulty levels (above 1800), AlphaCodium's performance became far less stable, with success rates fluctuating significantly as complexity increased. These inconsistencies appear linked to the combined effects of longer problem statements, more intricate constraints, and architectural limitations in processing extensive contextual information. Nonetheless, even when full correctness was not achieved, the ability to pass initial public tests indicated that the model often captured elements of the correct reasoning path. This partial alignment suggests that AlphaCodium's solutions, while incomplete, may only require refinement or additional guidance to reach full correctness

The final evaluation of AlphaCodium's performance, based on the successful completion of all test cases on the Codeforces platform, reveals a marked decline in accuracy as problem difficulty increases. For lower difficulty levels (800-1100), the model achieved relatively high success rates, with correct solutions in approximately 45-50% of the cases. However, as problem complexity grew, AlphaCodium's effectiveness decreased rapidly. In the medium difficulty range (1200-1700), success rates fell to between 10% and 25%, indicating difficulties in handling problems that require advanced algorithmic reasoning.

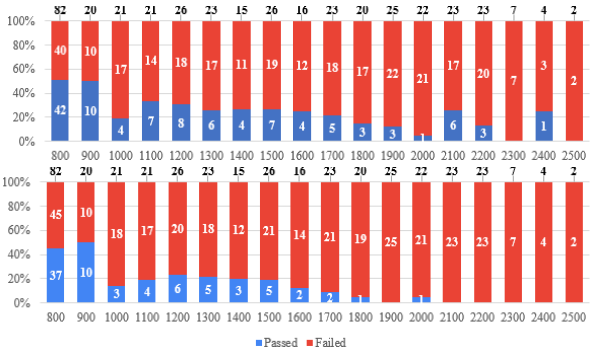

Figure 2. AlphaCodium results – first test (top) vs. all tests (bottom)

At higher difficulty levels (1800 and above), AlphaCodium's performance became negligible, with only two problems solved correctly across all attempts, and no successful solutions for problems rated 2000 or higher. This highlights clear limitations in the model's capacity to solve expert-level tasks.

The average runtime per problem was between four and six minutes, suggesting a substantial computational effort. However, this processing time was generally insufficient to produce correct solutions for complex problems, likely due to the increased logical depth and solution space exploration required.

For the final passing rates, AlphaCodium abstained 28.57% for the first public test threshold, while 19.79% was scored for the overall success rate.

ChatGPT – Results

Similar to the methodology applied to AlphaCodium, ChatGPT was evaluated using the same testing framework, which involved passing both the initial public test and the comprehensive validation set on the Codeforces platform. Unlike AlphaCodium, ChatGPT successfully processed the entire dataset without encountering token limit issues. This robustness is primarily due to the model's streamlined communication protocol, where only the problem description was transmitted in a single API call, thereby avoiding overloading the communication channel. Consequently, ChatGPT demonstrated significantly faster response times, averaging approximately 10 seconds per problem.

ChatGPT's performance on the initial public test, used as a preliminary indicator of conceptual understanding, showed favorable results for low-difficulty problems. In this range, success rates varied between 25% and 40%, aligning with expectations for entry-level tasks that demand limited contextual reasoning. For medium-difficulty problems, the model maintained a relatively stable accuracy of around 20%, with slight improvements near the upper boundary of

this interval. However, as problem complexity increased, its effectiveness declined sharply. In the high-difficulty range, success rates did not exceed 20% and dropped to zero for the most challenging problems, reflecting the model's limitations in addressing tasks that require advanced algorithmic reasoning.

A closer analysis of the distribution confirms this trend, with accuracy strongly correlated to problem difficulty. At the lowest tier, ChatGPT solved 35 of 82 problems rated 800, corresponding to an accuracy of roughly 43%. Performance decreased with increasing complexity, with only 7 of 20 problems solved at a rating of 900, and just 4 and 5 correct solutions at ratings of 1000 and 1100, respectively. These findings indicate that while ChatGPT demonstrates efficiency in producing rapid responses, its accuracy is not guaranteed, even for relatively simple problems, and deteriorates substantially as problem complexity increases.

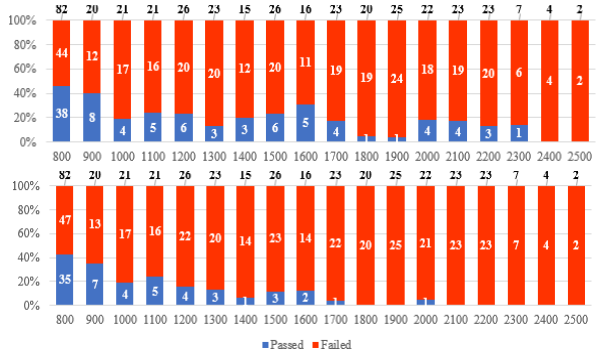

Figure 3. ChatGPT results – first test (top) vs. all tests (bottom)

As the difficulty increases, the model's performance deteriorates markedly. At a score of 1200, only 4 out of 26 problems were solved correctly, and at 1300, just 3 out of 23. For problems rated at 1400 and 1500, the success rate drops to minimal levels: only 1 out of 15 problems at 1400 and 3 out of 26 at 1500 were resolved correctly. These outcomes suggest that ChatGPT faces progressively greater challenges in generating accurate solutions as the logical and algorithmic requirements of the tasks become more intricate.

In the case of high-difficulty problems, the model's performance is nearly negligible. Only two problems were solved correctly, while the remaining attempts were unsuccessful. This result clearly underscores a major limitation of ChatGPT in addressing problems that require deep comprehension and advanced problem-solving strategies. It highlights the model's significant difficulties in handling tasks with high algorithmic complexity.

In the end, ChatGPT managed to score 24.06% for the first threshold, while 16.54% was marked for the second.

Comparing results

The following diagram presents the final acceptance rates of both models across varying levels of problem difficulty. Consistent with the trends observed in the previous illustrations, AlphaCodium generally outperforms ChatGPT or, at minimum, achieves comparable results. An exception is observed in the lower difficulty range (1000 - 1100), where ChatGPT attains higher success rates, whereas AlphaCodium demonstrates superior performance across the remaining difficulty levels.

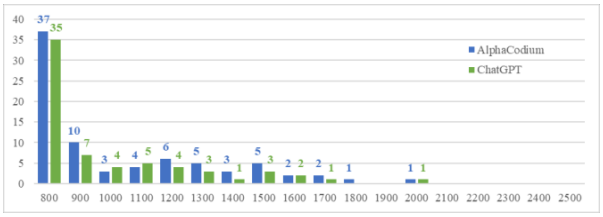

Figure 4. AlphaCodium vs. ChatGPT – accepted solutions

To gain deeper insights into the underlying problem-solving processes of the models, the following table summarises the outcome of each generated solution. The table indicates whether a problem was solved correctly, incorrectly, or failed due to issues such as memory or time limit constraints.

Output status	AlphaCodium (GPT-4o mini)		ChatGPT (GPT-4o mini)	
	First test	All tests	First test	All tests
Passed	114	79	96	66
Failed	278	305	297	320
Time lim.	7	14	6	12
Mem. lim.	0	1	0	1

Table 1. Problem output status

The comparative results indicate that AlphaCodium achieves higher pass rates than ChatGPT at both evaluation stages. While both models exhibit a substantial number of failed attempts, ChatGPT records slightly higher failure counts across both stages. Time-limit and memory-limit errors occur infrequently in both models, though AlphaCodium shows marginally higher susceptibility to time-limit issues. Overall, these findings suggest that AlphaCodium demonstrates a modest advantage in generating correct solutions, albeit at the cost of a slightly higher incidence of execution-time constraints.

FUTURE WORK

Building upon the findings of the present study, several directions for future research can be identified. Firstly, the experimental framework will be extended to incorporate more recent and larger-scale models, such as GPT-4 and its successors, in order to evaluate the extent to which architectural advancements and expanded training corpora improve performance in competitive programming

contexts. This extension will provide a more comprehensive understanding of the state of the art in AI-driven code generation.

Needless to say, future experiments will incorporate human-generated solutions alongside those produced by the models. This addition will enable a more organic and realistic benchmark, situating AI performance within the broader spectrum of human problem-solving abilities and offering valuable insights into complementarities between human and machine-generated approaches.

Furthermore, a more in-depth diagnostic analysis of model outputs will be undertaken. Beyond simply classifying solutions as correct or incorrect, this analysis will aim to investigate the specific causes of failure, including logical missteps, algorithmic inefficiencies, and misinterpretations of problem constraints. Furthermore, the degree of deviation from the correct solution will be systematically assessed, thereby providing a finer-grained perspective on the proximity of erroneous outputs to viable implementations.

The scope of the evaluation will be broadened to include additional programming languages beyond Python. This expansion will test the generalizability of the models' problem-solving capabilities across diverse syntactic and semantic environments, ultimately yielding a more holistic understanding of their applicability to real-world programming scenarios.

CONCLUSIONS

Following the experiments and the subsequent analysis of the obtained results, several clear conclusions can be drawn regarding the performance and applicability of the evaluated models.

Both AlphaCodium and ChatGPT have demonstrated their potential as valuable tools for addressing competitive programming problems. Considering the accuracy rates achieved during the testing phase, both systems prove to be viable solutions. Although AlphaCodium achieved marginally superior results, approximately 3% higher accuracy compared to ChatGPT, both approaches can be effectively employed in the problem-solving process. In terms of operational differences, ChatGPT offers the advantage of significantly faster response times, making it suitable for scenarios where prompt feedback is essential. In contrast, AlphaCodium provides a higher probability of generating a correct solution, making it preferable when solution accuracy is prioritised over processing speed.

From a strictly statistical and performance-oriented perspective, AlphaCodium consistently outperforms ChatGPT in solving competitive programming tasks. Despite its limitations in processing a smaller number of problems due to token constraints, AlphaCodium achieved higher success rates in the problems it was able to attempt. With an observed performance advantage of approximately 3%, AlphaCodium emerges as the more effective approach for obtaining correct solutions. However, this improved accuracy is accompanied by higher computational costs and longer processing times, suggesting a trade-off between solution quality and resource efficiency. Ultimately, the superior results obtained by AlphaCodium justify the additional investment in both time and computational resources.

REFERENCES

1. Welsby, P., & Cheung, B. M. (2023). ChatGPT. Postgraduate Medical Journal, 99(1176), 1047-1048.
2. Ridnik, T., Kredo, D., & Friedman, I. (2024). Code generation with alphacodium: From prompt engineering to flow engineering. arXiv preprint arXiv:2401.08500.
3. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with alphacode. Science, 378(6624), 1092-1097.
4. Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in Neural Information Processing Systems, 36, 21558-21572.
5. Zhou, S., Alon, U., Agarwal, S., & Neubig, G. (2023). Codebertscore: Evaluating code generation with pretrained models of code. arXiv preprint arXiv:2302.05527.
6. Tong, W., & Zhang, T. (2024). CodeJudge: Evaluating Code Generation with Large Language Models. arXiv preprint arXiv:2410.02184.
7. Siam, M. K., Gu, H., & Cheng, J. Q. (2024, October). Programming with ai: Evaluating chatgpt, gemini, alphacode, and github copilot for programmers. In Proceedings of the 3rd International Conference on Computing Advancements (pp. 346-354).
8. Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt. arXiv preprint arXiv:2304.10778.
9. Yan, D., Gao, Z., & Liu, Z. (2023, September). A closer look at different difficulty levels code generation abilities of chatgpt. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1887-1898). IEEE.