Improving the Coverage of GPT for Automated Feedback on High School Programming Assignments

Shubham Sahai, Umair Z. Ahmed, Ben Leong
National University of Singapore
{shubham, umair}@nus.edu.sg, benleong@comp.nus.edu.sg

Abstract

Feedback for incorrect code is important for novice learners of programming. Automated Program Repair (APR) tools have previously been applied to generate feedback for the mistakes made in introductory programming classes. Large Language Models (LLMs) have emerged as an attractive alternate to automatic feedback generation since they have been shown to excel at generating both human-readable text as well as code. In this paper, we compare the effectiveness of LLMs to APR techniques for code repair and feedback generation in the context of high school Python programming assignments, by evaluating both APR and LLMs on a diverse dataset comprising 366 incorrect submissions for a set of 69 problems with varying complexity from a public high school. We show that LLMs are more effective at generating repair than APR techniques, if provided with a good evaluation oracle. While the state-of-the-art GPTs are able to generate feedback for buggy code most of the time, the direct invocation of such LLMs still suffer from some shortcomings. In particular, GPT-4 can fail to detect up to 16% of the bugs, gives invalid feedback around 8% of the time, and *hallucinates* about 5% of the time. We show that a new architecture that invokes GPT using a conversational interactive loop can improve the repair coverage of GPT-3.5T from 64.8% to 74.9%, at par with the performance of the state-of-the-art LLM GPT-4. Similarly, the coverage of GPT-4 can be further improved from 74.9% to 88.5% with the same methodology within 5 iterations.

1 Introduction

Introductory programming courses are among the most popular courses but they often suffer from a high dropout rate, sometimes ranging up to 60% for some institutions [31]. One of the possible reasons for this is the lack of adequate support for novice students when they encounter programming errors [21, 16]. In an ideal world, every student should be personally mentored by a teaching assistant who can intervene when they run into difficulty. However, this is impractical and infeasible given the severe shortage of computer science instructors [33, 13].

A common approach to learning support for students is the use of test cases to evaluate students' code and identify failing input-output pairs. This is widely adopted in classrooms, MOOCs, online judges, and training websites such as LeetCode [30, 19], but it is not sufficient for novice learners.

Recently, it has been shown that Automated Program Repair (APR) tools can be applied to automatically repair a majority of the mistakes made by learners in introductory programming classes [10, 2, 12]. However, the feedback generated by existing APR tools is often not suitable to be used directly as feedback to students, and a rule-based system or human intervention is needed to translate the repair into pedagogically-sound feedback.

NeurIPS'23 Workshop on Generative AI for Education (GAIED).

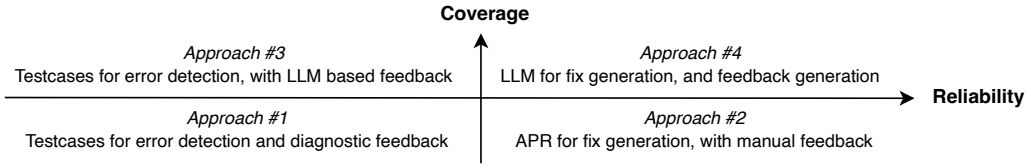

Figure 1: Comparative analysis of approaches for feedback reliability and coverage

Generative AI, driven by recent breakthroughs in Large Language Models (LLMs), offers yet another plausible approach since LLMs excel at generating both human-readable text as well as code [38, 32]. In particular, Large Language Models trained on Code (LLMC), like Codex [24], are found to be able to enhance developers' coding efficiency and bug detection [6, 28, 29].

In this paper, we compare the effectiveness of a classic APR-based approach to the repair generated by OpenAI GPT-3.5T [8] and GPT-4 [9] for programming problems from a large public high school. We show that while APR-based techniques are guaranteed to generate correct feedback, they often have relatively low coverage; on the other hand, GPT-3.5T and GPT-4 are able to generate good feedback most of the time, but they can often suffer from hallucination [20]. In particular, we investigate the following 2 questions:

1. Coverage: How do LLMs compare with existing state-of-the-art APR techniques in generating repair for incorrect submissions for high-school student assignments?
2. Reliability: Is the feedback generated by LLMs trustworthy and correct?

We show that if provided with a good evaluation oracle, LLMs are significantly more effective than classic APR techniques at generating good feedback. However, LLMs, specifically GPT-3.5T and GPT-4, currently suffer from the following shortcomings: (i) failure to detect bugs; (ii) invalid feedback; and (iii) potential hallucination.

2 Related Work

Generating good teaching feedback is harder than repairing incorrect code because the goal is to guide students to learn how to write correct code and not to give them the answers directly. In general, we can quantify the effectiveness of an approach with two attributes: *coverage* and *reliability*.

Given a piece of incorrect code, *coverage* is the probability that an approach is able to identify and generate feedback for the mistake; reliability is a measure of correctness, i.e. given the feedback that is generated for a piece of incorrect code, it is the probability that the feedback is correct and relevant, and hence useful for the learner. In this light, previous work on automated feedback generation can be placed within the quadrant space show in Figure 1.

Approach 1: Use of test cases for error detection and diagnostic feedback. The simplest feedback that can be given to students are the results for test cases coupled with compilation errors. This can help detect failure conditions in code but does not to pinpoint the specific reasons for failure or provide actionable guidance, making them particularly challenging for novice programmers [5]. Rule-based approaches have been proposed to enhance the quality of test cases and compilation errors [4], but these approaches lack generalizability for new class of mistakes.

Approach 2: Use of APR for fix generation, with manual feedback. APR tools can automatically identify the mistake and fix incorrect student code with high reliability since they generally employ an internal evaluation oracle that guides their repair [7, 22]. APR tools for assignments typically work by comparing the buggy student code against given reference solutions, to identify and fix the mistakes with the help of a constraint solver. These repairs have been shown to improve the efficiency by providing students with partial repairs as hints, and teachers with complete repairs for grading purposes [36]. Clara [10], a semantic APR tool for assignments, uses Integer Linear Program (ILP) solver to identify mistakes in incorrect student code and borrow repairs from multiple reference solutions. BIFI [35], the state-of-the-art tool for repairing syntax errors in introductory Python programs, trains encoder-decoder transformers on massive amounts of buggy-repaired data pairs generated with the help of a breaker-critic combination.

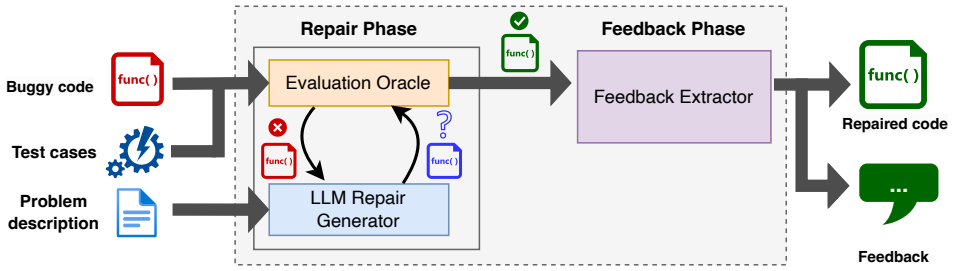

Figure 2: Two phase process for program repair and feedback generation. In the **repair phase**, LLM generates a repair and its corresponding feedback iteratively, using test case validation feedback from an evaluation oracle and problem description. Subsequently in the **feedback phase**, valid feedback is extracted within the context of original buggy code.

However, these approaches often suffer from two limitations: (a) directly revealing the repaired code as feedback for novice programmers does not improve conceptual understanding [1]; (b) moreover, as we will verify in our experiments in §4.1 (see Table 1), APR tools in practice only target a subset of incorrect programming assignments due to a variety of reasons [2, 37, 10, 36].

Approach 3: Use of test cases for error detection, and LLM for feedback generation. The advent of Large Language Models for Code (LLMCs) such as GPT [8, 9] offers a straightforward integration for providing feedback to novice programmers. Phung et al. [26] benchmarked GPT-3.5T and GPT-4 on 25 introductory python programming and reported that the performance of GPT-4 was close to that of human tutors, while struggling in certain scenarios such as grading feedback and task synthesis. Kiesler et al. [15] investigate the ChatGPT-3.5 responses to 13 incorrect student solutions and show that it can provide incorrect or even misleading information, which can potentially confuse the student. Hellas et al. [11] characterize the GPT-3.5T responses to 150 sample help requests from an online programming course, and highlight that while it can identify one or more valid mistake in 82% of the cases, it hallucinates in 48% of the cases by identifying non-existent issues.

To the best our knowledge, we are the first to quantify the effectiveness of state-of-the-art LLMs GPT-4 and GPT-3.5T with manual evaluation by human experts on a large dataset of 366 incorrect high-school Python programs. In §4.3, we demonstrate that because GPT does not have any internal evaluation mechanisms, they can sometimes generate incorrect or even hallucinatory responses, potentially creating confusion for novice learners.

Approach 4: Use of LLM for fix generation and feedback generation. The effectiveness of LLMCs in *last-mile* program repair for tasks such as error localization and code transformation have been verified experimentally [14]. Moreover, the use of unified syntax and semantic LLMC-based repair has been demonstrated to achieve superior repair coverage with smaller fixes when compared to state-of-the-art traditional APR tools [37]. Xia and Zhang [34] propose a conversational automated program repair, where the LLM is tasked with repeatedly generating a repaired code until it passes all the test cases, with the help of test case validation result provided to LLM in each turn.

In light of this, our proposed solution involves a modified setup where we utilize LLMs as both Automated Program Repair (APR) and feedback engines. This approach entails generating repaired code alongside feedback, and repeatedly validating the repair against a testcase evaluation oracle. In §4.2, we show that we can improve trust in LLM-generated output by effectively filtering out incorrect suggestions while simultaneously improving repair coverage.

3 Experimental Setup

We organize the task into two phases: a Repair phase and a Feedback phase. Instead of using existing APR techniques, we use an LLM during the Repair phase to generate both the repaired code for buggy submission and student consumable feedback - a short natural language text that describes the mistake and its fix. The generated repair is validated using an oracle that evaluates it against instructor defined testcases. Our key insight is that despite the initial repair failure, a conversational interaction with LLM by systematically revealing the failing testcases in each iteration, could potentially generate a repair that passes all the test cases. Once a valid repair is obtained, the corresponding, hopefully

trustworthy and correct, feedback can be utilized as hints for novice programmers or for grading support by teaching assistants. This system architecture is depicted in Figure 2.

Dataset: Our full dataset consists of $6,294$ student submissions for 73 diverse Python programming assignments used at the *NUS High School, Singapore*. Among these, there were 366 incorrect submissions from a subset of 69 assignments. These assignments cover a wide range of topics from basic *input-output* to advanced tasks involving nested-loops and functions for manipulating lists and strings. In order to facilitate further research and independent verification, we are publicly releasing the anonymized dataset [3].

Tools: In this paper, we use Clara [10], a popular APR tool for programming assignment, as the baseline. Clara generates feedback by first clustering the students' correct attempts in an offline phase, followed by repairing incorrect attempts by aligning their Control Flow Graph (CFG) with a closest matching correct attempt in the online phase, and borrowing patch ingredients from it. Clara includes an in-built oracle which evaluates the fitness of its repair against provided test-suite. The human consumable feedback generated by Clara is limited to *Insert*, *Delete* and *Replace* code suggestions.

To evaluate the effectiveness of an LLM, we evaluate GPT-3.5T [8] and GPT-4 [9] by OpenAI. GPT-3.5T is the LLM powering the free version of the widely popular ChatGPT application. It utilizes Codex which is trained on open-source Github repositories written by professional developers to identify bugs and provide intricate feedback. GPT-4 is the state-of-art LLM by OpenAI that exhibits human level performance on various academic benchmarks [23], and is shown to outperform competing LLMs by a significant margin on a wide variety of tasks, including programming [38, 26].

Parameters: To run our experiments, a timeout of 5 minutes per invocation was chosen to generate repair for all the three tools. We note that the average time taken by Clara, GPT-3.5T and GPT-4 in returning a response for our context is 2 seconds, 6 seconds, and 32 seconds respectively. Notably, Clara is known to work best when it has access to multiple reference solutions for each problem. This requirement was met by leveraging the $5,928$ *correct* student submissions from our dataset.

In context of GPT-based evaluations, our input prompt includes the problem description, buggy student code, and the set of both failing and passing test cases. The task for the model was specified as generating both the repaired code and accompanying feedback for the student. Furthermore, to evaluate the impact of interaction on the models output, we devised an *iterative prompt*, which contains the failing and passing test cases for the repaired code, in conjunction with the problem statement and the initial student code. The detailed prompts are provided in Appendix A. Finally, we opted for a temperature value of 0.3 to further constrain the generated responses to our specification and retained all remaining OpenAI API parameters at their default values.

4 Evaluation

In this section, we present our findings on the coverage and correctness of the repairs and feedback generated by both APR tools and GPT.

4.1 Coverage: How do LLMs compare with APR tools for high-school student assignments?

To compare the repair coverage of Large Language Models (LLMs) to APR tools within the context of high school student assignments, we compare Clara [10] to GPT-3.5T [8] and GPT-4 [9] on our dataset of 366 buggy programs. The results are presented in Table 1.

We see in Table 1 that Clara successfully repaired only 16.1% of the 366 buggy student programs. For 40.4% of the cases, Clara encountered operations such as lambda functions and external library invocations, which are currently not supported by its implementation. A structural-mismatch was observed in 32.2% of cases, where it struggled to align the buggy program's control-flow structure with a corresponding correct program, and hence failed to initiate repair. Finally, 11.2% of the buggy programs contained syntax errors which are beyond Clara's scope. It should be noted that since this is a research prototype, not all features are fully implemented. We estimate that with sufficient implementation effort, its repair coverage can potentially be increased to more than 56%.

In contrast, we see that both GPT-3.5T and GPT-4 are remarkably successful. GPT-3.5T was able to repair 64.8% of the 366 buggy programs, addressing both syntax and logical errors. GPT-4 performed even better and achieving a repair coverage rate of 74.9%. Unlike traditional APR tools, GPT-3.5T

Table 1: Repair coverage comparison of APR tool Clara [10] with GPT-3.5T and GPT-4 LLMs on our dataset of 366 buggy high-school Python programming assignments. The number in parenthesis is the number of buggy submissions.

Category	**Clara**		**GPT-3.5T**		**GPT-4**	
Successful Repair	16.1%	(59)	64.8%	(237)	74.9%	(274)
Unsupported operations	40.4%	(148)	-		-	
Structural mismatch	32.2%	(118)	-		-	
Syntax errors	11.2%	(41)	-		-	
Invalid repair	-		15.0%	(55)	22.1%	(81)
Invalid output format	-		20.2%	(74)	3.0%	(11)

Table 2: Repair@k coverage metric for GPT-3.5T and GPT-4, i.e. repair success rate after k iterations.

Model	Repair@1	Repair@2	Repair@3	Repair@4	Repair@5
GPT-3.5T	64.8%	71.3%	73.5%	74.6%	74.9%
GPT-4	74.9%	84.2%	86.1%	88.0%	88.5%

and GPT-4 are largely unaffected by issues related to "Unsupported operations", "Structural mismatch" or "Syntax errors", likely because of extensive pre-training on massive amounts of similar data.

The unsuccessful repairs observed for the LLMs were one of two cases: (i) unlike APR tools, the repairs suggested by LLMs can fail at the test-case evaluation stage; (ii) the GPT output format may deviate from the guidelines provided in our prompt. The latter could potentially be avoided by using a framework such as LangChain [18], which remains as future work.

4.2 Improving Coverage: Reducing LLM repair failure using a conversational interaction

Given that GPT-4 was found to be unable to generate valid repair about 25% of the time, we attempted to improve the repair performance with a conversational interaction, as described in the repair-phase of our architecture in Figure 2. Instead of a single input prompt, we use a multi-prompt approach where there is a dynamic exchange of messages in the event of a repair failure. This is achieved by executing the GPT-generated repair code, and automatically prompting it again by revealing the failing test cases in a new and separate prompt.

In Table 2, we present the success rate after k iterations for our dataset of 366 buggy programs. Our findings reveal that the GPT-3.5T repair coverage can be improved from 64.8% to 74.9% within five iterations. In contrast, GPT-4 repair coverage will improve from 74.9 % to 88.5 % after five iterations. In other words, the repair coverage of a relatively weaker LLM like GPT-3.5T could potentially match that of a $10\times$ more powerful [23, 38, 17] and $20\times$ expensive [25] LLM, within 5 iterations using a multi-prompt approach. Currently we are restricted to running a maximum of five iterations due to token limit constraints with our current methodology. It is worth noting that this issue can be overcome by using LLM frameworks like LangChain [18].

4.3 Reliability: Is LLM-generated feedback trustworthy and correct?

To assess the reliability of the LLM-generated feedback, we manually categorized the GPT generated feedback, or lack thereof, into five categories: (a) True Positive (feedback suggested by LLMs that are valid), (b) False Negative (actual errors that LLMs failed to detect or repair correctly), (c) False Positive *Extra* (feedback suggested that is unnecessary); (d) False Positive *Invalid* (feedback generated is incorrect and does not work); and (e) False Positive *Hallucination* (feedback that is "fabricated" and completely unrelated to the code's actual errors). Ideally, we would want all feedback generated to fall into the True Positive category. Our manually annotated feedback quality is publicly released [3]. Figure 3 provides an example illustrating these feedback categories.

In Table 3, we present the counts of these quality categories for the initial feedback generated by GPT-3.5T and GPT-4 models on our dataset of 366 buggy programs. Each incorrect program can receive multiple feedback items corresponding to the various errors and required fixes. Notably, GPT-3.5T yields a similar range of True Positives (TP), False Negatives (FN), and False Positives

```
1  def convert16Bit(decimal):
2      s = '{0:b}'.format(decimal)
3      s = s.zfill(16)
4      print(s)
```

#	Category	Feedback
1	False Positive (Hallucination)	The function definition is missing a colon (:) at the end
4	True Positive	Instead of printing the result, you should return it.

(a) Feedback by GPT-3.5T

#	Category	Feedback
2	False Positive (Extra)	Use "0:016b" instead of "0:b". This will automatically pad the binary number with leading zeros to make it 16 bits.
4	True Positive	In Python, functions should return their result for use elsewhere in the program. Replace "print(s)" with "return s"

(b) Feedback by GPT-4

Figure 3: Example buggy-program and their corresponding feedback by GPT-3.5T and GPT-4.

Table 3: Confusion Matrix of the feedback generated by GPT-3.5T and GPT-4 in the first iteration.

		GPT-3.5T		GPT-4	
		Positive	Negative	Positive	Negative
Actual	Positive	345 (TP)	310 (FN)	656 (TP)	125 (FN)
	Negative	327 (FP)	—- (TN)	254 (FP)	—- (TN)

(FP). In comparison, GPT-4 is significantly more effective at addressing a greater number of errors, as evident from the much higher TP rate and fewer False Positives.

In Table 4, we compare the performance of GPT-3.5T and GPT-4 in the initial iteration and after multiple iterations. We observe that, we can improve the feedback recall and precision marginally after multiple conversational iterations ($k \leq 5$) for both GPT-3.5T and GPT-4. This suggests that the generated feedback became more accurate and relevant to the code errors with more iterations.

GPT-4 achieves a Recall (True Positive Rate) of 84.0% in the first iteration, which is about 30% higher than GPT-3.5T. GPT4 also achieves a significantly higher precision (Positive Predictive Value) of 72.0% than the 51.2% of GPT-3.5T. However, both models are susceptible to hallucination and generated *extra false positives*, including optimization suggestions. The invalid and hallucination are of greater concern as they could potentially confuse students. Notably, GPT-3.5T had a significantly higher false positive rate of 20.8% compared to GPT-4. Nevertheless, a 4.3% hallucination rate and 9.0% invalid feedback for GPT-4 is still a cause for concern. In teaching, we want to *do no harm*.

5 Discussion

We evaluated the efficacy of GPT-3.5T and GPT-4 in generating automated feedback for incorrect programming assignment submissions. We observe that even in cases where LLMs could not generate a complete repair, its feedback can accurately address majority of the mistakes in student's code. Studies have shown that such partial feedback can be used to improve the performance of programming students and their teaching assistants [36]. We note that the quality of generated feedback is more nuanced than simple correctness. While two pieces of feedback may both be correct for the same incorrect code, one may prove more effective in helping students learn. The assessment of the natural language quality of the feedback falls outside the the scope of this paper and has been explored in prior research [23, 26], and we focus solely on correctness.

Conventional APR tools, like Clara [10], use constraint solvers to determine a minimal set of repairs that are necessary for the program to pass the given test-suite. LLMs do not have an internal interpreter, which makes them vulnerable to the generation of erroneous or even fictitious feedback. Verification techniques such as Verifix [2] employ Satisfiability Modulo Theories (SMT) solvers to generate verifiably correct repairs. In comparison, we make use of a relatively weaker testcase evaluation oracle to validate the repairs generated by LLMs, which is susceptible to testcase overfitting. However, we note that the repairs generated by LLMs can be further subjected to a similar verification by SMT solvers for additional guarantees.

Phung et al. had propose a framework where state-of-art LLMs such as GPT-4 are used to generate feedback through multiple iterations until it is validated successfully using weaker LLM like GPT-

Table 4: Feedback quality of GPT-3.5T and GPT-4 LLMs, based on manual assessment by authors. The *False Positive* cases are further categorized into *Extra*, *Invalid*, and *Hallucination*.

Iteration	Model	Recall (TPR)	Miss (FNR)	Precision (PPV)	False Positive Rate (FPR) Extra	Invalid	Hallucination
Single	**GPT-3.5T**	52.7%	47.3%	51.2%	15.7%	15.0%	18.0%
Single	**GPT-4**	84.0%	16.0%	72.0%	14.8%	9.0%	4.1%
Multiple	**GPT-3.5T**	53.1%	46.9%	51.4%	15.2%	16.5%	16.9%
Multiple	**GPT-4**	87.2%	12.8%	72.4%	14.4%	7.7%	5.4%

3.5T [27]. Using this technique, the authors report a substantial improvement in precision that is comparable to human tutors on a dataset of 73 buggy programs, viz trade-off in coverage. In comparison, we use multiple conversational interaction with the same GPT-4 model to improve the repair coverage by evaluating against test case oracle. Human experts were used to validate the quality of LLM feedback on a sizeable dataset of 366 incorrect Python programs, leading to increased reliability of our results.

Limitations. The output of LLMs is inherently probabilistic, this means that we cannot be certain that LLMs will consistently generate correct outputs even if they are shown to do so in our experiments. Furthermore, the human evaluation of the descriptive feedback is somewhat subjective. To allow independent verification and ensure transparency, we have made the GPT-generated feedback along with our manual annotations publicly available [3]. Additionally, GPT models are likely trained on programming assignments commonly encountered in high school curriculum, which means their performance may vary on new unseen problems. As such, the results observed in our evaluation of high-school assignments may not necessarily extend to more complex assignments, or problems from CS1 introductory programming courses at universities.

Future Work. In this paper, we focus on evaluating the correctness of both the repaired code and descriptive feedback generated by state-of-the-art LLMs. Assessing the quality of feedback across more complex attributes, such as informativeness and comprehensibility, remains as future work. Furthermore, conducting a large-scale user-study to evaluate its real-world usability, in terms of pedagogical effectiveness on student's learning outcomes and teacher's grading process, is planned for a future investigation.

6 Conclusion

To the best of our knowledge, we are the first to comprehensively compare the effectiveness of Large Language Models (LLMs) with traditional Automated Program Repair (APR) techniques in code repair and feedback generation, using an extensive dataset of 366 incorrect high school Python programs. We have demonstrated that LLMs significantly outperform traditional APR methods in terms of repair coverage. In particular, the feedback generated by GPT-4 is capable of addressing 84.0% of student errors with 72.0% precision. To further enhance the coverage and ensure reliability, we leveraged a conversational interactive loop that validates LLM generated repairs against an evaluation oracle.

While our system is a work in progress, we believe that our preliminary results are promising and can benefit the estimated millions of high-school programming learners. What is particularly exciting is that with the right prompts and oracle, even a "weaker" LLM like GPT-3.5T can achieve the same repair coverage as GPT-4, which is 20x more expensive. Beyond improving performance, our approach could potentially achieve significant savings.

Acknowledgement

We express our gratitude to NUS High School for providing us with access to their anonymized dataset, which was instrumental in conducting our experiments. Additionally, we would like to thank the anonymous reviewers for their valuable feedback and helpful comments. This research/project is supported by the National Research Foundation Singapore under the AI Singapore Programme (AISG Award No: AISG2-TC-2023-009-AICET).

References

[1] Umair Z Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, pages 139–150, 2020.

[2] Umair Z Ahmed, Zhiyu Fan, Jooyong Yi, Omar I Al-Bataineh, and Abhik Roychoudhury. Verifix: Verified repair of programming assignments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(4):1–31, 2022.

[3] AI Centre for Educational Technologies (AICET). High-school programming assignments dataset and experimental results. https://github.com/ai-cet/neurips-gaied-2023.

[4] Brett A Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 126–131, 2016.

[5] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education*, pages 177–210, 2019.

[6] Robert W Brennan and Jonathan Lesage. Exploring the implications of openai codex on education for industry 4.0. In *International Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*, pages 254–266. Springer, 2022.

[7] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62, 2019.

[8] OpenAI GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.

[9] OpenAI GPT-4. https://platform.openai.com/docs/models/gpt-4.

[10] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 465–480, 2018.

[11] Arto Hellas, Juho Leinonen, Sami Sarsa, Charles Koutcheme, Lilja Kujanpää, and Juha Sorva. Exploring the responses of large language models to beginner programmers' help requests. *arXiv preprint arXiv:2306.05715*, 2023.

[12] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 388–398. IEEE, 2019.

[13] Richard M Ingersoll. The teacher shortage: A case of wrong diagnosis and wrong prescription. *NASSP bulletin*, 86(631):16–31, 2002.

[14] Harshit Joshi, José Cambronero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radiček. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 5131–5140, 2023.

[15] Natalie Kiesler, Dominic Lohr, and Hieke Keuning. Exploring the potential of large language models to generate formative programming feedback. *arXiv preprint arXiv:2309.00029*, 2023.

[16] Päivi Kinnunen and Lauri Malmi. Why students drop out cs1 course? In *Proceedings of the second international workshop on Computing education research*, pages 97–108, 2006.

[17] Anis Koubaa. Gpt-4 vs. gpt-3.5: A concise showdown. 2023.

[18] LangChain. https://www.langchain.com.

[19] LeetCode. https://leetcode.com.

[20] Nick McKenna, Tianyi Li, Liang Cheng, Mohammad Javad Hosseini, Mark Johnson, and Mark Steedman. Sources of hallucination by large language models on inference tasks, 2023.

[21] Rodrigo Pessoa Medeiros, Geber Lisboa Ramalho, and Taciana Pontual Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2019. doi: 10.1109/TE.2018.2864133.

[22] Martin Monperrus. The living review on automated program repair. Technical Report hal-01956501, HAL Archives Ouvertes, 2018. URL `https://www.monperrus.net/martin/repair-living-review.pdf`.

[23] OpenAI. GPT-4 Technical Report, 2023.

[24] OpenAI Codex. https://openai.com/blog/openai-codex.

[25] OpenAI Pricing. https://openai.com/pricing.

[26] Tung Phung, Victor-Alexandru Padurean, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generative AI for programming education: Benchmarking chatgpt, gpt-4, and human tutors. *CoRR*, abs/2306.17156, 2023. doi: 10.48550/arXiv.2306.17156. URL `https://doi.org/10.48550/arXiv.2306.17156`.

[27] Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation. *arXiv preprint arXiv:2310.03780*, 2023.

[28] Julian Aron Prenner and Romain Robbes. Automatic program repair with openai's codex: Evaluating quixbugs. *arXiv preprint arXiv:2111.03922*, 2021.

[29] Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai's codex fix bugs? an evaluation on quixbugs. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 69–75, 2022.

[30] Coursera programming assignment grading. https://www.coursera.support/s/article/209818753-Programming-assignments.

[31] Anthony Robins. Learning edge momentum: A new account of outcomes in cs1. *Computer Science Education*, 20(1):37–71, 2010.

[32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[33] Esther Shein. The cs teacher shortage. *Communications of the ACM*, 62(10):17–18, 2019.

[34] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[35] Michihiro Yasunaga and Percy Liang. Break-it-fix-it: Unsupervised learning for program repair. In *International Conference on Machine Learning*, pages 11941–11952. PMLR, 2021.

[36] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 740–751, 2017.

[37] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing bugs in python assignments using large language models. *arXiv preprint arXiv:2209.14876*, 2022.

[38] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *arXiv preprint arXiv:2306.05685*, 2023.

A Prompts to GPT for generating repair and feedback

````
The student has made some mistakes in his python program, and it is
failing a couple of test cases. Can you go through the incorrect
program  below and try to fix it with as few changes as possible.
I have provide the description of the programming task, student's
buggy code and the list of passing test cases and failing test cases
below:

Problem Description: ###
    {problem_description}
###

Student's Buggy Code: ###
    {student_code}
###

Passing Test Case: ###
    {passing_test_cases}
###

Failing Test Case: ###
    {failing_test_cases}
###

Can you fix the above buggy code, such that it passes all the test cases.
Please try to make as few changes as possible. Please output STRICTLY
in the following format. Provide the fixed code between delimiters:

```python

```

Additionally, please provide the feedback to the student as a JSON
array sticking to the following format:

```json
[
    {
    "line_number":  line number where the mistake occurs,
    "feedback": the feedback you would like to give the student, to
    fix the mistake.
    },
]
```
````

Figure 4: *Initial Prompt* for generating the repaired code and feedback. The prompt has 4 placeholders for problem description, buggy student code, list of passing and failing test cases. We expect LLM to generate a repaired code along with the feedback in JSON format.

````
You have not successfully generated the repaired code. The generated
code is still failing some test cases. Please find the passing test
cases and the failing test cases on your repaired code below.

Passing Test Case: ###
    {passing_test_cases}
###

Failing Test Case: ###
    {failing_test_cases}
###

Additionally, please note the original problem description and the
original student buggy code below.

Problem Description: ###
    {problem_description}
###

Student's Buggy Code: ###
    {student_code}
###

Can you fix the above buggy code, such that it passes all the testcases.
Please try to make as few changes as possible. Please output STRICTLY
in the following format. Provide the fixed code between delimiters:

```python

```

Additionally, please provide the feedback to the student as a JSON
array sticking to the following format:

```json
[
    {
    "line_number":  line number where the mistake occurs,
    "feedback": the feedback you would like to give the student, to
    fix the mistake.
    },
]
```
````

Figure 5: *Iterative prompt* for "chatting" with LLM, in case an incorrect fix is generated. The prompt loops back with LLM, providing the list of passing and failing test cases for the generated repair. While maintaining the chat history, we also provide the problem description and buggy student code as part of the iteration prompt.