Benchmarking Educational Program Repair

Charles Koutcheme
Aalto University
Espoo, Finland
charles.koutcheme@aalto.fi

Nicola Dainese
Aalto University
Espoo, Finland
nicola.dainese@aalto.fi

Sami Sarsa
Aalto University
Espoo, Finland
sami.sarsa@aalto.fi

Juho Leinonen
University of Auckland
Auckland, New Zealand
juho.leinonen@auckland.ac.nz

Arto Hellas
Aalto University
Espoo, Finland
arto.hellas@aalto.fi

Paul Denny
University of Auckland
Auckland, New Zealand
paul@cs.auckland.ac.nz

Abstract

The emergence of large language models (LLMs) has sparked enormous interest due to their potential application across a range of educational tasks. For example, recent work in programming education has used LLMs to generate learning resources, improve error messages, and provide feedback on code. However, one factor that limits progress within the field is that much of the research uses bespoke datasets and different evaluation metrics, making direct comparisons between results unreliable. Thus, there is a pressing need for standardization and benchmarks that facilitate the equitable comparison of competing approaches. One task where LLMs show great promise is program repair, which can be used to provide debugging support and next-step hints to students. In this article, we propose a novel educational program repair benchmark. We curate two high-quality publicly available programming datasets, present a unified evaluation procedure introducing a novel evaluation metric rouge@k for approximating the quality of repairs, and evaluate a set of five recent models to establish baseline performance.

1 Introduction

In education, the emergence of large language models (LLMs) has led to a plethora of research evaluating their performance on educationally relevant tasks [1, 2]. LLMs have been explored for educational tasks such as providing automatic feedback to students [3, 4, 5], generating novel exercises [6], and producing novice-friendly explanations of source code [7, 8]. Within computing education, one long-standing challenge has been generating automated feedback on students' programs [9]. This has traditionally been achieved using unit-test-based automated assessment systems [10, 11], or intelligent tutoring systems that provide hints on the next steps that students should take [12, 13, 14]. A key parallel stream to this research is 'automated program repair', which employs rule-based and machine-learning-based strategies for automatically fixing bugs in code [15, 16, 17, 18, 19]. Drawing from this extensive body of prior research, we recognize automated program repair as a crucial intermediary step toward enhancing the accuracy and reliability of feedback for novice programmers.

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

Currently, substantial interest lies in harnessing LLMs for generating feedback based on program repairs [20, 15, 16].

In general, much of the prior work exploring LLMs in education has used bespoke datasets due to the lack of widely used benchmarks. This makes it difficult to meaningfully compare results and identify effective techniques. Benchmarks are an essential tool in computer science and software engineering, including machine learning. They provide a standardized method for comparing various techniques and for monitoring progress within an area over time, driving innovation and improvement. Datasets in particular have served as benchmarks for comparing algorithms in a reproducible way which is the bedrock of modern science. For example, within the field of machine learning standard benchmarks such as ImageNet [21], MNIST [22] and HumanEval [23] have been crucial for identifying breakthroughs in algorithms for image recognition and program synthesis.

In this paper, we propose a framework for evaluating program repair techniques using LLMs. By providing a benchmark for this task, we hope to facilitate future research by allowing easy replication of results and evaluation of new models using the same data that older models have been evaluated with. In particular, this work makes the following contributions: we (1) formalize the task of educational program repair, (2) describe an evaluation procedure including a novel metric for the quality of program repairs called rouge@k, (3) curate two high-quality and publicly available datasets suitable for serving as a benchmark to evaluate program repairs, and (4) report the performance of several recent decoder-only transformer models on these datasets.

2 Related Work

Benchmarking program synthesis. In assessing the capabilities of code language models across various contexts, programming benchmarks have become a common tool [23, 24, 25, 26]. However, when it comes to program synthesis, few of these benchmarks offer a large number of problems that are suitable for introductory programming courses [24, 25]. While code language models have traditionally been used in industry and research, the emergence of highly proficient easily accessible language models such as ChatGPT has sparked interest in Computing Education Research (CER). In line with program synthesis principles, extensive research in CER has explored these models' abilities to solve real-world introductory programming problems under authentic course conditions, moving beyond artificial examples found in standard code benchmarks [27, 28, 29, 30, 31]. We notice a similar transition happening with program repair.

Benchmarking program repair. The task of program repair is often approached as a synthesis problem. In our study, we draw inspiration from this paradigm to design components of our evaluation procedure. Similarly to program synthesis, there exist program refinement benchmarks [32, 33] and particularly close to our work is a recent extension [34] to the human evaluation benchmark [23] that introduces bugs in code for LLMs to rectify. Although the bugs in all these datasets are relevant to student programming, they do not fully encapsulate the complexities of errors encountered in student-written code, which can present a wide array of issues extending beyond mere bugs. Consequently, while such refinement benchmarks assume the presence of ground truth annotations, our evaluation procedure operates without assuming the existence of a single ground truth repair, and, in fact, did not necessitate manual annotations.

Our work also builds upon recent efforts utilizing large language models to provide feedback and repair programs for students. Notably, LLMs have shown great promise in fixing syntax errors [17, 18, 19, 35] and rectifying bugs in student programs [20], and preliminary work exists on using them to automatically generate feedback [5, 4, 3, 36, 37]. While most of these endeavours leverage closed-source systems such as OpenAI Codex, some explore the use of open-language models [38, 16]. Many of the methods we introduce align with these previous efforts. In particular, our proposed evaluation procedure aims to encompass the different scenarios explored in this prior research while leveraging recent insights into evaluation methodologies [39]. Although some prior work has integrated human judgment into the evaluation [40, 35], we focus on a fully automated evaluation approach which is more scalable, primarily relying on program correctness and NLP scoring metrics as proxies for quality. In addition to presenting the evaluation procedure, we also identify two publicly available datasets that we curate in such a way that they are suitable as benchmarks for evaluating program repair.

3 Educational Program Repair

In this section, we formalize the educational program repair task for large language models and present an evaluation procedure for that task (Figure 1 shows an overview). We also introduce two educational scenarios of interest and propose two high-quality publicly available datasets that match these scenarios.

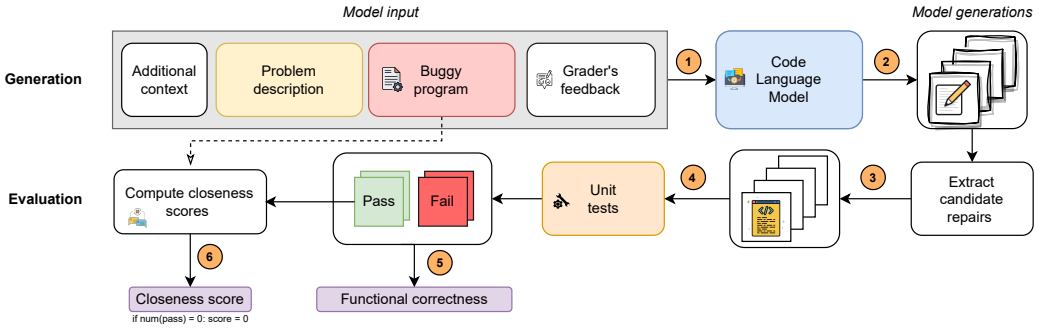

Figure 1: **Overview of the program repair task and its evaluation procedure.** (1) For each incorrect program, we pass to a language model a prompt which contains the programming task, and the incorrect student code. In addition, the prompt may contain more contextual information, e.g. grader's feedback. (2) The model generates multiple feedbacks based on the instructions; each feedback must contain (at least) a full correction to the incorrect program. (3) We extract the full code repairs, which we (4) pass to unit tests. The samples which pass or fail the unit tests allow us to evaluate (5) functional correctness and (6) closeness to the incorrect program.

3.1 Task and Evaluation Procedure

Problem setup and motivations. For this task, we consider a student working on a programming assignment. We have access to the problem description, associated unit tests, and a grader that employs these unit tests to provide summative feedback to students. When presented with an incorrect program that doesn't pass all unit tests, our goal is to utilize a language model to generate a repair to the student's code, addressing all the issues in the incorrect program. In essence, we view student program repair as a program synthesis problem, conditioned upon an existing (non-working) solution [20]. Our primary objective is to ensure that the generated programs are not only functionally correct but also closely aligned with the original incorrect program. This closeness is akin to the preference for hint-generation systems to provide hints that are in proximity to the student's current state of knowledge [12, 13, 14]. The basic assumption is that enforcing this closeness will enhance the comprehensibility of the associated feedback [41], such as hints generated by an Intelligent Tutoring System (ITS), or the natural language explanations generated by an LLM [19].

Evaluation procedure. Generally, we consider each buggy program as an independent "problem", and use our language model to generate one or multiple repairs for each problem. Classically, (educational) program repair has reported performance in terms of the total number of buggy programs successfully repaired (i.e., the total number of problems solved) and the average edit distance between the buggy programs and their associated code repairs [20, 19]. In this work, we generate up to $n = 10$ different repairs and introduce two distinctions to the classical evaluation procedure.

pass@k First, following prior program synthesis research, we assess functional correctness using the pass@k estimator [23]. We first generate n repairs. Then for a given k ($k \leq n$), we generate all combinations of k elements from the n repairs, and count those with at least one repair passing all unit tests. Dividing this count by the total combinations gives the probability of a random sample of k repairs fixing a buggy program. We average this probability for each program in our test set.

rouge@k Second, recent research [39] suggests that the Rouge score [42] serves as a better repair quality indicator compared to sequence edit distance. Building upon this insight, we introduce a novel evaluation metric, 'rouge@k' mirroring 'pass@k'. For each of the passing n model generations,

we calculate Rouge[1] scores with the buggy program (using the buggy program as the reference 'sentence'). Importantly, candidates that fail unit tests receive a score of 0. For some k, we generate all combinations of k scores from the n candidates and average the highest scores from each combination. We repeat this for each buggy program in our test set, and average the values to produce the final rouge@k score.

3.2 Educational Scenarios and Datasets

We propose two versions of the program repair task: a version where prior educational data is available, and one where no prior data is available. For both scenarios, we propose to rely on publicly available real-life datasets collected from various institutions. Due to space limitations, we refer the reader to the original dataset papers (resp. Appendix A.1) for curation (resp. processing) details.

Prior data available. In this scenario, we assume having access to prior data. Specifically, we define "prior" data as a repository of both correct and incorrect solutions submitted by students in preceding iterations of the same programming assignments, typically offered on a semester basis. Automated Program Repair has frequently relied on such prior data to construct repairs for students' buggy programs [43, 44, 45, 46]. In our context, access to prior data offers significant advantages for large language models, whether for few-shot prompting [20] or supervised fine-tuning [38]. Additionally, it facilitates the proper division of our datasets into training, validation/development, and testing subsets. This mirrors a real-life scenario in which educational teams continually enhance their models each semester as they accumulate more data. For this scenario, we propose harnessing the recently released FalconCode dataset [47]. Beyond its substantial scale, this dataset distinguishes itself through the presence of associated unit tests, and the inclusion of pertinent metadata (e.g. assignment difficulty level). Notably, it contains free-form assignments where students are not necessarily required to write functions. This change from the focus of prior work to repair functions [16, 20] allows for a broader evaluation of LLM-based repair techniques. For evaluation purposes, we propose to utilize a smaller curated subset of 1,305 solutions for testing. It's worth noting that code benchmarks with fewer than 1000 problems are common in the field [24, 25, 26, 23], partly due to the computational and/or financial resources required for LLM-based evaluation techniques.

No prior data available. In the second version of the problem, we introduce a setup where models are tasked with fixing programs in a dataset where no prior data is available. This scenario aligns with contexts such as new courses (typically small) or courses where data about students' previous submissions is not retained. To address this challenge, we propose leveraging a dataset collected at the National University of Singapore [43]. This dataset comprises five assignments where students are required to write functions to solve specific tasks involving loops, conditions, and other standard introductory concepts. The reasons for selecting this dataset include an appropriate size, the selection of exercises with intermediate levels of difficulty, the presence of associated unit tests, and a good variety of issues in the submissions. We use a curated subset of 1238 incorrect submissions.

4 Baseline and Experiments

In this section, we present a set of simple baseline models obtained from fine-tuning a variety of open-source pretrained language models of code. We leave the evaluation of instruction-tuned [48] and chat models [49] for future work.

4.1 Models

We experiment with several decoder-only transformer models, using the CodeGen family of models [50] with 350M, and 2B parameters, as well as models from the StarCoder family [51] with 164M, 1B, and 3B parameters. We chose these models for their relatively small size which allows them to fit fully (without quantization) inside a single consumer GPU. Although we could evaluate the repair capabilities of these models using zero-shot and few-shot prompting [17], prior work suggests that their relatively small size (less than several billion parameters) prevents them from fully taking advantage of in-context examples [52]. For this reason, we choose to finetune these models to repair

[1]Rouge embodies a family of metrics. For this study, we use the Rouge-L [42] score.

student programs via supervised learning. Because most educational datasets do not contain ground truth repairs to students' incorrect programs, prior work in neural program repair [53] suggested creating artificial repair examples by mapping each incorrect program in a dataset to the closest correct program submitted to the same assignment. Instead of using the closest correct program, we follow recent work [38] and map each incorrect program to the correction found by an Automated Repair Tool applied to the same dataset. Using an Automated Repair Tool (in our case, Refactory [43]) allows for efficiently selecting a semantically close program while smoothing out many irrelevant syntactic differences (e.g., variable name conventions, needless reformulations) between the incorrect program and the chosen one."

We trained our supervised baselines on the FalconCode dataset with the above-mentioned methodology, using the first two semesters of data as training and development sets (1299 and 1339 distinct programs respectively). Appendix A.2 details our exact training procedure including prompting strategy, and hyperparameter tuning. As an additional contribution, we released the code for conducting our experiments on GitHub[2].

4.2 Experiments and Results

We report the performance of the supervised baselines on the test set of the FalconCode dataset. Following [24], we separate the results for two distinct assignment difficulty levels: "easy" and "hard" (with 767 and 538 incorrect programs respectively). We also report the performance of the models fine-tuned on FalconCode when prompted on the 1238 incorrect submissions of the Singapore dataset. The results for these three scenarios, including pass rates and Rouge scores, are presented in Table 1. We can make the following observations.

Table 1: Functional and closeness results for our baselines models (higher is better).

(a) **Pass@k** for k = 1, 5, 10

		falconcode_easy			falconcode_hard			singapore		
model	size	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
starcoder	164M	6.88	12.46	14.68	20.27	35.93	41.85	2.54	7.62	10.46
codegen-mono	350M	12.08	23.19	27.88	13.68	28.28	33.77	3.72	12.92	18.90
starcoder	1B	16.91	30.38	36.06	**26.44**	**44.94**	**50.72**	7.35	21.02	28.77
codegen-mono	2B	16.10	25.99	30.67	13.85	25.92	30.38	11.24	24.97	31.63
starcoder	3B	**19.11**	**31.68**	**37.73**	14.63	29.88	36.51	**12.29**	**29.37**	**36.14**

(b) **Rouge@k** for k = 1, 5, 10

model	size	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10	k = 1	k = 5	k = 10
starcoder	164M	5.91	10.66	12.57	12.14	23.43	28.32	2.18	6.58	9.07
codegen-mono	350M	10.56	20.14	24.12	9.06	19.32	23.50	2.49	8.66	12.92
starcoder	1B	14.32	25.41	29.93	**17.02**	**30.91**	**35.90**	5.60	15.95	21.68
codegen-mono	2B	13.70	21.91	25.66	10.22	19.14	22.70	7.63	13.59	15.91
starcoder	3B	**15.89**	**26.18**	**31.12**	10.48	21.59	26.54	**7.85**	**19.97**	**25.73**

Model performance scales with training compute and model sizes. Within the same architectural family, performance scales with model sizes and number of generations. However, between different model families, the quality of the pretraining can make smaller models (e.g., starcoder 1B) outperform larger ones (e.g., codegen-mono 2B). The generalization capabilities of language models have been shown to scale smoothly with the amount of computing used during pretraining in proportion with model size and dataset size [54, 55].

Fine-tuned language models can overfit their datasets. Some models demonstrate superior performance on the 'hard' section of the Falcon-code dataset compared to the 'easy' section. We posit that this divergence stems from a training data imbalance, leading to 'overfitting' on more

[2]https://github.com/KoutchemeCharles/gaied_nips23

challenging assignments [56]. Notably, overfitting is more pronounced in smaller models, which may exhibit better performance than larger counterparts on the 'hard' subset (e.g., codegen-mono 350M outperforming Codegen 2B on Falcon-code hard). However, this advantage does not extend to the smaller training subset (FalconCode easy) or to the Singapore dataset.

Supervised training can transfer across datasets. Despite being fine-tuned on the FalconCode dataset, our models managed to fix some of the buggy programs in another dataset featuring distinct problem types and other unique issues within the incorrect programs. Interestingly, beyond 1B parameters, the models reach similar performance on the FalconCode and Singapore datasets.

5 Discussion and Conclusion

We have introduced a benchmark for program repair using LLMs. However, the versatility of language models extends beyond program repair; they are also capable of generating natural language explanations paired with functioning solutions. Evaluating the quality of these explanations poses a challenge in the absence of ground truths or manual assessments. Although we cannot guarantee the accuracy of generated explanations with certainty, we assert that the relative quality of repairs can act as a reliable proxy for the relative quality of associated Natural Language Explanations (NLEs) when these NLEs are generated prior to the repairs. Previous research suggests that, for a given model, repairs of higher quality that are closely aligned with the student's code are more likely to be derived from in-context, high-quality explanations [57]. Conversely, generating natural language explanations from in-context, high-quality repairs [19, 58] – first generating the repair and then generating explanations – reduces the risk of introducing inaccuracies or inventing non-existent issues, thus mitigating the risk of hallucinations [59, 5]. Thus, while our primary focus is on program repair, our work also serves as a step towards improving the feedback that can be provided to students.

While our baseline models provide valuable insights, it's essential to recognize the growing presence of powerful language models capable of program repair without additional training. In the source code modelling landscape, we observe a division between permissive (open-source) and non-permissive (proprietary) models. While prior work in educational program repair primarily focused on using non-permissive models [20, 19, 40, 18, 35], we forecast a potential shift occurring with the emergence of robust open-source alternatives [55, 60]. Historically, non-permissive models such as ChatGPT and OpenAI Codex have demonstrated state-of-the-art performance on benchmarks, but open-source alternatives are beginning to match or surpass them. However, running such powerful permissive instruction-tuned and chat models requires significant computational resources (e.g. custom GPUs) due to their size. This is becoming less of a barrier due to the rise of open-source hosting services [61], the ability of smaller models to achieve high performance [54], and the use of Parameter Efficient Finetuning Techniques (PEFT) [62]. While non-permissive models such as ChatGPT or OpenAI Codex offer high performance without the need for custom hardware, they also raise significant concerns about privacy and potential non-reliability due to model changes [63].

Prior studies in program repair have yielded valuable insights into the application of various methods. However, meaningful comparisons between methods have been hindered by the lack of accessible, high-quality datasets. To overcome this limitation, we have curated the FalconCode and Singapore datasets specifically for program repair tasks relevant to educational settings. Nevertheless, we acknowledge concerns that any individual educational benchmarking dataset can not fully capture the diversity of exercises and instructional approaches present in programming classes [64, 65]. We feel it is important to emphasize that the purpose of these benchmarks should not be to establish a rigid hierarchy of method superiority but to serve as reference points for comparing methods in common scenarios, and for identifying issues in prior studies [66]. We believe that educational AI researchers can benefit from presenting and evaluating their novel techniques on both private datasets and the common benchmark, providing a well-rounded perspective.

Finally, we acknowledge several limitations to our current work. First, both the FalconCode and Singapore datasets are limited to Python and represent data from only two institutions. Furthermore, syntax errors are currently not included due to our curation processes. We are also aware of the potential risk of contamination, a common challenge in code benchmarks [67]. While the FalconCode dataset is unlikely to be part of any model's pretraining dataset (as access requires a manual request via an online form), the Singapore dataset is readily available in a GitHub repository. Lastly, we recognize that assessing qualitative performance based solely on code closeness may introduce biases and may

not always align with instructors' pedagogical objectives, such as teaching specific programming composition methods or emphasizing certain coding approaches (see e.g. [68, 69, 70, 71]). To address these limitations, our future plans include introducing other high-quality datasets that cover a broad range of programming languages and scenarios (e.g. from online MOOC courses). We have also identified datasets suitable for evaluating both syntax and semantic errors, and we are working on annotating a subset of our datasets to provide accessible ground truth annotations [39]. Looking further ahead, we intend to expand our benchmark to encompass a broader range of educational programming tasks [40] (e.g. hint generation), and we aim to establish an open online leaderboard [61] for result tracking. As educational program feedback presents distinctive challenges and real-life constraints, we anticipate cross-domain collaborations between educational program feedback research and pure NLP research to be essential.

References

[1] Enkelejda Kasneci, Kathrin Seßler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günnemann, Eyke Hüllermeier, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and individual differences*, 103:102274, 2023.

[2] Paul Denny, James Prather, Brett A Becker, James Finnie-Ansley, Arto Hellas, Juho Leinonen, Andrew Luxton-Reilly, Brent N Reeves, Eddie Antonio Santos, and Sami Sarsa. Computing education in the era of generative ai. *arXiv preprint arXiv:2306.02608*, 2023.

[3] Maciej Pankiewicz and Ryan S Baker. Large language models (gpt) for automating feedback on programming assignments. *arXiv preprint arXiv:2307.00150*, 2023.

[4] 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.

[5] 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. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 93–105, New York, NY, USA, 2023. Association for Computing Machinery.

[6] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. Automatic generation of programming exercises and code explanations using large language models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*, pages 27–43, 2022.

[7] Stephen MacNeil, Andrew Tran, Arto Hellas, Joanne Kim, Sami Sarsa, Paul Denny, Seth Bernstein, and Juho Leinonen. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 931–937, 2023.

[8] Juho Leinonen, Paul Denny, Stephen MacNeil, Sami Sarsa, Seth Bernstein, Joanne Kim, Andrew Tran, and Arto Hellas. Comparing code explanations created by students and large language models. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2023, page 124–130, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)*, 19(1):1–43, 2018.

[10] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)*, 22(3):1–40, 2022.

[11] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93, 2010.

[12] Sebastian Gross, Bassam Mokbel, Benjamin Paaßen, Barbara Hammer, and Niels Pinkwart. Example-based feedback provision using structured solution spaces. *International Journal of Learning Technology 10*, 9(3):248–280, 2014.

[13] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27:37–64, 2017.

[14] Thomas Price, Rui Zhi, and Tiffany Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. *International Educational Data Mining Society*, 2017.

[15] Julian Aron Prenner and Romain Robbes. Automatic program repair with openai's codex: Evaluating quixbugs, 2021.

[16] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Arto Hellas, and Paul Denny. Automated Program Repair Using Generative Models for Code Infilling. In Ning Wang, Genaro Rebolledo-Mendez, Noboru Matsuda, Olga C. Santos, and Vania Dimitrova, editors, *Artificial Intelligence in Education*, pages 798–803, Cham, 2023. Springer Nature Switzerland.

[17] Harshit Joshi, José Cambronero, Sumit Gulwani, Vu Le, Ivan Radicek, and Gust Verbruggen. Repair is nearly generation: Multilingual program repair with llms, 2022.

[18] Toufique Ahmed, Noah Rose Ledesma, and Premkumar Devanbu. Synfix: Automatically fixing syntax errors using compiler diagnostics, 2022.

[19] Tung Phung, José Cambronero, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. Generating high-precision feedback for programming syntax errors using language models, 2023.

[20] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. Repairing bugs in python assignments using language models, 2022.

[21] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[22] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2, 2010.

[23] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating language models trained on code, 2021.

[24] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[25] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021.

[26] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

[27] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA, 2022. Association for Computing Machinery.

[28] James Finnie-Ansley, Paul Denny, Andrew Luxton-Reilly, Eddie Antonio Santos, James Prather, and Brett A. Becker. My ai wants to know if this will be on the exam: Testing openai's codex on cs2 programming exercises. In *Proceedings of the 25th Australasian Computing Education Conference*, ACE '23, page 97–104, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Paul Denny, Viraj Kumar, and Nasser Giacaman. Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 1136–1142, New York, NY, USA, 2023. Association for Computing Machinery.

[30] Michel Wermelinger. Using github copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 172–178, New York, NY, USA, 2023. Association for Computing Machinery.

[31] Jaromir Savelka, Arav Agarwal, Christopher Bogart, Yifan Song, and Majd Sakr. Can generative pre-trained transformers (gpt) pass assessments in higher education programming courses? *arXiv preprint*, 2023.

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[33] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, page 55–56, New York, NY, USA, 2017. Association for Computing Machinery.

[34] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2023.

[35] Juho Leinonen, Arto Hellas, Sami Sarsa, Brent Reeves, Paul Denny, James Prather, and Brett A. Becker. Using language models to enhance programming error messages. In *Proceedings of the 2023 ACM SIGCSE Technical Symposium on Computer Science Education*, 2023.

[36] Mark Liffiton, Brad Sheese, Jaromir Savelka, and Paul Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes, 2023.

[37] Mike Wu, Noah D. Goodman, Chris Piech, and Chelsea Finn. Prototransformer: A meta-learning approach to providing student feedback. *CoRR*, abs/2107.14035, 2021.

[38] Charles Koutcheme. Training Language Models for Programming Feedback Using Automated Repair Tools. In Ning Wang, Genaro Rebolledo-Mendez, Noboru Matsuda, Olga C. Santos, and Vania Dimitrova, editors, *Artificial Intelligence in Education*, pages 830–835, Cham, 2023. Springer Nature Switzerland.

[39] Charles Koutcheme, Sami Sarsa, Juho Leinonen, Lassi Haaranen, and Arto Hellas. Evaluating distance measures for program repair. In *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*, ICER '23, page 495–507, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Tung Phung, Victor-Alexandru Pădurean, 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, 2023.

[41] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. Cognitive load theory in computing education research: A review. *ACM Trans. Comput. Educ.*, 22(4), sep 2022.

[42] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.

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

[44] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. Automated Clustering and Program Repair for Introductory Programming Assignments, June 2018. arXiv:1603.03165 [cs].

[45] Yana Malysheva and Caitlin Kelleher. An algorithm for generating explainable corrections to student code. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, Koli Calling '22, New York, NY, USA, 2022. Association for Computing Machinery.

[46] Ke Wang, RIshabh Singh, and Zhendong Su. Data-Driven Feedback Generation for Introductory Programming Exercises. *arXiv:1711.07148 [cs]*, November 2017. arXiv: 1711.07148.

[47] Adrian de Freitas, Joel Coffman, Michelle de Freitas, Justin Wilson, and Troy Weingart. Falconcode: A multiyear dataset of python code samples from an introductory computer science course. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2023, page 938–944, New York, NY, USA, 2023. Association for Computing Machinery.

[48] Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. Finetuned language models are zero-shot learners, 2022.

[49] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[50] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open language model for code with multi-turn program synthesis, 2023.

[51] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

[52] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[53] Rahul Gupta, Aditya Kanade, and Shirish Shevade. *Neural Attribution for Semantic Bug-Localization in Student Programs*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[54] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[55] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[56] Kushal Tirumala, Aram H. Markosyan, Luke Zettlemoyer, and Armen Aghajanyan. Memorization without overfitting: Analyzing the training dynamics of large language models, 2022.

[57] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[58] Charles Koutcheme. Towards open natural language feedback generation for novice programmers using large language models. In *Proceedings of the 22nd Koli Calling International Conference on Computing Education Research*, Koli Calling '22, New York, NY, USA, 2022. Association for Computing Machinery.

[59] Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity, 2023.

[60] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

[61] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020.

[62] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp, 2019.

[63] Lingjiao Chen, Matei Zaharia, and James Zou. How is chatgpt's behavior changing over time?, 2023.

[64] Arto Vihavainen, Jonne Airaksinen, and Christopher Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the tenth annual conference on International computing education research*, pages 19–26, 2014.

[65] Andrew Luxton-Reilly, Simon, Ibrahim Albluwi, Brett A Becker, Michail Giannakos, Amruth N Kumar, Linda Ott, James Paterson, Michael James Scott, Judy Sheard, et al. Introductory programming: a systematic literature review. In *Proceedings companion of the 23rd annual ACM conference on innovation and technology in computer science education*, pages 55–106, 2018.

[66] Sami Sarsa, Juho Leinonen, Arto Hellas, et al. Empirical evaluation of deep learning models for knowledge tracing: Of hyperparameters and metrics on performance and replicability. *Journal of Educational Data Mining*, 14(2), 2022.

[67] Shahriar Golchin and Mihai Surdeanu. Time travel in llms: Tracing data contamination in large language models, 2023.

[68] Greg L Nelson, Benjamin Xie, and Amy J Ko. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM conference on international computing education research*, pages 2–11, 2017.

[69] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2018.

[70] Juha Sorva and Otto Seppälä. Research-based design of the first weeks of cs1. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, pages 71–80, 2014.

[71] John Edwards, Joseph Ditton, Dragan Trninic, Hillary Swanson, Shelsey Sullivan, and Chad Mano. Syntax exercises in cs1. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, pages 216–226, 2020.

A Appendix

A.1 Dataset details

In this section, we detail the processing steps applied to each of our datasets.

A.1.1 FalconCode

We requested and acquired the FalconCode dataset from the authors' website website, which consists of multiple tables as detailed in the paper by [47]. The downloaded dataset encompasses 1,433,323 submissions written by 1,563 students across 478 distinct assignments spanning three semesters.

There are three levels of difficulty of assignments in the dataset: "skill", "lab" and "project". For this study, we excluded "project" assignments, which have high complexity, often require extensive code writing across multiple files, and lack automated tests. For all our analysis, we refer to "skill" and "lab" assignments as "easy" and "hard" (respectively). Several other assignments require access to external files (e.g, .csv files or .txt files) which the authors did not release in the current version of the dataset. However, we are currently in the process of retrieving this information. We excluded all assignments that require these external files. We then removed students' intermediate runs (not submitted for grading). All other submissions present were automatically graded and assigned a score comprised between 0 and the exercise maximum score (typically 100). We removed all submissions which have a score of 0, as manual inspection reveals that these solutions are often the results of students' obvious mistakes (e.g. forgetting to print) or students' "trial and error". After removing intermediate runs and zero-passing codes, we have in total 270,478 submitted programs to 334 problems written by 1557 students.

Following [43], we selected only the final submissions for each student for each assignment. This resulted in our test set comprising submissions from students who did not complete the assignments successfully. While we acknowledge that this selection may not fully capture the range of difficulties students encounter during their attempts, it aligns with the idea that a student's last attempt often reflects their improved understanding of the problem. Additionally, students are more likely to seek help after several unsuccessful attempts. Thus, our setup can be viewed as providing feedback to students as a last resort for elements they may not have grasped. In our dataset, this selection process yielded 118,764 distinct solutions. We recognize the potential for future work to explore more refined methods for automatically selecting meaningful solutions within large datasets.

For dataset splitting, we adopted a methodology similar to [38]. We divided the dataset by semester, designating the first two semesters (fall and spring 2021) for training and development purposes, respectively. The remaining semester constituted the testing set (spring 2022). Within each split, we independently removed submissions that shared identical abstract syntax tree (AST) structures after variable normalization. Table 2 provides an overview of the dataset statistics after this processing.

Table 2: **FalconCode dataset statistics.**

subset	training		development		testing	
difficulty	easy	hard	easy	hard	easy	hard
# correct solutons	4413	16463	4521	13552	3678	12164
# incorrect solutions	398	901	622	717	538	767
# problems	89	104	87	68	83	89

Note that we intentionally also limited the size of our training and dev subsets to match the distribution of the test set. We encourage further experimentation with larger subsets of training datasets.

A.1.2 Singapore

We obtained the Singapore dataset from the original paper's GitHub repository [43]. The downloaded dataset initially contains 4225 submissions from 361 students. As with the FalconCode data, we removed similar submissions based on AST comparison on the Singapore dataset, which left us with a total of 2436 submissions. Table 3 displays the dataset statistics. It's important to note that the original dataset lacks information regarding which student submitted which assignment, and that we do not make use of the correct solutions.

Table 3: **Singapore dataset statistics.**

Assignment	1	2	3	4	5	Total
# correct solutions	480	157	152	186	223	1196
# incorrect solutions	296	331	246	261	104	1238

A.2 Experiment details

In this section, we detail how we created artificial code repair examples for supervised training (fine-tuning) of the used models, and how we trained and evaluated our models.

A.2.1 Neural Automated Repair

Prior to training, we utilized the Refactory Automated Repair Tool (ART) [43] to generate repairs for each incorrect program in the training and development datasets from FalconCode [38]. Detailed information about Refactory's functionality can be found in the original paper and its associated GitHub repository[3].

We configured Refactory with online refactoring, sampling 100% of correct student programs from both the training and development subsets, and enabled structure mutation and block repair phases (i.e., using the best settings available). Unlike in [38], if ART couldn't successfully repair a faulty program, we followed the approach described in [53]. In such cases, we employed the Rouge-L score [39, 42] to map the incorrect program to the nearest correct program.

A.2.2 Supervised training

We finetuned our models for next token prediction using HuggingFaces' transformers library [61] using separate training and development sets. To ensure efficient training, we truncated each sequence to a maximum of 512 tokens. We decayed the learning rate with a cosine schedule, using an initial learning rate of 5e-5. We then performed a hyperparameter search, varying the number of epochs (1, 2, or 3) and batch sizes (4, 8, 16, or 32); the model selected for final evaluation with the testing set is the one showing the lowest validation loss on (a subset of) the development set. Our template prompt displayed in Figure 2 shows the structure of the prompts that we used to generate the repairs with the models. Notably, the last part of the prompt that includes the repair example found with ART is included only for the training and development set and excluded for testing sets. It's important to note that the prompt used did not include the graders' feedback. In our future work, we will evaluate separately the effect of adding such feedback on program repair performance.

A.2.3 Evaluation

We generated 10 repairs for each incorrect program in our test set using our prompt template shown in Figure 2. The repaired programs were generated using top_p nucleus sampling with a fixed temperature of 0.6, similarly as in [23] (all other parameters are default). We refer the reader to section 3.1 for the rest of the evaluation procedure.

A.2.4 Computational resources.

The training and evaluation pipelines were run individually and separately for each model on a single Nvidia Tesla A100 GPU using our institution research cluster.

[3] https://github.com/githubhuyang/refactory

```
Repair the incorrect program. (1)

{PROBLEM_DESCRIPTION} (2)

**Incorrect program** (3)

{BUGGY PROGRAM} (4)

**Repaired code**: (5)

{CORRECTED PROGRAM} (6)
```

Figure 2: The prompt template used for model training and evaluation: (1) a preamble orienting the model to the task of the program repair (although pretrained models are not instruction tuned, a preamble remains a useful signal), (2) the problem description extracted from the dataset, (3) an indication of the start of the buggy program, (4) the buggy program to fix, (5) an indication of the start of the repaired program, (6) the artificial repair found for the buggy program (see section A.2.1) – included only in the training phase.