Simulating Programming Traces and State Spaces for Hint Generation

Jesper Dannath
Bielefeld University
jdannath@techfak.uni-bielefeld.de

Benjamin Paaßen
Bielefeld University
bpaassen@techfak.uni-bielefeld.de

ABSTRACT

Data-driven approaches for next-step hint generation in programming suffer from a cold-start problem, meaning that high-quality hints depend on the presence of sufficiently similar student data – which is missing whenever systems are deployed for new programming tasks. Pre-trained large language models (LLMs) circumvent this cold-start problem. However, because LLMs are not fine-tuned for specific tasks, their hints may contain irrelevant information, or reveal the solution outright. Further, LLMs require substantial resources and long response times to generate hints. We propose to combine the strengths of both LLMs and data-driven approaches: We harness the flexibility of LLMs to generate diverse artificial student solutions. Then we compare methods for simulating student programming traces, thus addressing the cold-start problem. Importantly, we synthesize traces subtractively, starting from the solution and recursively removing elements, thus achieving a fine-grained simulation of programming progress. From simulated traces, state spaces can be constructed and used to generate hints. Our results show that the simulated state spaces are similar to the ones obtained from real student data. Additionally, student step prediction using a simulated state space leads to comparable results to step prediction via prompting LLMs – while being substantially more efficient at inference time.

Keywords

programming traces, state space simulation, programming hints, intelligent tutoring systems

1. INTRODUCTION

A lack of resources for personalized tutoring remains a major challenge in introductory programming education [12]. Personalized hints provided by intelligent tutoring systems during programming tasks can complement teaching and enhance the quality of programming education [23]. AI tools are already used by many programming students for assistance on homework tasks [28]. However, these tools can lead to over-reliance and an offloading of cognitive effort, potentially hindering learning [26]. According to Vygotsgyan pedagogy and scaffolding theory, teachers should give only enough assistance to students such that they can proceed on the given task by themselves [7]. Next step hints provide a framework for applying scaffolding theory in AI-assisted learning environments [11]. Large Language Models (LLMs) can be used to generate next step hints in programming effectively [21] but consume significant resources and even when prompted not to, they tend to give the students access to multiple steps or even complete solutions [1]. On the other hand, data-driven methods for generating next step hints usually rely on a pool of pre-existing student and hint data [11]. However, these approaches suffer from the cold-start problem, meaning that they rely on the availability of interaction data or teacher-generated data.

In this work, we aim to bridge the gap between LLM-driven and data-driven hint-generation by simulating an initial set of trace data for data-driven algorithms. We utilize LLMs to generate diverse verified solutions across a variety of approaches and solution-complexity levels. We then employ our subtractive trace-simulation approach, meaning that we start from a (student-generated or LLM-synthesized) correct solution to a task and recursively remove elements until the empty program is reached. By analysing the state spaces that arise from such simulated programming traces, we aim to answer the main research question whether and under which circumstances using simulated interaction data is a feasible approach for programming hint-generation? We also address the question whether the simulated student traces resemble real student data and might be useful as synthetic training data or for testing purposes. We approach these questions with the following core contributions:

1. Novel methods for subtractively simulating programming traces from correct solutions including a finetuned LLM for reversing programming steps in Python.
2. An experimental comparison of simulated versus real state spaces.
3. An investigation on the feasibility of applying the simulated traces for downstream hint generation methods.

2. RELATED WORK

Data-driven hint generation methods often rely on state spaces to represent student behavior [18]. A state space

Jesper Dannath, and Benjamin Paaßen. Simulating Programming Traces and State Spaces for Hint Generation. In Anthony Botelho, Maria Mercedes T. Rodrigo, Adish Singla, Hiroaki Ogata, Hyojeong So, and Young Hoan Cho (eds.) Proceedings of the 19th International Conference on Educational Data Mining, Seoul, Republic of Korea, June, 2026, pp. 774–779. International Educational Data Mining Society (2026).

© 2026 Copyright is held by the author(s). This work is distributed under the Creative Commons Attribution NonCommercial NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.
https://doi.org/10.5281/zenodo.21039736

consists of a directed graph where the nodes are student states and the edges represent student steps between states. Each state is hereby closely related to the current student program (snapshot). Often, some form of canonicalization or clustering is applied to group slightly different program snapshots and thus achieve a more robust and concise state space [19, 18, 17]. Next step hints can be generated from state spaces using a two-step approach: First, the student's current snapshot is matched to an existing state in the state space [17]. Second, an algorithm selects one of the outgoing states as a prediction of the next state – which is then provided as a hint [11]. One example of such an algorithm would be the hint factory, which employs a Markov Decision Process to select a next step that leads to a correct solution on a short path [18].

For a long time, simulating student behavior has been used to model and understand student learning [24, 22]. The focus used to be on modeling student's cognitive properties and predict the outcome of student task attempts [27]. In programming, it is very difficult to obtain fine-grained simulation data of student interactions, since the solution spaces for programming tasks are vast (combinatorially exploding with the size of the program). Today, LLMs offer a new paradigm for generating synthetic data in domains with vast solution spaces [13]. Previous work has utilized in-context learning and fine-tuning LLMs to generate realistic student attempts based on a reference trace by the same student [14]. Even more recent work is using LLM-Agents that employ reasoning-LLMs and tool-calling in order to derive each student step [6, 27]. In particular, CoderAgent includes memory-retrieval and update steps as well as planning and reflection steps yielding higher Accuracy compared to previous methods [27]. The resulting traces can be used in downstream tasks such as learner modeling and training hint generation models taking student knowledge and learning into account. Our proposed subtractive approach sacrifices some versatility, but circumvents the complexity of predicting a next state by simulating backwards, such that a model merely has to decide which part of a program was likely added last, requiring substantially fewer resources.

3. METHOD

We are approaching programming hint generation through the use of simulated state spaces which are aggregated from simulated programming traces. We can view a programming trace as a sequence of program snapshots s_t for $t \in \{1, ..., T\}$. For simulating traces, we need to predict the mapping from s_t to s_{t+1}, which is a difficult problem, as the number of possible line steps is vast. Additionally, the mapping needs to be applied recursively to obtain a complete trace with no guarantee that a correct solution will be reached (and risk of error accumulation). On the other hand, if s_{t+1} is given, estimating s_t is significantly easier in many cases, as usually we just have to decide which line to delete. Therefore, our strategy is to construct an approximate mapping f s.t. $f(s_{t+1}) \approx s_t$. This mapping can be applied recursively to yield a reverse trace $s_T, s_{T-1}, \ldots, s_0$ with $s_0 = \varepsilon$, the empty program. We call this a *subtractive* approach.

A state space for a task is traditionally build from a range of example attempts, aggregated on a specific step-granularity (programming traces). As outlined above, scaffolding theory provides a pedagogically inspired ideal for step-granularity. A previous user-study indicates that the line-level might be a good approximation of a reasonable scaffolding step-granularity for introductory python tasks [3]. Therefore, we build state spaces from simulated programming traces such that transitioning between nodes is equivalent to applying one line-level change to the current user program (see Figure 3). Below we describe the simulation-method in further detail.

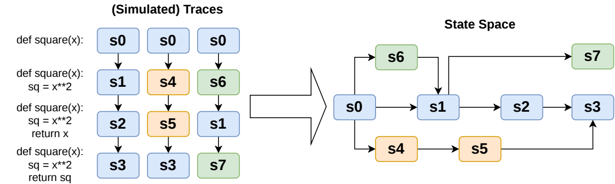

Figure 1: Method: simulated line-based programming traces are aggregated to state spaces

3.1 Generating solutions

For generating diverse correct solutions to a task we are applying a two-step prompting approach. We ask a reasoning-LLM (Qwen3.5-397B) to generate a list of 15 approaches to the task described in one or two sentences. Then, in a second step, a LLM (Qwen3-235B) is prompted without reasoning to implement different versions of each of the approaches. Specifically we first ask the LLM to provide a baseline implementation for a student with medium python skill. Then we repeatedly ask the LLM to make a simpler or a more complex version of it, until the quota (61 solutions total per task) is reached. If a generated solutions wasn't unique after up to four retries, we reduced the quota by one and proceeded to the next attempt. The complete prompt and all other prompts as well as the code for all analysis and supplementary analysis are accessible in our GitLab repository[1].

3.2 Deleting from a random block (RBLL)

We evaluate a programmatic approach for step subtraction which is to simply remove the last line from a random control block. To achieve this, we use the Python abstract syntax tree (AST)-module to list all control structures and obtain the lines which their respective block spans (note that blocks can be nested). Then, we sample one random control block (uniform probability) and delete its last line as well as other lines that might need to be deleted to retain a correct syntax. We call this method RBLL (Random Block Last Line; speak rebel). Note that correct syntax is needed to analyze the block structure of Python code. Hence, some Python-specific mechanisms are imposed to ensure that RBLL never deletes lines in ways that would break the syntax (accounting for multi-line statements, except blocks, inserting pass statements for empty blocks, resolving try blocks if their except is deleted). While this code-processing procedure is specific to Python, the reasoning of first standardizing the syntax, then reversing a step and then fixing a few edge cases of breaking syntax can be applied to other imperative

[1]`https://gitlab.ub.uni-bielefeld.de/publications-ag-kml/step-simulation`

programming languages as well. We do not claim that our approach addresses all possible edge-cases in Python. However, we could not observe any simulation error for RBLL throughout our experiments.

3.3 Prompting LLMs for reversed steps

We compare RBLL with a step-subtraction method based on pretrained LLMs. We utilize an iteratively developed prompt format which is structured into system prompt, context and query. The prompt context contains three elements, starting with a description of how the outputted reversed line steps should be structured. Then, we give the model the correct solution it is deducting steps from. Finally, we embed the first three snapshots and the three most recent snapshots in the model context. In the query we then continue to prompt the model to delete a single line from the current snapshot. Alternatively, we sometimes prompt the model to introduce an error or a misconception into one existing line of the current snapshot. We sample the query such that the model is asked to introduce errors or misconceptions 30% of the time. We did find that this value yields a significant increase in trace diversity while maintaining plausibility. Generally, this method relies on the LLM to branch off into different directions on repeated simulation runs, while RBLL explicitly employs random branching. The complete prompt is available in GitLab. Since LLMs will sometimes fail to produce a reverse step, we are validating the success of the step reversion. When the LLM was unsuccessful (i.e., leaving code unchanged, manipulating more than a single line, or breaking the syntax), we fall back to RBLL.

3.4 Fine-tuning LLMs for reversed steps

Finding that in many cases the LLM was incapable of reversing a line step correctly (see Section 4.2), we concluded fine-tuning might be a viable strategy to improve performance. We used the described prompting format to create a dataset of real reverse traces. In addition to our dataset (Section 4), we used a second keystroke-level dataset from Utah State University (2019) for additional training examples [5]. Then we employed supervised fine-tuning utilising unsloth [2] with the real reverse steps as a target. 50% of the tasks of each of both datasets (20 tasks in total) were selected randomly for training, resulting in 16465 training examples. We trained a 16-bit LoRA [8] with rank 128 for one epoch (lr 1e-4) on a single A40 GPU. The model was inferenced in 8-bit. We used Qwen3-8B [25] as a base model without reasoning traces or thinking mode. We reference the fine-tuned model as Qwen3-8B-SR (SR for Step Reversion).

4. EXPERIMENTS

We evaluate the state space simulations on a real-world dataset. The dataset was collected at a Mid-European university and consists of programming traces from 29 Python tasks recorded via an intelligent tutoring system [4]. In total, 209 students donated 1400 programming traces across Python Introduction sessions between 2023 to 2025. From these traces, which are recorded on a keystroke level, 7370 Line steps could be inferred. After randomly selecting 50% of tasks for fine-tuning, 14 tasks remained for computing the below experiments. For obtaining line steps, We also remove all complete detours from student traces. "Complete" refers to the kind of detours that lead to the exact same state later on. The tasks are about basic python concepts and NumPy as well as Data Mining methods. For evaluation, we used pretrained LLMs of different sizes from the Qwen3 model family, namely Qwen3-8B, Qwen3-coder-30B-A3B (short Qwen3-30b) and Qwen3-235B [25]. Additionally, we utilized RBLL as a baseline and Qwen3-8B-SR as our own fine-tuned model based on Qwen3-8B (Section 3.4). The 8B models were both used in 8-bit quantization on a single NVIDIA RTX 5060 Ti with 16GB of VRAM. Qwen3-30B and Qwen3-235B were used in 16-bit quantization using the scaleway-API. We use three traces per solution for the simulation of state spaces.

Table 1: Fallback rates for step reversion by LLM

Qwen3-8B	Qwen3-30B	Qwen3-235B	Qwen3-8B-sr
76.9%	74.1%	55.0%	35.8%

4.1 Metrics

We compare state spaces of the real student traces (from tasks in the test set) with simulated state spaces. All metrics are based on a matching of states between the real and the simulated state spaces. For matching, we canonicalize all programs using their AST, normalizing variable, function, and class names, and standardizing strings (programmatically formatted parts of strings are kept). To compare states, our metrics are *precision*, meaning the fraction of states in the simulated state space that also occur in the real state space, and *recall*, meaning the fraction of states in the real state space that also occur in the simulated state space. We also report *recall-5*, where we relax the state matching and allow up to five tokens edit-distance (e.g. single function call with one argument; NLTK wordPunctTokenizer). Recall-5 aims to yield a more realistic estimate of a classroom-setting, where one would usually allow relaxed matching to be able to deliver hints to more students. To compare steps in the state space (i.e. edges from one state to another), we search for matches of real states in the simulated state space and compute the number of the available outgoing edges of these matches (0 if the real state does not exist in the simulated state space). We call this metric *average outgoing score*. The outgoing score aims to estimate how many candidate hints we can expect for a randomly selected state (without accounting for state likelihood). All metrics are averaged across tasks.

4.2 Fallback-rates

We found that, in many cases, the pretrained LLMs were not able to follow the prompt of deleting a single line. Instead, the LLMs often returned the exact same snapshot as before, edited multiple lines, or added lines to the current snapshot. On occasions where the LLM fails to reverse a line-step, we simply fallback to RBLL. Throughout all traces simulated for the experiments in this paper, we observe that failure rates decrease with model size, with 76.9% for the Qwen3-8B and 55% for Qwen3-235B (Table 1). Still, fine-tuning is even more effective than model size with merely 35.8% failure rate for the fine-tuned Qwen3-8B-SR.

4.3 Simulation based on existing solutions

In order to evaluate the step-reversion performance in isolation, we examine the performance of the simulation meth-

Table 2: Mean and SD of state space metrics

Method	Precision	Recall	Recall-5	Outgoing
RBLL	**.43 (.19)**	.29 (.09)	.52 (.16)	.46 (.14)
Qwen3-8B	.38 (.15)	.32 (.08)	.55 (.15)	.56 (.16)
Qwen3-30B	.32 (.13)	.31 (.1)	.55 (.17)	.61 (.19)
Qwen3-235B	.24 (.11)	.32 (.11)	.55 (.16)	**.75 (.26)**
Qwen3-8B-SR	.31 (.14)	**.33 (.09)**	.57 (.16)	.63 (.15)

Table 3: Mean and SD of state space metrics for AI-generated solutions

Method	Precision	Recall	Recall-5	Outgoing
RBLL	**.06 (.05)**	.1 (.06)	.31 (.17)	.36 (.21)
Qwen3-8B	.05 (.03)	.11 (.07)	.3 (.14)	.47 (.31)
Qwen3-30B	.05 (.03)	.11 (.07)	.31 (.16)	.5 (.31)
Qwen3-235B	.03 (.02)	**.12 (.07)**	.32 (.17)	**.67 (.43)**
Qwen3-8B-SR	.05 (.05)	.11 (.07)	.29 (.13)	.53 (.36)

ods for recovering state spaces from corresponding real student solutions (Table 2). The results show that, while the control-block based simulation method has the highest precision (.43), it is slightly outperformed by the LLM-based simulation methods on recall and recall-5. In terms of outgoing score, the LLM-based methods outperform the programmatic method by a larger margin, with the Qwen3-235B performing best (.75). For the use-case of synthetic trace-dataset generation for testing or model training, one would have to decide between using RBLL for maximum precision or using the LLM-based approach for more variability and slightly improved recall. Also, while being sufficient on our introductory dataset, the performance of all methods might decrease for longer tasks (See Section 5 and Appendix A).

4.4 Simulation with generated solutions

In many scenarios such as introducing a completely new task, it is reasonable to assume that little to no correct solutions to the task are available. In this section, we investigate how well we can recover real state spaces using only LLM-generated solutions to tasks. LLM-generated solutions are verified using unit tests (therefore, two tasks without unit tests had to be excluded). Table 3 shows the metrics based on up to 61 generated solutions per task (A task might have fewer as described in Section 3.1). In fact, the number of unique solutions that could be generated per task varies between 10 and 61 with a mean of 50.0. The relative performance between the different methods remains similar compared to using the real student solutions: RBLL achieves highest precision, LLMs better recall and more variability. For all methods, overall performance is lower compared to having real student solutions available. Interestingly, the outgoing metric degrades only slightly, whereas precision and recall decrease by a factor of 3. Recall-5 remains more stable between the two scenarios decreasing only by approximately 1/3. We expect that an increased number of generated solutions will generally lead to a higher number of recovered states. In further experiments we can observe that the recall indeed increases for all methods up to 61 generated solutions but the increase diminishes significantly (Figure 2).

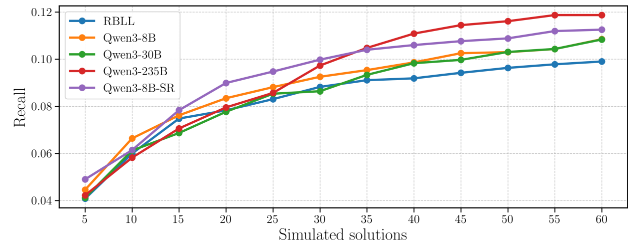

Figure 2: Avg. Outgoing states by number of generated solutions

Table 4: Comparing direct-LLM and state space based step prediction

Method	Tree-edit dist.	chrF	Acc	Acc-5
direct-LLM (8B)	**6.88**	0.76	**0.2**	**0.43**
direct-LLM (235B)	7.27	0.76	0.2	0.41
Sim RBLL	7.03	0.76	0.10	0.37
Sim 235B	7.37	0.76	0.06	0.32
Sim 8B-SR	7.55	**0.77**	0.07	0.33

4.5 Predicting steps

Deriving next step hints using simulated state spaces is not a trivial task. First, a student's new state needs to be matched against an existing state in the simulated state space. We do so by using the closest state according to tree edit distance (TED) on AST [15]. To select a next-step hint we consider the successor of the matched state in the simulated state space closest to the student's state (in terms of TED). If the successor differs in exactly one line from the student's state, this is defined as the next-step hint. Otherwise, we compute the line edit between the matched state and the successor and apply them to the student state, inspired by [10]. If this does not yield correct syntax, we attempt syntax repair by fixing indentation-levels; and if this does not succeed, we predict the successor directly (even if it differs by more than one line). As a baseline, we compare with an LLM prompted to predict the next line step. The prompt contains information on the expected format and step-granularity (line-steps) as well as the task description and current student step and is also available in the linked repository.

Since step prediction is a different task compared to the above evaluations, we employ different metrics. In particular, we report the TED to the actual next step of a student (lower is better), the character-level F-score (chrF; higher is better), which has been shown to align well with human assessment on code generation tasks [16], and the accuracy, meaning the rate of exact matches after canonicalization (higher is better). We also report Accuracy-5, where we mark a prediction as correct based on a relaxed matching up to an token edit-distance of 5 (similar to recall-5). Table 4 shows the results for predicting the next student step on the test-dataset. Among the simulation-based step predictors, RBLL performs best in terms of accuracy and TED. The LLM next step hints perform best in terms of accuracy but are slightly outperformed in terms of chrF. TED is comparable between RBLL simulation and LLMs.

5. CONCLUSIONS

In this work, we investigated the potential of simulating programming traces and state spaces using a subtractive approach based on LLMs and/or programmatic line removal. Such simulation has two major purposes: creating synthetic datasets and providing data for generating next step hints. Our experiments confirm that we can recover real intermediate student states if realistic solutions are available. The recall and precision vary by the employed simulation method: random block last line (RBLL) provides generally higher precision compared to LLMs; and larger LLMs generally achieve lower precision but higher average number of matched outgoing states. This is likely explained by larger LLMs generating more diverse programming traces. Throughout all experiments, the fine-tuned LLM Qwen3-8B-SR performs mostly on par with the much larger 235B model while also retaining a bit more precision. Additionally, the fine-tuned model has a much lower fallback rate, indicating that the task of removing lines from existing code is beyond the scope of the model's pre-training procedure. Using the simulated state spaces for step prediction shows that the LLM-based approaches do not have a benefit compared to RBLL. Given that RBLL is also much more efficient, it is currently the most advisable method for next step hint generation with simulated state spaces.

We also believe that the potential of the selection in state spaces is not fully realized in this work yet, as we didn't investigate more advanced methods of predicting steps within the simulated state spaces. Future work could investigate this and might also find ways to take advantage of the more diverse traces that LLM-based step simulation produces. Also, we did not investigate the effect of combining the simulated state spaces with student or teacher-provided trace data, as might be the case in real-world deployment. Some caution is advised for applying the presented methods for more complex tasks since our dataset only consists of easy introductory python tasks. Our supplementary experiments show decreased performance for state space reconstruction with real solutions on the Utah State keystroke-level programming dataset from 2019 [5], which consists of print statement heavy longer tasks like turtle tasks or CLI based games (see Appendix A). These tasks are considerably longer compared to the ones in our own dataset, but unfortunately lack unit tests and therefore could not be tested for state space reconstruction with LLM-generated solutions and step prediction. Generally, there is a lack of openly available line level or sub-line level python-trace datasets that could be used for evaluating the performance of our method for a larger variety of tasks [9].

Future work should also investigate whether the generated hints from simulated state spaces are effective in a pedagogical sense. This could also be supported by additional effort for personalizing the retrieved simulated states to the students current snapshot, like proposed in [20]. Finally, the large difference in performance between Table 2 and Table 3 suggest that future approaches might be able to improve the quality of simulated state spaces by focusing on the generation of more realistic student solutions. Still, RBLL provides a cheap and effective baseline for future work on simulated state spaces.

6. ACKNOWLEDGMENTS

This work is funded by the Ministry of Culture and Science of the State of North Rhine-Westphalia within the scope of the SAIL project. Funding ref: NW21-059A.

APPENDIX

A. UTAH STATE 2019 DATASET RESULTS

Table 5: Utah 2019 - Mean and SD of state space metrics

Method	Precision	Recall	outgong
RBLL	**.1 (.05)**	.1 (.03)	.15 (.04)
Qwen3-8B	.09 (.04)	.1 (.03)	.16 (.04)
Qwen3-30B	.09 (.04)	.11 (.03)	.16 (.04)
Qwen3-235B	.07 (.03)	.11 (.03)	**.21 (.05)**
Qwen3-8B-SR	.07 (.03)	.11 (.03)	.18 (.04)

2. REFERENCES

[1] A. Birillo, E. Artser, A. Potriasaeva, I. Vlasov, K. Dzialets, Y. Golubev, I. Gerasimov, H. Keuning, and T. Bryksin. One step at a time: Combining llms and static analysis to generate next-step hints for programming tasks. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*, New York, NY, USA, 2024.

[2] M. H. Daniel Han and U. team. Unsloth, 2023.

[3] J. Dannath, A. Deriyeva, and B. Paassen. What is a Step? A User Study on How to Sub-divide the Solution Process of Introductory Python Tasks. In *Proceedings of the 18th International Conference on Educational Data Mining*, Palermo, Italy, July 2025.

[4] A. Deriyeva, J. Dannath, and B. Paaßen. SCRIPT: Implementing an intelligent tutoring system for programming in a german university context. In C. Stracke, S. Sankaranarayanan, P. Chen, and E. Blanchard, editors, *Proceedings of the 26th International Conference on Artificial Intelligence in Education (AIED 2025) Practitioner's Track*, Palermo, Italy, 2025.

[5] J. Edwards, K. Hart, and R. Shrestha. Review of CSEDM Data and Introduction of Two Public CS1 Keystroke Datasets. *Journal of Educational Data Mining*, 15(1):1–33, 2023.

[6] W. Gao, Q. Liu, L. Yue, F. Yao, R. Lv, Z. Zhang, H. Wang, and Z. Huang. Agent4Edu: Generating Learner Response Data by Generative Agents for Intelligent Education Systems. In *Proceedings of the 39th AAAI Conference on Artificial Intelligence*, Philadelphia, PA, USA, Feb. 2025. AAAI Press.

[7] J. Hammond and P. Gibbons. Putting scaffolding to work: The contribution of scaffolding in articulating ESL education. *Prospect*, 20(1):6–30, 2005.

[8] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-Rank Adaptation of Large Language Models. In *Proceedings of the 10th International Conference on Learning Representations*, pages 1–15, Virtual, Apr. 2022.

[9] N. Kiesler, J. Impagliazzo, K. Biernacka, A. Kapoor, Z. Kazmi, S. G. Ramagoni, A. Sane, K. Tran, S. Taneja, and Z. Wu. Where's the Data? Finding and Reusing Datasets in Computing Education. In *Working Group Reports on 2023 ACM Conference on*

Global Computing Education, pages 31–60, Hyderabad India, Sept. 2024. ACM.

[10] T. Lazar and I. Bratko. Data-driven program synthesis for hint generation in programming tutors. In S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, editors, *Intelligent Tutoring Systems*, pages 306–311, Cham, 2014. Springer International Publishing.

[11] J. McBroom, I. Koprinska, and K. Yacef. A Survey of Automated Programming Hint Generation – The HINTS Framework. *ACM Computing Surveys*, 54(8):1–27, Nov. 2022.

[12] R. P. Medeiros, G. L. Ramalho, and T. P. Falcão. A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, 62(2):77–90, 2019.

[13] M. Nadas, L. Diosan, and A. Tomescu. Synthetic Data Generation Using Large Language Models: Advances in Text and Code. *IEEE Access*, 13:134615–134633, 2025.

[14] M. H. Nguyen, S. Tschiatschek, and A. Singla. Large Language Models for In-Context Student Modeling: Synthesizing Student's Behavior in Visual Programming. In P. Benjamin and D. E. Carrie, editors, *Proceedings of the 17th International Conference on Educational Data Mining*, Atlanta, GA, USA, July 2024. International Educational Data Mining Society.

[15] B. Paaßen. Revisiting the tree edit distance and its backtracing: A tutorial. *arXiv*, 1805.06869, 2022.

[16] D. G. Paul, H. Zhu, and I. Bayley. Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review. In *Proceedings of the 1st IEEE International Workshop on Testing and Evaluation of Large Language Models*, San Jose, CA, USA, July 2024. IEEE.

[17] T. Price, R. Zhi, and T. Barnes. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *Proceedings of the 10th International Conference on Educational Data Mining*, pages 318–323, Wuhan, China, 2017. International Educational Data Mining Society.

[18] T. W. Price, Y. Dong, and T. Barnes. Generating Data-driven Hints for Open-ended Programming. In *Proceedings of the 9th International Conference on Educational Data Mining*, Raleigh, North Carolina, 2016.

[19] K. Rivers and K. R. Koedinger. A Canonicalizing Model for Building Programming Tutors. In S. A. Cerri, W. J. Clancey, G. Papadourakis, and K. Panourgia, editors, *Intelligent Tutoring Systems*, volume 7315 of *Lecture Notes in Computer Science*, pages 591–593. Springer, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[20] K. Rivers and K. R. Koedinger. Automating Hint Generation with Solution Space Path Construction. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, A. Kobsa, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, D. Terzopoulos, D. Tygar, G. Weikum, S. Trausan-Matu, K. E. Boyer, M. Crosby, and K. Panourgia, editors, *Intelligent Tutoring Systems*, volume 8474, pages 329–339. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science.

[21] L. Roest, H. Keuning, and J. Jeuring. Next-step hint generation for introductory programming using large language models. In *Proceedings of the 26th Australasian Computing Education Conference*, Sydney, Australia, 2024.

[22] X. Tian, G. Pan, and H. Wang. Large Language Models in Student Simulation: A Survey, Nov. 2025. https://doi.org/10.21203/rs.3.rs-8148343/v1.

[23] K. VanLehn. The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems. *Educational Psychologist*, 46(4):197–221, Oct. 2011.

[24] K. VanLehn, S. Ohlsson, and R. Nason. Applications of Simulated Students: An Exploration. *Journal of Artificial Intelligence in Education*, 5(2):135–135, 1994.

[25] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, C. Zheng, D. Liu, F. Zhou, F. Huang, F. Hu, H. Ge, H. Wei, H. Lin, J. Tang, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Zhou, J. Lin, K. Dang, K. Bao, K. Yang, L. Yu, L. Deng, M. Li, M. Xue, M. Li, P. Zhang, P. Wang, Q. Zhu, R. Men, R. Gao, S. Liu, S. Luo, T. Li, T. Tang, W. Yin, X. Ren, X. Wang, X. Zhang, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Zhang, Y. Wan, Y. Liu, Z. Wang, Z. Cui, Z. Zhang, Z. Zhou, and Z. Qiu. Qwen3 Technical Report, May 2025.

[26] C. Zhai, S. Wibowo, and L. D. Li. The effects of over-reliance on AI dialogue systems on students' cognitive abilities: a systematic review. *Smart Learning Environments*, 11(1):28, June 2024.

[27] Y. Zhan, Q. Liu, W. Gao, Z. Zhang, T. Wang, S. Shen, J. Lu, and Z. Huang. CoderAgent: Simulating Student Behavior for Personalized Programming Learning with Large Language Models. In *Proceedings of the 34th International Joint Conference on Artificial Intelligence*, Montreal, Canada, Aug. 2025. IJCAI.

[28] R. Zviel-Girshin. The Good and Bad of AI Tools in Novice Programming Education. *Education Sciences*, 14(10):1089, Oct. 2024.