

Instruct, Not Assist: Improving Socratic Questioning for Active Multi-Turn Code Debugging

Priyanka Kargupta
pk36

Ishika Agarwal
ishikaa2

Nirav Diwan
ndiwan2

Abstract

We propose TreeInstruct, an instructor agent that is designed to *teach* a student how to debug their programming code. It uses a tree structure for modeling conversational flow and state space estimation for tracking student understanding. Combined, these modules show promising performance. In this report, we outline the motivation, methodology, and experimental sections for our Instructor agent. We are able to show, through empirical evaluation, that our agent is capable of guiding students to debug their code with a moderate amount of turns and a highly Socratic questions.

1 Introduction

1.1 Motivation

Typically, large language models (LLM) are optimized to generate customer-serving, assistant-like responses, which also translates into the types of questions asked. However, this style of questioning can be sub-optimal depending on the specific domain that question generation is applied to, especially educational domains. For instance, if a student is seeking help from an instructor for debugging their code, we consider two forms of potential responses: **assistant-like and instructor-like**. As shown in Figure 1, an assistant-like response would not be a successful educational interaction as it leads to the Assistant directly providing an answer. On the other hand, an Instructor-like response reflects the educational philosophy of *Socratic questioning*.

Socratic questioning is a teaching strategy where the student is *guided* towards solving a problem on their own, instead of being given the *solution directly*. This strategy allows for more effective learning and more meaningful interactions between the student and instructor. Therefore, we aim to train and thereby re-orient an assistant-like LLM to ask Socratic questions that (1) help the user under-



Single-Turn Assistant:

Student: Hey, I really need help. This function is not returning the correct number.

Assistant: I'll be happy to help you. You made an error in the recursive function. Here is the corrected code:

```
def fibonacci(n):  
    if n <= 0:  
        return 0  
    elif n == 1:  
        return 1  
    elif n == 2:  
        return 1  
    else:  
        sequence = fibonacci(n-1)  
        sequence.append(sequence[-1] + sequence[-2])  
    return sequence
```



Multi-Turn Instructor:

Student: Hey, I really need help. This function is not returning the correct number.

Instructor: I'll be happy to help you. What happens when you pass 5 to the function `fibonacci(n)`?

Student: When I run the function with `n` as 5 the output is 8 instead of 5.

Instructor: I see. What do you think might be the cause of this? Can you walk through what happens when you pass 5?

...

Student: Oh, I see. I made an error in the recursive condition. I should change `n-2` to `n-1`.

Figure 1: The Instructor’s goal is generate multi-turn Socratic questions while guiding the student towards the correct solution.

stand their mistakes and (2) do not directly provide the answer.

The motivation behind this work is based on the notion that instructors typically guide the conversation according to an initial estimate of the student’s current knowledge and understanding as reflected by their initial code and its “distance” from the optimal, correct code. We denote all possible paths from the student’s current buggy code to the correct code as the **instruction “state space”** and claim that

Prior approaches (Kazemitabaar et al., 2024; Sheese et al., 2024; Lyu et al., 2024) use a prompt-based approach with the aim of increasing student understanding. Primarily, these approaches use zero-shot prompting, designed to answer the student query on the code within the context of their program.

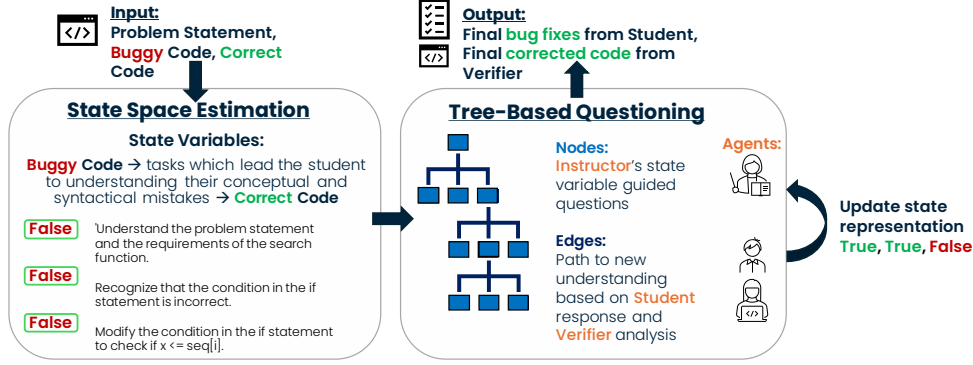


Figure 2: We propose TreeInstruct, a novel tree-guided instructional questioning framework for meaningful educational debugging guidance.

While these approaches are effective in fixing the student code with high success, they are limited in cultivating critical thinking skills. They lack a coherent structure of reasoning that considers the the student’s understanding. Several approaches are also single-turn, lacking the ability to respond to student follow-up questions. Moreover, they are prone to directly revealing code answers. For example, CodeAid (Kazemitabaar et al., 2024) explicitly provides the answer 57% of the time. It achieves a mere 55% rate of effective assistance, with one of the main issues being vague and inadequate explanations to the student queries. Similarly, CodeTutor (Lyu et al., 2024) fails to effectively develop critical thinking skills, primarily due to its lacks the capability to ask relevant follow-up questions.

In contrast with these existing works, we propose **TreeInstruct**, which is comprised of (1) state space and representation estimation and (2) tree-based questioning. Specifically, TreeInstruct dynamically constructs a tree to guide an Instructor’s reasoning for generating socratic questions. The dynamic tree construction allows for a structured multi-turn conversational setting, where based on an estimated state representation and feedback from a Verifier agent, the Instructor can adequately respond to a student’s knowledge either remaining static or progressing.

We explore our framework applied to various models to demonstrate that it can be universally applied to both open and closed source-settings, as well as showcasing our framework’s strong socratic questioning abilities through rigorous human and automatic evaluation over a wide variety of conceptual and syntactical coding problems.

2 Related Works

2.1 Knowledge Tracing

Knowledge Tracing is the act of tracking student knowledge to personalize their learning experience. The tracked components may range from the student knowledge about specific concepts, their behavioral aspects and their tendency to recall concepts. There are primary two methods of knowledge tracing in the literature - (a) Probabilistic and (b) Deep Learning Based.

Probabilistic Knowledge Tracing employs a Hidden Markov Model (HMM) to maintain binary states—learned and unlearned—for each skill as learners interact with exercises. This approach (Corbett and Anderson, 1994) updates the likelihood of these states based on the learner’s performance. However, the binary nature of skill representation and the non-transparent hidden states in HMMs reduce the practical applicability of this method. There are also models that have opened paths to reach a state (Rafferty et al., 2016), but the need to explore their large state spaces makes ineffective in practice.

Deep Learning-Based Methods provide more nuanced state representations, enhancing long-term memory capabilities essential for learning. (Piech et al., 2015) demonstrate the efficacy of LSTM and RNN-based frameworks in online learning environments. However, they require large amounts of annotated data to train limiting their utility in smaller educational settings.

Our methodology addresses the challenge of limited annotated data by dynamically generating states during interactions between instructors and students. We monitor these evolving states through a component we refer to as the Verifier. Using these dynamically generated states, we tailor the educa-

Algorithm 1 TreeInstruct

Require: P (Problem Description), B (Buggy Code), C (Correct Code)

```
1:  $S \leftarrow \text{GenerateState}(P, B, C)$   $\triangleright$  Initialize state representation
2:  $L \leftarrow 0$   $\triangleright$  Initialize tree level
3:  $Q_s \leftarrow []$   $\triangleright$  Initialize Question List at State level  $s$ 
4:  $H \leftarrow []$   $\triangleright$  Initialize Conversation History
5:  $F \leftarrow []$   $\triangleright$  Initialize Bug Fixes
6: while  $\exists \tau \in S$  do  $\triangleright$  Process while tasks remain
7:   while  $L$  is not updated or  $\text{len}(Q_\tau) < k$  do
      $\triangleright$  At max  $k$  candidate questions
8:      $q \leftarrow \text{GenerateQuestion}(\tau, Q_\tau, H)$ 
9:      $r \leftarrow \text{StudentResponse}(q)$   $\triangleright$  Get student response
10:     $v \leftarrow \text{VerifyResponse}(r, \text{prompt } 9)$ 
11:    if  $v = \text{false}$  then
12:      Continue at tree level  $L$ 
13:    else
14:       $H.add(q)$ 
15:       $H.add(r)$ 
16:       $\text{UpdateUnderstanding}(\tau, v)$   $\triangleright$  Update understanding for task  $\tau$ 
17:      if  $\text{IsResolved}(\tau) = \text{false}$  then
18:        Advance to next tree level
19:         $L \leftarrow L + 1$   $\triangleright$  Update tree level
20:      else
21:         $n \leftarrow \text{CountResolvedTasks}(S)$ 
22:         $S \leftarrow \text{UpdateState}(S, \tau)$ 
23:         $L \leftarrow 0$   $\triangleright$  Reset tree level
24:         $L := L + 1$ 
25:         $F \leftarrow \text{ListBugFixes}(H)$   $\triangleright$  Generate bug fixes list
26:         $\text{ApplyBugFixes}(F)$   $\triangleright$  Apply fixes to  $B$ 
27:      end if
28:    end if
29:  end while
30: end while
```

tional experience by personalizing the sequence and type of questions posed to learners.

2.2 Socratic Reasoning in Education:

Prior work (Al-Hossami et al., 2023b,a) has highlighted the poor performance of prompting-based methods in performing Socratic Reasoning for the education domain. Specifically, prompting GPT-3.5 and GPT-4 (Achiam et al., 2023) assistants for

Socratic questioning is largely ineffective. Prompting the models with more examples using Chain-of-thought (CoT) (Wei et al., 2022) Prompting -does not improve the qualitative and quantitative performance. Most glaringly, the GPT assistants give away the direct answer to the student without asking clarifying questions. In instances where the assistant generates a clarifying question, the questions are unrelated to the student response or the original error in the student’s code.

2.3 Focused & Funneling Questions:

Another closely related line of work in the education domain are Focusing and Funneling Questions (Herbel-Eisenmann and Breyfogle, 2005; Wang and Demszky, 2024). Funneling questions aim to enhance overall comprehension of a subject (e.g "what is recursion"?). In contrast, focused questions are detailed questions catered to student understanding and responses. Typically, both funneling and focused questions are asked in Socratic questioning. Notably, prior work (Alic et al., 2022; Demszky and Hill, 2022) introduced a dataset of focusing and funneling questions created from transcripts of interactions between students and teachers in a classroom setting. Each question has a rating between 0 to 5, with higher scores indicating a greater degree of focus. However, the dataset does not contain multi-turn conversations. Moreover, no principled approach has been suggested to create a assistant to ask funneling and focused questions.

3 Methodology

As shown in Figure 2, **TreeInstruct** aims to dynamically guide the multi-turn conversation based on the state space of the conversation. As described in Section 3.2, we estimate a state space based on the optimal goal of our conversation: the student resolving their own conceptual and syntactical errors in a Socratic fashion and reaching the correct code. Additionally, defining a state representation allows us to represent and update the student’s knowledge throughout the interactions.

Furthermore, for traversing through the state space, we employ *tree-based questioning*, explained in Section 3.3. This allows TreeInstruct respond to the student’s current level of understanding adequately. Bringing all these concepts together, 1 contains the pseudocode for our method.

In our framework, we use three agents (all rep-

resented with different language models) that represent a Student, an Instructor, and a Verifier. The details of these agents are outlined in Section 3.1.2.

3.1 Preliminaries

3.1.1 Problem Description

A natural use-case where Socratic questioning is employed is for code debugging. Consider the following scenario: a Student is given a coding problem description/statement. After attempting the problem, they develop an initial code that contains errors. To help the Student arrive at their own optimal solution, the Instructor employs Socratic questioning - asking clarifying questions catered to the current response, encouraging the Student to arrive at the solution. Ultimately, our framework’s expected output is the Student to generate a list of bug fixes based on their Socratic interactions with the Instructor.

3.1.2 Agents

The *Student agent* is based on the Mistral-7B-Instruct model. We use a smaller model to mimic the abilities of a student, therefore, its job is to respond to the Instructor’s questions. The persona prompt is outlined in 6.

In real life, an educator (e.g., an instructor, a teaching assistant, a professor) has two jobs: (1) to ask relevant questions to their students, and (2) to assess the student’s understanding. Following this, we break down our educator into two roles: an Instructor and a Verifier, with persona prompts of 5 and 7, respectively. The *Instructor agent* is backed by the Claude 3 Sonnet model and its job is to generate questions to ask the Student. The *Verifier agent* is also backed by the Claude 3 Sonnet model, but it has a much more involved role. It will estimate - and update - the state space and identify the gaps (or lack thereof) in the Student’s understanding.

3.2 State Space Estimation

The goal of the state space estimation is to keep track of the student’s global understanding, such that from the initial buggy state, we can traverse the space to reach the optimal goal state (correct code).

We define the state space as the set of all possible tasks that a Student could perform based on their buggy code. We define the representation of the optimal state as the series of tasks which leads

the student from their buggy code to understanding their conceptual and syntactical mistakes and correcting their code. Each of these tasks is a **state variable** which either has a value of *True* or *False* based on whether or not the Student has completed it. At the very beginning of the Instructor-Student conversation, all these variables are set to “False”.

1. **False**, *Understand the problem statement and the requirements of the search function.*
2. **False**, *Recognize that the condition in the `if` statement is incorrect.*
3. **False**, *Modify the condition in the `if` statement to check if $x \leq seq[i]$.*

The state variables are structured such that earlier tasks have a higher priority, as their completion may consequently resolve later tasks. For instance, a student’s buggy code may reflect that they do not understand an edge case mentioned in the problem statement. However, once this misunderstanding is resolved, the Student may simultaneously correct their related syntactical mistakes. On the other hand, attempting to resolve the syntactical mistakes, “Modifying the condition in the `if` statement”, beforehand may lead to an unproductive and less structured conversation overall.

3.3 Tree-Based Questioning

Using a tree-based questioning framework helps to structure the flow of the conversation. In this tree, (1) each of the nodes are questions, (2) each of the parent-child edges connect nodes that guide to new understanding, and (3) sibling nodes reflect questions which aim to *sequentially* solidify the current misunderstanding. Guided by the state representation in Section 3.2, each level in the tree has questions of a similar difficulty and depth; the last level of the tree indicates that a specific state attribute has been resolved. The Verifier agent dictates the movement from level to level and tree to tree.

Conditional Generation of Sibling Nodes. Questions at the same level will be *conditionally generated* if and only if the Student incorrectly answers the Instructor question. These questions must lead to the same knowledge, therefore, they can be rephrased, and/or remind the Student of knowledge that they already know. To ensure this, we ground the question generation based on two things: (1) the previous questions from the same

level, and (2) the Verifier’s explanation for why the Student got the question wrong.

3.4 Updating the State Representation

Once the Student answers one of the Instructor’s questions correctly, the state space needs to be updated. With the Instructor question, the Student response, and the target state as input, we use the Verifier agent to determine whether the target state/task has been resolved. We repeat this process until either (1) a task is deemed unresolved, or (2) all tasks in the state space have been resolved. It is possible that even after an Instructor-Student interaction, a task has not been resolved. In this case, we will still advance to the next tree level, which means the next set of questions will be slightly more advanced.

If at least one task has been resolved, then the code needs to be regenerated. Given the entire conversation history with the Instructor, the Student is prompted to provide a list of bug fixes (e.g. "Replace ‘i’ with ‘i+1’ on line 6."), which the Verifier agent will apply. This code will be used as the Student’s code in future interactions. The reason we do not ask the Student to generate the code themselves is because the Mistral-7B-Instruct language model is not well known for generating code. We also found that in certain cases, Mistral-7B-Instruct will generate correct bug fixes that are not present in the conversation history.

4 Evaluation

4.1 Dataset

We use the Socratic Debugging Benchmark dataset from (Al-Hossami et al., 2023c). This dataset contains 149 data instances. Each data instance has the description of the problem, the buggy code written by the user, the bug description and corresponding fixes, a few unit tests, a conversation between the user and instructor, and the final, fixed code. The conversation is a multi-turn conversation where the instructor only asks Socratic questions to the user.

4.2 Baselines

Our main baseline is a prompting based method that initializes the teacher to ask Socratic questions. We do not guide our conversation with the state space representation, nor do we use the tree-based questioning. We keep the interaction between the teacher and student limited to 3 turns. We experiment with various models for the Instructor/Verifier and Student roles:

1. **Baseline** (A, M) uses Claude 3 Sonnet for the Instructor/Verifier and Mistral-7B-Instruct for the Student,
2. **Baseline** (A, A) employs Claude 3 Sonnet for the Instructor/Verifier and Claude 3 Sonnet for the Student.

4.3 Ablations

We ablate the model based on the conditional generation of the candidate questions:

1. - **Conditional Siblings** (A, M) generates the k sibling candidate questions at once using only the target state variable. This is in contrast to our main method, which dynamically generates the candidate questions based on the prior sibling questions at the same level, the Student’s respective responses and Verifier analysis.
2. - **State Representation** (A, M) generates the k candidate questions at once but does not use the target state variable or state representation at all for tree-based questioning. Instead, initially, the candidate questions are conditionally generated by contrasting the Student explanation of their line of buggy code with the Teacher’s explanation of their correct code. For the subsequent levels, the candidate questions are generated only based on the previous level’s questions and Verifier analysis of the Student’s respective answers.

We ablate the model performance on 45% of our complete evaluation dataset which is 68 code problems.

4.4 Experimental Setup

We perform qualitative and quantitative evaluation of our methods.

Qualitative: We develop a Likert Scale to assess the Socratic quality of questions. Previous research identifies multiple dimensions of Socratic questioning, including relevance to specific needs, implicitness of the answer, and structural coherence. For each question, we measure the below attributes of the conversation manually:

- **Relevance:** The instructor’s question was pertinent to the errors in the student’s code.
- **Indirectness:** The instructor’s question refrained from directly revealing solutions to the bugs.

Table 1: Detailed performance metrics for the conceptual and syntactical bug types, including average Socratic scores (AS), average number of turns (AT), and success rates (SR). Metrics compare baseline methods with instructional methods across different system iterations. Through side-by-side comparison, It was found that **our method was preferred 77.94% of the time over the Baseline (A, M)** .

Methods	Conceptual			Syntactical		
	AS	AT	SR	AS	AT	SR
Baseline (A, M)	2.47	3.00	71.43%	2.50	3.00	61.90%
Baseline (A, A)	2.53	3.00	92.86%	2.49	3.00	85.71%
TreeInstruct (A, M)	2.83	8.95	73.81%	2.93	9.58	78.09%

- **Logical Flow:** The instructor’s question promoted a coherent conversation, facilitating the student’s problem-solving process.

We use the Likert Scale to evaluate each question by assessing adherence to predefined attributes (0 indicates none of the attributes are met, while 3 signifies all attributes are met). Next, we qualitatively evaluate the performance of our framework using the **Average Socratic Score (AS)**. To quantitatively measure the Socratic quality of a conversation, we compute the average Likert rating for all questions posed during the conversation. This score provides a single metric that reflects the overall effectiveness and adherence to the Socratic method as implemented in the questioning technique.

Quantitative: In addition to qualitative measures, we apply quantitative metrics to objectively evaluate the effectiveness and efficiency of our framework.

1. **Overall Success Rate (SR):** We check whether the final code generated by the student passes all the assertion tests in the database. The success rate is calculated as the ratio of tests passed to the total number of tests.
2. **Average Number of Turns (AT):** We calculate the average number of turns required by the method to successfully reach the code generation. This metric provides insight into the efficiency and depth of the interaction process.

We perform this evaluation on the full dataset of 149 questions.

Side-by-Side evaluation: We also perform a side-by-side evaluation that aims to measure the number of times a user prefers our Instructor-Student conversations (from TreeInstruct) over

those from our Baseline (A, M) method. The evaluation was a double-blind study where two conversations on the exact same problem were displayed to the evaluator. The evaluator does not know which conversation came from which Instructor agent. They were tasked to choose which conversation they prefer. Criteria includes length of conversations, ability to stick on topic of the relevant bug fixes, Socratic nature of questions, and general flow of conversation, among others.

5 Experimental Results

5.1 Overall Results

Table 1 shows that our method is quite competitive against the baselines. The Socratic score of TreeInstruct (A, M) shows a significant increase of 10.5-17.2% (conceptual and syntactical respectively) from the best baseline. Comparing the results of TreeInstruct (A, M) against Baseline (A, A) highlights the effect of a strong Student - a stronger Student can lead to a higher success rate. Specifically, because Baseline (A, A) exploits a 70B parameter, closed-source model as the Student—as opposed to TreeInstruct (A, M) utilizing a 7B parameter, open-source model—the Student (despite its role definition) is able to immediately point out the problems within the first turn within the its buggy code regardless of the Instructor’s question. Hence, the **success rate of Baseline (A, A) is seemingly independent of the generated questions**. We qualitatively witness this in the lower Socratic nature of the baseline questions—often these questions explicitly state or directly hint at the solution, failing our indirectness check, but still passing the relevance and logical flow check. Nonetheless, when compared with the same (weaker) Student, TreeInstruct (A, M) is demonstrably better, having a 2.28% and 16.1% improvement over Baseline

Table 2: Detailed performance metrics for the conceptual and syntactical bug types, including average Socratic scores (AS), average number of turns (AT), and success rates (SR). Metrics compare baseline methods with instructional methods across different system iterations on **ablations which were run on a subset of the data**.

Methods	Conceptual			Syntactical		
	AS	AT	SR	AS	AT	SR
TreeInstruct-Subset (A, M)	2.85	5.89	81.48%	2.81	9.35	77.78%
- Conditional Siblings (A, M)	2.89	13.59	77.78%	2.95	13.67	82.05%
- State Representation (A, M)	2.77	2.96	53.85%	2.76	4.67	48.83%

(A, M). Furthermore, through our side-by-side comparison, it was found that **our method was preferred 77.94% of the time over the Baseline (A, M)**.

5.2 Ablation Studies

Table 2 displays the effect of each of the modules on a subset of the dataset (45%), where the performance steadily drops as each component is removed. We can see that by completely removing the estimation and use of the state space and representation, the model performance drops significantly (27.63% and 28.95% for conceptual and syntactical respectively). This shows that our state space estimation and representations are crucial for quality tree-based questioning.

We can also see that by modifying the sibling question generation from parallel (- Conditional Siblings (A, M)) to conditionally sequential (TreeInstruct-Subset (A, M)) demonstrates a slight trade-off between average socratic score (AS) and the average number of turns (AT). For - Conditional Siblings (A, M), the questions at the same level are more directed towards the Student’s previous incorrect responses towards a similar question and hence run the risk of being less socratic. On the other hand, this leads to a smaller number of turns. Overall, this reflects only a slight difference in performance.

5.3 Observations

During the human evaluation, there were a few discoveries that further explained the performance of these methods.

Student quality affects number of turns. We found that Mistral-7B-Instruct is not a very good student simulator with several conversations having over 50+ turns. Specifically, in many instances, the Student got confused, or provided incomplete

answers. Consequently, the Instructor did not move on until the Student got the correct answer. This led to larger conversations with the same question being rephrased multiple times to the Student.

High question quality. Next, we saw many cases where the success rate was low, but the question quality was high. This means that the Instructor asked good questions, but those good questions were not able to lead to debugged code. This could be due to the language model’s (Mistral-7B) poor capabilities in extracting bug fixes out of conversational text. Hence, the bug fixes will be compromised, and the success rate will drop. This further means that just because the success rate is low, does not mean the questions asked by the Instructor are bad.

TreeInstruct (A, M) is more Socratic. Finally, TreeInstruct (A, M) is more Socratic than the baselines. The average Socratic score for the Tree-Instruct (A, M) is much higher than the baselines. This difference is more prominent for syntactic problems than conceptual ones. Additionally, many questions asked by the Baseline Instructors directly mentioned the state of the program – explaining their low Socratic score.

Syntactic problems require higher number of turns for resolution. Syntactic bugs often need more interactions to resolve than Conceptual bugs. This is likely because syntactic problems generally contain multiple bugs. In contrast, conceptual problems typically involve a single bug. Therefore, resolving syntactic bugs usually requires more interaction between Student and Instructor compared to conceptual bugs.

5.4 Case Study

We present a case-study of a problem to compare the interaction between the Instructor and Student

in Table 3 and Table 4. While both the interactions achieve in fixing the student code, the TreeInstruct Instructor interacts with the student by asking them Socratic Questions. Specifically, in the baseline interaction the Instructor explicitly mentions the line number where the bug may be present and prompts the student to rectify the error. Whereas, in the TreeInstruct case study, the Instructor asks questions in a structured way using Socratic Questioning to help the student arrive at the fix themselves.

6 Conclusion

In this report, we introduce an instructor agent called TreeInstruct that will help students learn to debug their programming code. Our experiments identified several limitations of TreeInstruct that warrant further exploration. First, modeling off of real life, deciding when the Instructor should *teach* the Student something rather than simply asking questions can be a helpful feature. Secondly, the current dataset represents bugs as independent events, which does not reflect their interdependent nature observed in practical scenarios. Future work should consider developing a model that encapsulates a joint state space representing the collective nature of bugs.

References

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.
- Erfan Al-Hossami, Razvan Bunescu, Justin Smith, and Ryan Teehan. 2023a. Can language models employ the socratic method? experiments with code debugging. *arXiv preprint arXiv:2310.03210*.
- Erfan Al-Hossami, Razvan Bunescu, Ryan Teehan, Laurel Powell, Khyati Mahajan, and Mohsen Dorodchi. 2023b. Socratic questioning of novice debuggers: A benchmark dataset and preliminary evaluations. In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 709–726.
- Erfan Al-Hossami, Razvan Bunescu, Ryan Teehan, Laurel Powell, Khyati Mahajan, and Mohsen Dorodchi. 2023c. [Socratic questioning of novice debuggers: A benchmark dataset and preliminary evaluations](#). In *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 709–726, Toronto, Canada. Association for Computational Linguistics.
- Sterling Alic, Dorottya Demszky, Zid Mancenido, Jing Liu, Heather Hill, and Dan Jurafsky. 2022. Computationally identifying funneling and focusing questions in classroom discourse. *arXiv preprint arXiv:2208.04715*.
- Albert T Corbett and John R Anderson. 1994. Knowledge tracing: Modeling the acquisition of procedural knowledge. *User modeling and user-adapted interaction*, 4:253–278.
- Dorottya Demszky and Heather Hill. 2022. The ncte transcripts: A dataset of elementary math classroom transcripts. *arXiv preprint arXiv:2211.11772*.
- Beth A Herbel-Eisenmann and M Lynn Breyfogle. 2005. Questioning our patterns of questioning. *Mathematics teaching in the middle school*, 10(9):484–489.
- Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Z Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. *arXiv preprint arXiv:2401.11314*.
- Wenhan Lyu, Yimeng Wang, Tingting Rachel Chung, Yifan Sun, and Yixuan Zhang. 2024. Evaluating the effectiveness of llms in introductory computer science education: A semester-long field study. *arXiv preprint arXiv:2404.13414*.
- Chris Piech, Jonathan Bassen, Jonathan Huang, Surya Ganguli, Mehran Sahami, Leonidas J Guibas, and Jascha Sohl-Dickstein. 2015. Deep knowledge tracing. *Advances in neural information processing systems*, 28.
- Anna N Rafferty, Emma Brunskill, Thomas L Griffiths, and Patrick Shafto. 2016. Faster teaching via pomdp planning. *Cognitive science*, 40(6):1290–1332.
- Brad Sheese, Mark Liffiton, Jaromir Savelka, and Paul Denny. 2024. Patterns of student help-seeking when using a large language model-powered programming assistant. In *Proceedings of the 26th Australasian Computing Education Conference*, pages 49–57.
- Rose E Wang and Dorottya Demszky. 2024. Educonvokit: An open-source library for education conversation data. *arXiv preprint arXiv:2402.05111*.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Appendix A Prompts

A few of the prompts use one-shot learning, and the fields are prefixed with "example". These examples are hand chosen, with no criteria in mind. The example problem relates to a solution that outputs the Fibonacci sequence of length n , where n is the input.

Table 3: Example of a bad Baseline (AI, M) conversation.

Problem	Write a function 'search(x: int, seq: List[int]) ->int' that returns the index of the first occurrence of 'x' in 'seq'. If 'x' is not in 'seq', return the index where 'x' should be inserted to keep 'seq' sorted. Assume that 'seq' is sorted in ascending order
Example	search(5, [-1, 5, 8, 10, 12]) =>1
Teacher	What should the function return if x is equal to an element in seq?
Student	If x is equal to an element in seq, the function should return the index of the first occurrence of x in seq. The current code only checks if x is less than seq[i], but it should also check for equality to handle the case when x is present in seq.
Teacher	What additional condition should be added to the if statement on line 3 to handle the case when x is equal to an element in seq?
Student	We should add the condition "or x == seq[i]" to the if statement on line 3 to return the index i when x is found in seq. This will handle the case when x is equal to an element in seq and return its index.

Table 4: Example of a good TreeInstruct (AI, M) conversation.

Problem	Write a function 'search(x: int, seq: List[int]) ->int' that returns the index of the first occurrence of 'x' in 'seq'. If 'x' is not in 'seq', return the index where 'x' should be inserted to keep 'seq' sorted. Assume that 'seq' is sorted in ascending order
Example	search(5, [-1, 5, 8, 10, 12]) =>1
Teacher	Can you provide an example where the current code fails to return the correct index for an element that is present in the sorted list 'seq'?
Student	If 'x' is the smallest element in the list 'seq', the code will not return the correct index.
Teacher	That is correct! Let's take a step further. Consider the case where 'x' is equal to an element in 'seq' at some index 'i' other than the first index (0). Can you provide an example where the current code would fail to return the correct index for 'x' in that situation?
Student	If 'seq' contains 'x' at some index 'i > 0', the current code would incorrectly return 'i-1'. This is because the code breaks the sorted order of sequence once 'x' is found.

You are an Instructor helping a Student debug their code to solve the following problem statement (after tag 'problem'). You have access to their buggy code (after tag 'bug code'). Do not ask questions that explicitly or implicitly mention the following:

Table 5: Instructor agent persona prompt

You are a Student writing code to solve the following problem statement (after tag 'problem') and have the following buggy code (after tag 'bug code'). You are seeking help from your Instructor help solve the bug.

Table 6: Student agent persona prompt

You are an assistant to the Instructor helping a Student debug their code to solve the following problem statement (after tag 'problem'). Your role is to determine the Student's understanding (or lack thereof) within the Instructor-Student interactions. You have access to the correct code (after tag 'correct code').

Table 7: Verifier agent persona prompt

Given the student's buggy code and the correct code for solving the problem statement, we define the optimal state of a set of Instructor-Student interactions as a series of tasks which leads the student to understanding their conceptual and syntactical mistakes within the input buggy code. We define a state representation as a list of state attributes, where each attribute denotes a specific task that is **NECESSARY** for the student to successfully understand and implement the given problem. If the student's buggy code shows that they have already understood and implemented a concept that is used for the correct code, do **NOT INCLUDE** the concept/task as a state attribute as it is **REDUNDANT**. The list should be ordered, with earlier attributes/tasks given priority over later ones (e.g., conceptual understanding tasks are a prerequisite and thus more important than syntactical tasks). The following is an example of the state representation for the given example problem statement: example problem: Implement a Fibonacci sequence using recursion. {example buggy code} {example correct code} {example state representation}. Now do the same for the following problem statement, correct code, and student buggy code: {problem statement}, {correct code}, {buggy code}

Table 8: Internal Verifier prompt to estimate the state space representation.

Instructor: {Instructor question}
Student: {Student response}
Based on the conversation history, do you believe that the Student correctly responded to your question?

Table 9: Internal Verifier prompt to assess the accuracy of the Student response with respect to the Instructor's question.

Instructor: {Instructor question}
Student: {Student response}
Based on the conversation history between the Student and Instructor, do you believe the Student has reached the below level of understanding?
target understanding: {Target State Task}

Table 10: Internal Verifier prompt to update the state space with respect to a single-turn Instructor-Student interaction.

We define a state representation as a list of state attributes, where each attribute denotes a specific task that is **NECESSARY** for the student to successfully understand and implement the given problem. Based on the conversation that you have had with the Instructor (under tag "conversation history") and on these resolved state representation attributes (under tags "resolved state attributes"), can you suggest a list of bug fixes (described in English) to make that reflects the changes mentioned in the conversation? An example of a bug fix would be: "Replace 'i' with 'i+1' on line 6."
conversation history: {convo history}
resolved state attributes: {state representation}
n is the number of bug fixes to make. Format the changes in the following YAML format, add it right after the colon without newlines:

Table 11: Instructor to Student prompt that asks the Student to generate a list of bug fixes.

Given a set of bug fixes (under tag "student bug fixes"), you should apply each bug fix in the set to the buggy code (under tag "buggy code"). Below is an example of buggy code, bug fixes, and your expected example output code (under tag "example corrected code") where the bug fixes are applied: {example buggy code} {example bug fixes} {example corrected code} Using this same format, can you apply the following bug fixes (under tag "student bug fixes") to the below buggy code (under tag "buggy code")? Your output code with the bug fixes applied should be under the "corrected code" tag.
{buggy code}

Table 12: Internal Verifier prompt to apply the given bug fixes to the given buggy code.