Instruct, Not Assist:
LLM-based Multi-Turn Planning and Hierarchical Questioning for Socratic Code Debugging

Anonymous ACL submission

Abstract

Socratic questioning is an effective teaching strategy, encouraging critical thinking and problem-solving. The conversational capabilities of large language models (LLMs) show great potential for providing scalable, real-time student guidance. However, current LLMs often give away solutions directly, making them ineffective instructors. We tackle this issue in the code debugging domain with **TreeInstruct**, an Instructor agent guided by a novel state space-based planning algorithm. TreeInstruct asks probing questions to help students independently identify and resolve errors. It estimates a student's conceptual and syntactical knowledge to dynamically construct a question tree based on their responses and current knowledge state, effectively addressing both independent and dependent mistakes concurrently in a multi-turn interaction setting. In addition to using an existing single-bug debugging benchmark, we construct a more challenging multi-bug dataset of 150 coding problems, incorrect solutions, and bug fixes– all carefully constructed and annotated by experts. Extensive evaluation shows TreeInstruct's state-of-the-art performance on both datasets, proving it to be a more effective instructor than baselines. Furthermore, a real-world case study with five students of varying skill levels further demonstrates TreeInstruct's ability to guide students to debug their code efficiently with minimal turns and highly Socratic questioning.

1 Introduction

With the rapidly expanding conversational and reasoning abilities of large language models (LLMs), there has been a substantial rise in demand for exploiting their capabilities within a multitude of educational applications (Kasneci et al., 2023) in order to widen accessibility to personalized feedback. Specifically, several recent works explore the use of LLMs for providing feedback and guidance to students (Wang et al., 2023; Kazemitabaar et al., 2024; Sheese et al., 2024; Lyu et al., 2024). However, LLMs are typically optimized to generate customer-serving, assistant-like responses, which also translates into the types of questions asked. This style of questioning can be sub-optimal depending on the specific domain that question generation is applied to, especially educational domains (Cotton, 1988; Sahamid, 2016; Yang et al., 2005; Wilson, 1987). For instance, if a student is seeking help from an instructor for correcting their mistakes (e.g., debugging their buggy 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*.

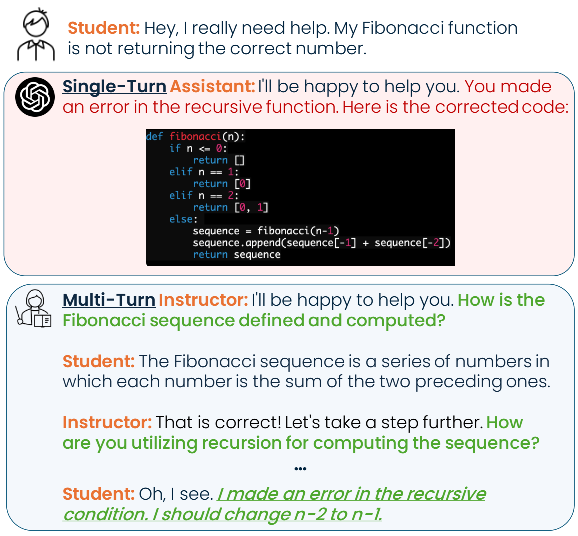

Figure 1: The Instructor's goal is to generate multi-turn Socratic questions while guiding the Student towards the correct solution.

Socratic questioning is a teaching strategy where the Student independently solves their problem by answering *guiding* questions, instead of being given the *solution directly* (Wilson, 1987). This is a more effective learning strategy because the weight of learning falls on the Student as they must put in effort to answer a question as opposed to solely relying on the model (Cotton, 1988; Kasneci et al., 2023). Therefore, we aim to re-orient an LLM to be an Instructor, not an assistant, by asking Socratic questions that (1) help the Student understand their mistakes, and (2) do not directly provide the answer. To tackle these challenges, we propose **TreeInstruct** based on the following principles:

1. **State space estimation:** An Instructor plans its conversation with a Student based on the "distance" between their initial answer and the optimal, correct answer within the estimated state space. In other words, it tracks the knowledge state of the Student within this space throughout the Instructor-Student interactions.
2. **Tree-based Socratic questioning:** An Instructor generates turn-level Socratic questions conditioned on both the Student's current knowledge state *and* misunderstanding(s), the latter derived from their responses to the Instructor's questions. This step dynamically constructs a Socractic question tree.
3. **Adaptive conversation restructuring:** An Instructor updates their initial conversation plan based on how the Student is progressing in the conversation, as reflected by updates (or lack thereof) to the Student's knowledge state. This planning can include both *questioning* and *teaching* actions.

While these principles can apply to many educational domains, this paper focuses on code debugging, which presents unique challenges. Real-world code debugging often involves multiple, potentially interdependent conceptual and syntactical bugs. For instance, Figure 1 shows that first resolving the Student's conceptual misunderstanding of recursion in Fibonacci helps them identify their recursive syntactical bug (Figure 1). However, existing work fails to account for such nuances and assumes single-turn feedback (Kazemitabaar et al., 2024; Wang et al., 2023; Lyu et al., 2024). This ignores the sub-steps required for the Student to understand each bug.

In contrast, TreeInstruct constructs a multi-turn debugging plan (*state representation*), defined as the set of Student misunderstandings and mistakes (*state variables*) to be resolved in order to comprehend and correct their bug(s). We define all potential paths to complete these tasks as the *state space*. We traverse the space using Socratic questions and trace which variables have been resolved, grounded based on the Student's responses.

While existing LLM-based tutors are effective in fixing the Student's code with high success, they are either prone to directly revealing code answers or cannot be adapted to new Student responses. For example, CodeAid (Kazemitabaar et al., 2024) directly reveals the code answer and provides code 57% of the time. It achieves a mere 55% rate of helpfulness. On the other hand, TreeInstruct exploits the state space to dynamically construct a tree of questions based on (1) incorrect Student responses, or (2) gaps in the Student's knowledge. The sibling and parent-child relationships between questions reflect the manner in which they traverse the state space. Finally, it exploits both the Student's knowledge state and any proposed bug fixes to serve as the dynamic stopping condition. Overall, TreeInstruct takes a more structured approach to multi-turn conversational feedback, as (1) grounding the conversation on the state space representation ensures that all bugs are sufficiently addressed, and (2) constructing a tree based on the Student's current level of understanding allows for more relevant and personalized question generation.

We summarize our contributions below:

- To the best of our knowledge, TreeInstruct is the first work to explore state space estimation and dynamic tree-based questioning for multi-turn Socratic instruction.
- We construct a novel challenging multi-bug debugging dataset with 150 expert-annotated conceptual and syntactical problems and buggy solutions/fixes.
- Extensive experiments on an existing benchmark and our constructed dataset demonstrate that TreeInstruct can be universally applied to both open and closed source-settings.
- We also showcase that TreeInstruct's strong Socratic questioning abilities widely outperform all baselines through both (1) rigorous quantitative and qualitative expert evaluation (on average, preferred over 78.43% of the time) and (2) real-world interactions with students of varying coding abilities.

Reproducibility: We release our data and source code[1] to facilitate further studies.

[1]https://anonymous.4open.science/r/TreeInstruct

Algorithm 1 TreeInstruct

Require: P (Problem Description), B (Buggy Code, Bug Descriptions), C (Corrected B Code, Bug Fixes)

$S = \{\tau_1, \tau_2, \ldots, \tau_k\} \leftarrow$ GenerateState(P, B, C) ▷ Section 3.2: State representation: (resolved?, task)
$l \leftarrow 0, Q \leftarrow \{l : []\}, H \leftarrow [], F \leftarrow \{\}$ ▷ Tree level, question list/level, conv. history, Student bug fixes
$q \leftarrow$ GenerateQuestion(P, B, C, τ_1) ▷ Section 3.3: Generate initial question

while $\exists\, \tau \in S$ s.t. $\neg$isResolved (S, F, C) **do** ▷ Section 3.4: Process while tasks or bugs remain
    $r \leftarrow$ StudentResponse(q)
    $v, w \leftarrow$ VerifyResponse(q, r) ▷ Section 3.3: is r to q correct (v); why or why not (w)?
    $H.add(q, r)$
    $Q[l].add(q)$
    if $v =$ false **then** ▷ **Incorrect student response**
        $q \leftarrow$ GenerateSiblingQuestion$(\tau, Q[l], H, w)$ ▷ Section 3.3: factor in *why* the student was incorrect
    else ▷ **Correct student response**
        $S, w \leftarrow$ UpdateUnderstanding(S, q, r) ▷ Section 3.3: tasks $\tau_i \ldots \tau_k$ resolved? If $\neg$S$[\tau]$, why (w)?
        if $\neg$S$[\tau]$ **then**
            $q \leftarrow$ GenerateChildQuestion$(\tau, Q[l], H, w)$ ▷ Section 3.3: factor in *why* τ was unresolved
            $l \leftarrow l + 1$ ▷ Advance to next tree level
        else ▷ **Task τ resolved**
            $F \leftarrow$ GetStudentBugFixes(H) ▷ Section 3.4: ask Student for bug fixes (if any)
            $l \leftarrow 0, Q \leftarrow \{l : []\}$ ▷ τ resolved $\rightarrow$ create new tree

2 Related Works

2.1 Knowledge Tracing

Knowledge tracing tracks student knowledge to personalize their learning experience, including understanding specific concepts, behavior, and recall ability. There are two primary methods: probabilistic and deep learning-based. Probabilistic knowledge tracing, as it was first introduced, uses a Hidden Markov Model (HMM) to maintain binary states, learned and unlearned, for each skill as learners engage with exercises. This approach, from which we draw inspiration, updates the likelihood of these states based on performance (Corbett and Anderson, 1994; Yudelson et al., 2013). Some models use open-ended paths to states (Rafferty et al., 2016), while others use deep learning-based, long-term memory capabilities essential for learning (Piech et al., 2015). These methods are performative, but such state spaces hinder effectiveness and require large amounts of annotated training data.

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

2.2 Socratic Reasoning in Educational AI

There have been several works exploring Socratic reasoning in education (Herbel-Eisenmann and Breyfogle, 2005; Wang and Demszky, 2024; Alic et al., 2022; Demszky and Hill, 2022). More recently, 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 (Achiam et al., 2023), even with Chain-of-Thought (CoT) (Wei et al., 2022), as they often give away answers without asking clarifying questions, or the questions are unrelated to the student's response or original bug (Achiam et al., 2023). In contrast, TreeInstruct mitigates this issue by explicitly grounding the question generation step on both a target state variable τ and any Student *misunderstanding* gauged from their previous response.

2.3 LLMs for Interactive Education

Recent generative approaches within the AI tutoring space have attempted to generate responses which cater to the student's type of mistake or request, but only in single-turn settings. CodeAid (Kazemitabaar et al., 2024) is an *assistive* tool that helps students debug their code. However, the Instructor is specifically designed to directly provide single-turn responses to the Student, such as answering student questions, explaining concepts, and helping to write code. In contrast, TreeInstruct aims to *instruct* the Student socratically through questions. BRIDGE (Wang et al., 2023) is an Instructor-like framework that aims to help students with math mistakes. The LM estimates the type of error, the strategy of error remediation, and the instructor intention behind the remediation (all are chosen from a predetermined set). However, our methodology makes use of a more structured approach that works well for multi-turn conversations while adapting to the Student's learning.

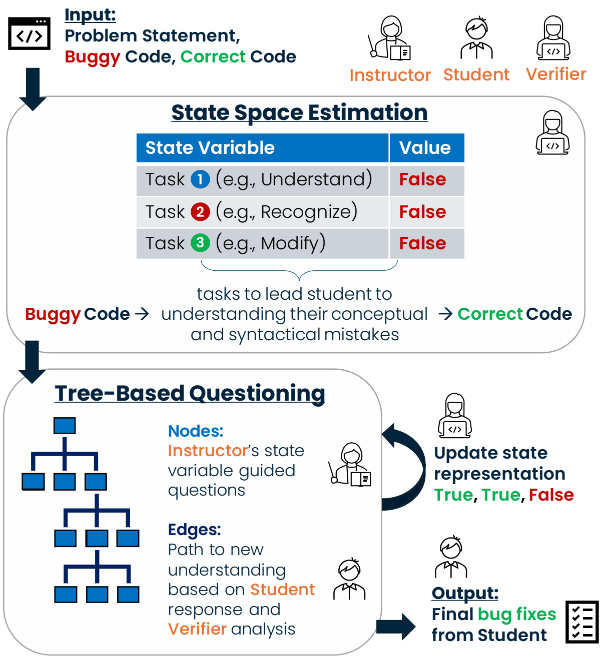

Figure 2: We propose TreeInstruct, a novel tree-guided instructional questioning framework for meaningful educational debugging guidance.

3 Methodology

As shown in Figure 2, **TreeInstruct** aims to dynamically guide the multi-turn conversation based on its estimated state space. Section 3.1.2 provides an overview of the three different agents we use and their respective roles during the state space generation/update and tree construction processes (outlined in Figure 3). This allows TreeInstruct to respond to the Student's current level of understanding adequately. Algorithm 1 contains the pseudocode for all components in our method.

3.1 Preliminaries

3.1.1 Problem Description

As input, the Instructor is given the Student's buggy code that contains e errors, a problem statement, bug descriptions, and their respective fixes. The Instructor guides the Student to generate a list of all bug fixes based on their interactions with the Instructor. The overall goal is for the Student to resolve their own conceptual and syntactical errors in a Socratic fashion to reach the correct code.

3.1.2 Agents

In a real-world setting, a Socratic educator (e.g., an instructor, a teaching assistant, a professor) executes two tasks when interacting with a Student: (1) ask relevant questions to the Student, and (2) assess the Student's understanding based on their responses. Following this cyclical pattern, we break down our educator into two roles: an Instructor and a Verifier, with persona prompts specified in Tables 9 and 11 in Appendix I, respectively. The Instructor and Verifier perform their respective tasks specified in Algorithm 1 via zero and one-shot prompting. The *Instructor* agent's job is to generate questions to ask the Student (GenerateQuestion, GenerateSiblingQuestion, and GenerateChildQuestion in Alg. 1; details provided in Section 3.3). The *Verifier* agent has a significantly more involved role:

1. *State space estimation (Section 3.2):* The Verifier determines a set of tasks which will lead a Student's to understanding and correcting their problem and buggy code. This is `GenerateState` in Alg. 1.

2. *Assess Student Response (Section 3.3):* Once the Student answers the Instructor's question, the Verifier must judge the response's accuracy, given the question-answer pair interaction. This is `VerifyResponse` in Alg. 1.

3. *Assess Student Understanding of Target State Variable (Section 3.3):* To update the Student's state space representation, the Verifier must determine whether the Student would have needed a sufficient understanding of the target state variable in order to generate their response. This is `UpdateUnderstanding` in Alg. 1.

4. *Verify Student Bug Fixes (Section 3.4):* Each time the Student understands a target state variable, they are asked to provide, if any, recommended bug fixes based on the conversation history. This serves as an early stopping condition. This is `isResolved` in Alg. 1.

3.2 State Space Estimation

The goal of state space estimation is to determine the optimal criteria to track a Student's global understanding of a problem P and their code, such that from the initial buggy state B, we can traverse the space to reach the goal state (correct code C).

We define the state space as the set of all possible tasks that a Student could perform to correct their buggy code. We claim that the optimal state space can be represented by a series S of k tasks which leads the Student from their buggy code B

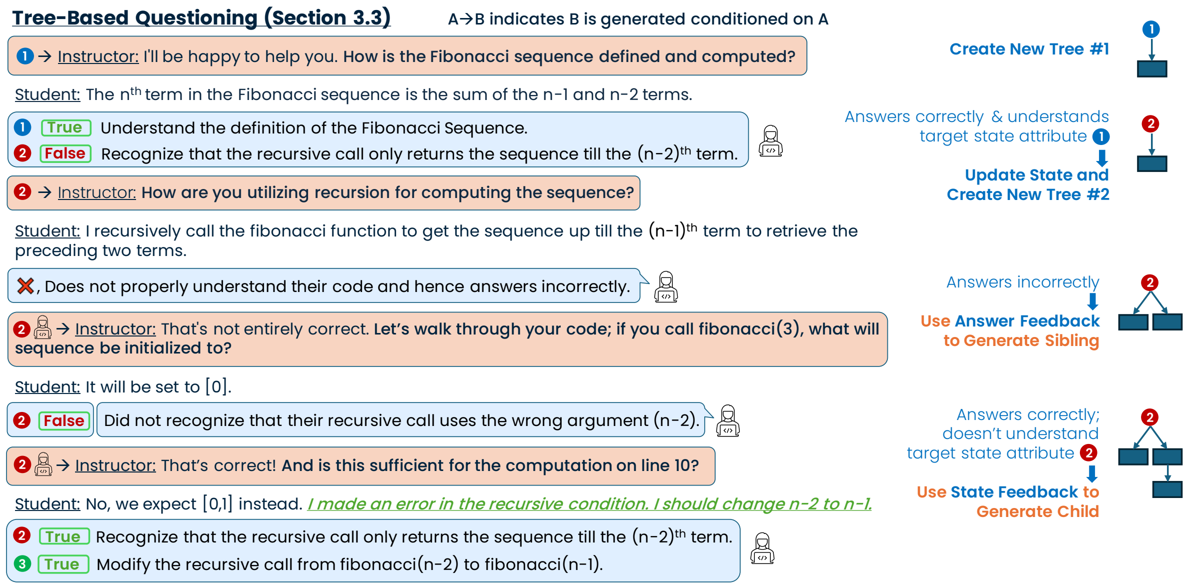

Figure 3: We detail the process for tree-based question generation. Blue and orange text/backgrounds indicate that the Instructor and Verifier are performing the task respectively.

to (1) understanding their conceptual and syntactical mistakes and (2) correcting their code. Each of these tasks is a **state variable** τ_i 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 of these variables are set to "False". We provide the estimated state space used in Figures 2 and 3.

1. τ_1: {**False**, *Understand the definition of the Fibonacci Sequence.*}
2. τ_2: **False**, *Recognize that the recursive call only returns the sequence till the* $(n-2)^{th}$ *term.*
3. τ_3: **False**, *Modify the recursive call from fibonacci(n-2) to fibonacci(n-1).*

The state variables τ_i 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

Tree-based questioning helps to structure the logical flow of the conversation and allows for more relevant, personalized questions. We use a tree to encode the Student's path to understanding at least one specific target state variable τ_i. In each tree, (1) nodes are questions, (2) sibling nodes reflect questions which aim to *sequentially* solidify the current misunderstanding, and (3) each of the parent-child edges connect nodes that guide to new understanding. Guided by the state space in Section 3.2, each level l in the tree has questions q of a similar difficulty and depth; the last level of the tree indicates that a specific state variable has been resolved. The Verifier agent dictates the movement from level to level and tree to tree.

Conditional generation of sibling questions. The Instructor *conditionally generates* sibling questions at level l if and only if the *Student incorrectly answers the Instructor question* (lines 6 and 10 in Alg. 1). As shown in the second and third question of Figure 3, these questions must lead to the same level of target understanding as the original generated question intended so therefore, the question can be rephrased or made more specific. To ensure this, we ground the question generation based on two things: (1) the previous questions from level l, and (2) the Verifier's explanation for why the Student got the question wrong.

Conditional generation of child questions. The Instructor *conditionally generates* child questions at level $l+1$ if and only if the *Student correctly answers the Instructor question* (addresses the question and has no mistakes in their answer), but still does not understand the target state variable τ_i (line 14 in Alg. 1). As shown in the fourth question of Figure 3, these questions aim to guide the Student to a more complete understanding of the target state. To ensure this, we ground the question generation on two things: (1) the previous questions from level $l-1$, and (2) the Verifier's explanation of the gaps in the Student's target state understanding.

3.4 Adaptive Conversation Restructuring

Once the Verifier agent determines that the target state/task has been resolved, we exploit the same process to update all remaining tasks $\tau \in S$, as multiple dependent bugs may have been concurrently resolved within the same tree. After at least the target state variable has been resolved (line 13 in Alg. 1), we create a new tree for any remaining tasks, as shown in the first interaction of Figure 3. This step is crucial to the multi-bug setting, as mutually independent bugs would benefit from having separate and distinct trees of questioning.

For further adaptiveness to the conversation, we additionally provide (1) an early stopping condition based on the Student's intermediate bug fixes, and (2) a maximum tree width and depth threshold, after which TreeInstruct chooses to teach the Student their remaining gap in knowledge.

- *Bug fixes:* After a task τ has been resolved, the Student is prompted to provide a list of natural language bug fixes (e.g. "Replace ‘i‘ with ‘i+1‘ on line 6.") based on their entire conversation history with the Instructor. The Verifier will determine if all of the ground-truth bug fixes have an *isomorphic* counterpart within the set of suggested Student bug fixes. Isomorphism can be defined as (1) having the same conclusion or output, (2) sharing the same underlying logical structure or pattern, and/or (3) being convertible to each other through a series of logical transformations. If all ground-truth bug fixes have been resolved, then we may stop early.

- *Teaching:* After generating a maximum number of sibling questions q or depth l, the Instructor appends the correct answer to $Q[l][0]$ and re-ask $Q[l][-1]$ to the Student. This ensures that the conversation flows in case the Student gets stuck.

4 Experiments

4.1 Experimental Setup

In order to evaluate TreeInstruct, we utilize a proxy *Student agent* based on the `Mistral-7B-Instruct` model (Jiang et al., 2023) to mimic the abilities of a student while responding to the Instructor. The prompt we use to define the Student persona is outlined in Table 10 of Appendix I. We additionally provide GPT4 API experimental set up details in Appendix G.

4.2 Datasets

We use two datasets to evaluate our method on. First, the Socratic Debugging Benchmark dataset from (Al-Hossami et al., 2023b), which consists of 149 problems, each with a problem statement, student buggy code, bug fixes and descriptions in English, and correct code. Each problem has one syntactical or conceptual bug. Second, to challenge our method, we also craft a novel dataset, **MULTI-DEBUG**, based on 50 popular programming problems[2]. For each of the 50 problems, we inject 1, 2, and 3 bug(s) that a student would make for a total of 150 different samples. We keep track of these bugs with matching bug fixes and descriptions.

Bugs are either conceptual or syntactical. Conceptual bugs usually cause runtime errors or result in incorrect output. Examples include misunderstanding the problem statement, encountering an infinite loop, or incorrectly using a library or mathematical operator (/ vs // in Python). Syntactical bugs cause compilation errors due to incorrect Python syntax (e.g., missing a colon).

4.3 Baselines

To determine the success of TreeInstruct, we also measure the performance on a few baselines. First is a baseline called **Vanilla**. Given the same input as TreeInstruct's Instructor, this method simply asks the base model to ask Socratic questions to the Student - it does not utilize the tree structure, nor does it estimate the Student's knowledge. We use both Meta-Llama-3-8B-Instruct (Touvron et al., 2023) and GPT-4 (Achiam et al., 2023) as base models for the Vanilla baseline.

Second, we use **BRIDGE** (Wang et al., 2023). Since we are adapting this for Socratic code debugging, we use the error type, the remediation strategy, and the remediation intention to guide

[2] https://github.com/Garvit244/Leetcode/

Table 1: Results on the Socratic Debugging Benchmark Dataset (Single Bug). **Bolded** and † values denote the top 2 methods respectively.

Methods	Avg. Turns	Syntactical (42 samples)				Conceptual (107 samples)			
		Success	Relevant	Indirect	Logic	Success	Relevant	Indirect	Logic
Vanilla	3.23	**80.95**	83.72†	76.19	78.70†	76.64†	87.35†	80.32†	78.79†
Bridge	6.00	78.57†	76.50	82.24†	41.72	62.14	78.12	79.86	34.38
TreeInstruct	5.41	77.27	**92.01**	**96.48**	**88.95**	**80.26**	**95.63**	**89.10**	**94.63**

Table 2: Results on the **MULTI-DEBUG** dataset. In total, 1-bug has 29 syntactical and 21 conceptual bugs, 2-bug has 50 syntactical and 50 conceptual bugs, and 3-bug has 78 syntactical and 72 conceptual bugs. **Bolded** and † values denote the top 2 methods respectively.

Bugs	Methods	Avg. Turns	Syntactical				Conceptual			
			Success	Relevant	Indirect	Logic	Success	Relevant	Indirect	Logic
1	Vanilla	2.36	71.43	92.16	55.12	84.15	**78.57**	94.58	59.17	84.17
	BRIDGE	16.60	50.00	93.93	**98.04**	24.23	68.00	97.27	**96.67**	35.38
	TreeInstruct	7.24	**76.19**	93.98†	94.08	85.28†	71.43	97.57†	93.02†	86.02†
	TreeInstruct	3.94	75.00†	**100.00**	95.59†	**96.63**	76.92†	**100.00**	88.01	**94.76**
2	Vanilla	8.32	53.26	83.45	74.41	60.82	62.50	86.96	74.13	59.90
	BRIDGE	15.28	34.88	89.47	89.33	52.40	42.71	89.67	88.06	46.64
	TreeInstruct	9.04	66.67†	93.00†	92.17†	84.59†	72.62†	94.15†	92.58†	81.46†
	TreeInstruct	6.14	**69.32**	**97.96**	**98.47**	**90.14**	**73.91**	**99.58**	**98.47**	**94.45**
3	Vanilla	17.48	44.00†	69.88	64.31	52.38	67.00	84.68	84.68	41.51
	BRIDGE	8.44	19.00	87.78	83.95	64.95	43.00	90.09	85.78	44.65
	TreeInstruct	10.46	43.00	95.68†	88.88†	80.94†	72.00†	96.76†	**97.95**	83.28†
	TreeInstruct	10.46	**73.00**	**100.00**	**99.27**	**95.57**	**92.00**	**98.40**	95.89†	**93.63**

the question generation, along with the problem-specific input given to TreeInstruct's Instructor. For both baselines, we limit the conversations to 20 turns per number of bugs.

4.4 Evaluation Metrics:

We perform qualitative and quantitative evaluation of our methods. Details for each metric are provided in Appendix A. The scores are averaged across all turns and then averaged across all problems. In the results, we scale the scores by 100.

Qualitative Metrics: We develop a binary scale to assess the Socratic quality of questions. We measure each metric manually, giving a score of 1 if the attribute is met, and 0 otherwise.

Relevance (**Relevant**) measures whether the instructor's question was pertinent to the errors in the Student's code. Indirectness (**Indirect**) measures if the instructor's question refrained from directly revealing solutions to the bugs. Finally, Logical Flow (**Logic**) checks if the instructor's question promoted a coherent conversation, facilitating the Student's problem-solving process. Each Instructor question is assigned with a binary value for each of the three attributes.

Quantitative Metrics: We apply quantitative metrics to objectively evaluate the effectiveness and efficiency of our framework. We calculate the overall Success Rate (**Success**) with the number of bug fixes generated by the Student that are isomorphic to the ground truth set of bug fixes. We also compute the average number of turns (**Avg. Turns**) required by the method to reach the goal state.

4.5 Overall Results

In Tables 1 and 2, we see that **with more structured representations of student knowledge and conversation state, TreeInstruct demonstrates significant improvements beyond the baselines.** Across all multi-bug settings, we see an overall improvement of 16.6% and 11.59% in the success rates for syntactical and conceptual bugs, respectively. We also see an improvement of 13.47% and 14.89% for syntactical and conceptual bugs, respectively, across the three conversation metrics. In the 1-bug setting, we see that the Vanilla baseline has the highest success for conceptual bugs. However, this setting simultaneously has the lowest Indirectness score, indicating that questions were very direct, and gave hints towards the bug fixes, which evidently increased the success rate. We see the same trend in the syntactical, single bug on Vanilla setting in Table 1. Overall, **TreeInstruct demonstrates strong performance despite drastically different base models, and .**

Side-by-Side evaluation: Using the conversational metrics, we performed a side-by-side evaluation that measures how often a user prefers our method TreeInstruct over the baselines. More details are in Appendix E, in which we see that **on average, TreeInstruct was preferred over BRIDGE 79.43% more, and over Vanilla 77.43% more.**

Human Student Interaction: We also conduct a separate case study where human students directly interact with TreeInstruct (details in Appendix F). We see that with varying levels of programming backgrounds, TreeInstruct is able to help all students resolve the bugs in their code.

4.5.1 Analysis

Conceptual bugs are easier to solve than syntactical. In both tables, across all settings, it can be seen that questions targeted towards conceptual bugs have higher scores than those towards syntactical bugs. Syntactical bugs might be "harder to see" for the language model as it goes against the generation process to generate syntactically incorrect code. Breaking it down, a language model trained to generate code will always add a colon at the end of for loops, if-statements, and method signatures because the model is trained to do that. Even though the buggy code might have a missing colon, the language model might ignore it.

More bug-specific state variables helps (1) generate more relevant questions and (2) maintain conversational flow. Table 2 shows that scores for Relevance and Logic on conceptual bugs decreases as the number of bugs increases (from 100% relevance and 94.76% logical flow in 1-bug to 98.4% relevance and 93.63% logical flow in 3-bug). The state space representation for the 1-bug setting is much less compact as that of the 2- and 3-bug settings. To elaborate, 1-bug state space representations suggest 3 or 4 state variables (more subtasks) to solve a bug that 2- or 3-bug state space representations take 1 or 2 state variables to solve (example provided in Appendix D). This indicates that, across states, 1-bug settings have an easier time keeping questions relevant to the bugs (as there is only one). Additionally, the conversation flows better as there are no inter-bug dependencies that the Instructor has to take into account.

Bug dependency affects success rate. Table 2 shows that the success rate (SR) for 3-bug and 1-bug are higher than that of 2-bug. 1-bug settings are overall relatively easier given that only one bug must be resolved. However, compared to the baselines, which feature low success and logic rates, Tree-Instruct demonstrates a comparatively strong performance in 3-bug settings. This is likely due to its state space and tree creation structure factoring in the inter-bug dependencies.

For example, in the Fibonacci problem in Figure 1, a student could have made the following two bugs: (1) they did not add a base case for the recursion, and (2) they did not correctly write the recursive call. Once they solve one of the bugs, they will have understood recursion better, enabling them to solve the other bug easily. TreeInstruct's prioritization of conceptual errors in the state space estimation (Section 3.2) and dependency awareness (Section 3.4) are the key to its high 3-bug performance. The same cannot necessarily be said for the 2-bug setting as it could have mutually independent bugs that require special attention to solve.

5 Conclusion

This paper proposes a novel method, **TreeInstruct**, for state space estimation and dynamic tree-based questioning for multi-turn Socratic instruction. We construct a novel multi-bug debugging dataset, **MULTI-DEBUG**, with 150 expert-annotated conceptual and syntactical problems and buggy solutions/fixes. Extensive experiments on an existing benchmark and **MULTI-DEBUG** demonstrate that TreeInstruct can be universally applied to both open and closed source-settings. We also showcase that TreeInstruct's strong Socratic questioning abilities widely outperform all baselines through both (1) rigorous quantitative and qualitative expert evaluation (preferred over 77.94% of the time), and (2) real-world interactions with students of varying coding abilities (in Appendix F).

6 Limitations & Future Work

While TreeInstruct provides an intuitive framework which demonstrates promising results for effective multi-turn Socratic instruction, it contains a few limitations that form the foundation for future, impactful research areas.

Firstly, Tables 1 and 2 shows high qualitative scores for the questions asked by TreeInstruct. While these are encouraging, the success rates still have large room for improvement– the highest success rate is 77.27%. This indicates that Socratic questions alone are not sufficient for teaching a student to debug their code. We judge the efficacy of questions locally, whereas the next step would be to judge them globally across the conversation. We leave it to future work to devise an effective global questioning scheme and evaluation metric.

Additionally, our method is dependent on the base model's reasoning capabilities, specifically for the Verifier agent. In our results, with a stronger model, we see higher scores for Logic and Success. Although our method shows comparable results between GPT-4 and Llama-3-8b, this may be a bottleneck, as stronger and bigger models require a higher deployment cost.

Next, in the few failure cases, we see some adverse effects of our method's reliance on the reasoning capabilities of the base model. First, our method can get stuck into a cyclical conversation with the Student if they are particularly weak in an area and cannot understand the target state even after multiple rounds of direct questioning and teaching. In these cases, the number of turns rises to 20-30.

Moreover, the base language models cannot fully grasp abstract concepts such as trees and linked lists. Even humans require diagrams to work out potential solutions or teach one another. With the language-reliant teaching strategies, our method might not be able to effectively teach in these domains.

These limitations give way to exciting future work. Firstly, we can make use of vision language models to provide students with multi-modal teaching strategies, instead of relying solely on language. Additionally, we can enhance the framework, so it will explore new instruction methods when the questioning becomes cyclical. This can also help make the Instructor more reliable to generate consistent output across multiple runs on the same problem. Furthermore, we can utilize a structured fine-tuning approach to help the model better leverage the Verifier feedback and tree-based question generation process and make hierarchical Socratic planning and questioning inherent to a model. Overall, TreeInstruct can also be extended to automatically generalize to different teaching domains (e.g., quantitative reasoning).

7 Ethics Statement

We are committed to the transparency and reproducibility of our research. We encourage our research community to make use of our open-source code and dataset to further improve our methodology. Our research involves the evolving intersection of large language models (LLMs) and education, where the deployment of language model instructors and their interactions with students have been relatively unexplored. The role of technology and language models is being widely discussed with respect to its impact on student dependence and lack of critical thinking. Given the rapid and wide-scale deployment of LMs to the public, we emphasize the importance of designing Socratic dialogue systems in the hopes of bettering educational support for all students and educators.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, 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.

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.

Kathleen Cotton. 1988. Classroom questioning. *School improvement research series*, 5:1–22.

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.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

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

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.

Husniah Sahamid. 2016. Developing critical thinking through socratic questioning: An action research study. *International Journal of Education and Literacy Studies*, 4(3):62–72.

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.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Rose E Wang and Dorottya Demszky. 2024. Edu-convokit: An open-source library for education conversation data. *arXiv preprint arXiv:2402.05111*.

Rose E. Wang, Qingyang Zhang, Carly Robinson, Susanna Loeb, and Dorottya Demszky. 2023. Bridging the novice-expert gap via models of decision-making: A case study on remediating math mistakes.

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.

Judith D Wilson. 1987. A socratic approach to helping novice programmers debug programs. *ACM SIGCSE Bulletin*, 19(1):179–182.

Ya-Ting C Yang, Timothy J Newby, and Robert L Bill. 2005. Using socratic questioning to promote critical thinking skills through asynchronous discussion forums in distance learning environments. *The american journal of distance education*, 19(3):163–181.

Michael V. Yudelson, Kenneth R. Koedinger, and Geoffrey J. Gordon. 2013. Individualized bayesian knowledge tracing models. In *Artificial Intelligence in Education*, pages 171–180, Berlin, Heidelberg. Springer Berlin Heidelberg.

A Evaluation Metrics

Here, we describe our qualitative and quantitative metrics in depth. The scores are averaged across all turns and then averaged across all problems of the same setting and dataset. In our results, we scale the scores by 100.

Qualitative: We develop a binary scale to assess the Socratic quality of questions. Previous work 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 (giving a score of 1 if the attribute is met, and 0 otherwise):

- **Relevance (Relevant):** The instructor's question was pertinent to the errors in the student's code.
- **Indirectness (Indirect):** The instructor's question refrained from directly revealing solutions to the bugs.
- **Logical Flow (Logic):** The instructor's question promoted a coherent conversation, facilitating the student's problem-solving process.

Quantitative: We apply quantitative metrics to objectively evaluate the effectiveness and efficiency of our framework.

- **Overall Success Rate (Success):** We check whether the final list of bug fixes generated by the Student, B_S, and the ground truth set of big fixes, B_{GT}, are isomorphic (Section 3.4). The success rate is calculated as $|B_{GT} \cup B_S|/|B_{GT}|$.
- **Average Number of Turns (Avg. Turns):** We compute the average number of turns required by the method to reach the goal state. This metric provides insight into the efficiency and depth of the interaction process.

B Human Expert Evaluators

As mentioned before, our metrics in Tables 1 and 2 were obtained using human expert evaluators: two computer science teaching assistants with at least four years of high-school, undergraduate, and graduate-level teaching experience, with proficiency in Python and located in the USA. They were given the same set of instructions and the following set of guidelines:

- Assign a score of 1 for Relevance if the question will eventually lead the Student to understand their bug(s).
- Assign a score of 0 for Indirect if a question explicitly or implicitly states a solution.
- Assign a score of 0 for Logic if the current question does not naturally flow from the Student's previous answer.

Below are some special cases/considerations the evaluators were also given:

- If the Verifier is wrong and asks the same question despite the Student getting the question correct, give a score of 0 for Relevance.
- If a question seems out of order, give a score of 0 for Logic.
- If a question deep into the conversation is vague, gives a score of 0 for Relevance and Logic.
- If the answer is provided in a hint after 2 rounds of similar questions, and the Student still does not understand, do not penalize the Instructor for Indirect.
- For determining Success, do not penalize the Student if the bug fix is in natural language rather than code.

C Ablation Studies

Table 3 compares the results of the 3-bug setting with two ablation settings:

- No Teaching: We remove the teaching functionality that kicks in when the Student has answered three consecutive questions incorrectly. Conversation is still guided by the state space representation, tree-based questioning, its updates, and the bug fixes proposed by the Student.
- No State: We remove the state space representation. We guide conversation based on the conversation history, the previous questions asked, the Verifier feedback on the Student's answer, and the bug fixes proposed by the Student.

Overall, we can see that when comparing the Llama-based ablations with the GPT4-based baselines, our ablation performance is still competitive, especially with respect to the relevance and indirectness of the questions. However, the significant drops in performance indicate the importance of our different modules, especially TI No State. We provide a detailed analysis of our ablation results below:

From TI to TI No Teaching, we see the Success rates and Logic scores to drop by 17.20% and 11.32%, respectively, on average. Teaching is a crucial part of our method because if the Student truly does not know a concept, then asking them more questions will not help them learn. While the Instructor's job in TreeInstruct is to ask questions, at a certain point, it should teach the Student to clear their confusion. Hence, when we remove teaching, we see even fewer bugs solved and more repeated questions being asked, leading to low logical flow. However, we still see

Next, **for TI No State, we see significant drops of 18.25% in Success rates, 46.63% in Relevance, and 51.39% in Logic scores, on average**. The state space representation guides the question generation, ensures the questions are on topic to the bugs, and keeps track of the Student's misunderstandings. Without this grounding, we noticed that the conversations (1) deviate from the

Table 3: Results on the 3-bug setting of the **MULTI-DEBUG** dataset compared with three ablation settings. TI indicates TreeInstruct, our method.

		Syntactical				Conceptual			
Methods	Avg. Turns	Success	Relevant	Indirect	Logic	Success	Relevant	Indirect	Logic
Vanilla	17.48	44.00†	69.88	64.31	52.38	67.00	84.68	84.68	41.51
BRIDGE	8.44	19.00	87.78	83.95	64.95	43.00	90.09	85.78	44.65
TreeInstruct	10.46	43.00	95.68†	88.88	80.94†	72.00†	96.76†	**97.95**	83.28†
TreeInstruct	10.46	**73.00**	**100.00**	**99.27**	**95.57**	**92.00**	**98.40**	95.89†	**93.63**
TI No Teaching	9.69	30.61	90.75	97.61†	72.84	50.00	94.62	95.17	68.78
TI No State	16.34	25.51	51.61	97.21	41.09	53.00	47.57	94.70	20.36

real bugs– exploring areas such as time complexity optimization, which might not be the focus of the problem, (2) contain countless repeated questions that the Student already answered, and (3) jump from topic to topic abruptly in consecutive turns. These results show how impactful the state space representation is.

D Comparing State Space Representations in Multi-Bug Settings

Here, we compare the state space representations of the 1-bug, 2-bug, and 3-bug settings for the two sum problem. In the two sum problem, given is an array of integers and a target value. The goal is to return the indices of two numbers that add up to the target value. Below is the correct code.

```
1. def twoSum(self, nums, target):
2.   d = {}
3.   for i in range(len(nums)):
4.     difference = target-nums[i]
5.     if difference in d:
6.       return [d[difference], i]
7.     d[nums[i]] = i
8.   return d
```

In the 1-bug setting, the Student mistakenly writes `nums[i]-target` instead of `target-nums[i]` on line 4. In the 2-bug setting, along with the previous bug, the Student also initializes `d` as a list (`d=[]`) instead of a dictionary on line 2. Finally, in the 3-bug setting, the Student forgets to add a colon at the end of the if-statement on line 5.

Tables 4, 5, and 6 outline the state space representations for the 1-bug, 2-bug, and 3-bug settings. As shown, 1-bug uses 3 states (states 1, 2, and 3) to solve the same but that 3-bug uses 1 state (state 1) to solve. This means the 1-bug state representation is much less compact than that for 3-bug.

1. Understand the problem statement and the requirement to find two numbers that add up to a specific target.
2. Understand the logic behind calculating the difference as target - nums[i].
3. Correctly implement the difference calculation in the code.

Table 4: State space representation for 1-bug on the two-sum problem.

1. Understand how to correctly calculate the difference between the target and the current number in the array.
2. Understand the difference between lists and dictionaries in Python.
3. Correctly initialize a dictionary in Python.
4. Understand how to use a dictionary to store and retrieve values in Python.

Table 5: State space representation for 2-bug on the two-sum problem.

E Side by Side Evaluation

As mentioned in the main text, we perform a side-by-side evaluation to measure the percentage of times a user prefers our method TreeInstruct over the baselines baselines. Preference was measured as the average of all conversation metrics across syntactical and conceptual bugs. Based on the metrics, we assign each method a ranking (1, 2, or 3). Table 7 shows that TreeInstruct was preferred 68-94.6% of the time over the baselines. On average, TreeInstruct was preferred over BRIDGE 79.43% of the time, and over Vanilla 77.43% of the time.

1. Understand how to correctly calculate the difference as ‘target-nums[i]‘.
2. Understand how to initialize a dictionary using ‘‘ instead of ‘[]‘.
3. Understand how to use a dictionary to store and retrieve values.
4. Understand the correct syntax for an if-condition, including the necessary colon at the end.

Table 6: State space representation for 3-bug on the two-sum problem.

Table 7: Results on the side-by-side evaluation. **Bolded** and † values denote the top 2 comparisons respectively. Note: S-bug refers to the Socratic Debugging Benchmark. We abbreviate TreeInstruct as TI.

Comparison	3-bug	2-bug	1-bug	S-bug
TI vs BRIDGE	71.43†	68.00†	**83.67**	**94.63**
TI vs Vanilla	**100.00**	**90.00**	69.39†	50.33†
BRIDGE vs Vanilla	57.14	62.00	40.82	24.83

Interpretation. When we say TreeInstruct was preferred 79.43% more over BRIDGE, this means that across all 50 3-bug problems and ranking configurations, TI was given a higher ranking than BRIDGE (TI is ranked #1 while BRIDGE is ranked #2, TI is ranked #1 while BRIDGE is ranked #3, TI is ranked #2 while BRIDGE is ranked #3) 79.44% of the time. Each of the 50 problems can have multiple preferences (TI over BRIDGE, TI over Vanilla, Bridge over Vanilla, etc.) which is why they will not necessarily add up to 100.

F Interactive Evaluation with Human Students

For our main evaluation, we used `Mistral-7B-Instruct` to represent a Student. We noticed that Mistral is an overconfident model that (1) suggests incorrect bug fixes in between the conversations and (2) jumps to fix bugs that do not exist in the code. Therefore, we worked with human students to test our method on the following two settings: Socratic Debugging on TreeInstruct and 3-bug on TreeInstruct. We gathered 5 human volunteers of varying levels of programming backgrounds and knowledge (ensuring to anonymize their identities):

- **Level 1:** Student knows how TreeInstruct works; they act as an adversary to intentionally provide bad inputs that will try to make the method fail.
- **Level 2:** Student is new to TreeInstruct; they are a basic programmer who has been learning to code in Python for a few months.
- **Level 3:** Student is new to TreeInstruct; they are a non-computer science major who does not use Python often, but knows the basic high level concepts of data structures and syntax.
- **Level 4:** Student is new to TreeInstruct; they have been using Python for 2 years and is in their final year of undergraduate education in computer science.
- **Level 5:** Student knows how TreeInstruct works; they act as an ally to intentionally provide good inputs so the method can resolve the bugs in as little turns as possible.

When conducting the study, we adhered to the following experimental process:

1. We presented the student with the problem statement and gave them as much time as they needed to fully understand it.
2. The students were given two minutes to review the buggy code. We noted down how many bugs each of the students were able to identify before their conversation.
3. The students conversed with TreeInstruct until they were able to identify all of the bugs present in the code.

We provide the results of this interactive study in Table 8. We used the same three single and 3-bug questions for all students, leading to 30 human student interactions in total. We also conducted a post-interaction interview with each of the students and provide an overview of their feedback below:

Socratic questioning helped students learn programming concepts. The Level 3 student stated that, *"If there was no conversation, I would be put off from attempting to fix and just try a bunch of different things based on the errors."* Overall, students of Levels 2-4 (students with no knowledge of the system) were not able to identify all of the bugs before their interactions, but ended up solving them independently under the Socratic guidance of TreeInstruct.

Table 8: Results of human student evaluation across s(ingle)-bug (Socratic Debugging benchmark) and 3-bug (**MULTI-DEBUG** dataset) settings, broken down by the student level.

			Syntactical				Conceptual			
Bugs	Methods	Avg. Turns	Success	Relevant	Indirect	Logic	Success	Relevant	Indirect	Logic
S-bug	Level 1	6.0	**100.00**	66.67	66.67	**100.00**	**100.00**	91.67†	**100.00**	50.79
	Level 2	12.0	**100.00**	66.67	83.33†	75.00†	50.00†	**100.00**	**100.00**	50.00
	Level 3	8.0	0.00†	87.50†	**100.00**	50.00	**100.00**	67.50	90.00†	42.50
	Level 4	1.0	**100.00**	**100.00**	0.00	**100.00**	**100.00**	57.14	**100.00**	64.29†
	Level 5	1.0	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	75.00	**75.00**
3-bug	Level 1	19.0	83.33†	75.93	97.92†	74.77	**100.00**	**100.00**	88.89†	79.49
	Level 2	11.7	83.33†	**100.00**	**100.00**	78.57	**100.00**	**100.00**	86.67	82.50
	Level 3	6.67	**100.00**	**100.00**	**100.00**	85.71†	**100.00**	**100.00**	**100.00**	**100.00**
	Level 4	4.7	**100.00**	93.33†	**100.00**	76.67	**100.00**	**100.00**	83.33	88.89†
	Level 5	3.0	**100.00**	**100.00**	83.33	**100.00**	**100.00**	83.33†	**100.00**	83.33

Underlying model had a significant impact on user experience. Students had a significantly better experience with TreeInstruct compared to TreeInstruct. Specifically, the quality of the Verifier determined whether or not the questions posed by the Instructor would be overly repetitive or not.

F.1 Analysis

Table 8 contains the results. We see that from Level 1 to Level 5, the conversation have fewer turns, especially in the 3-bug setting. Additionally, we see that syntactical bugs are harder to solve for weaker students (on average, a success rate of 86.67%), which is intuitive as these students do not have a strong foundation in Python syntax. On the other hand, conceptual bugs are easier to solve (on average, a 95% success rate). Overall, the results show that our method can adapt to various levels of students effectively.

G Model Inference Experimental Setup

G.1 GPT-4 API

For GPT-4, we made use of OpenAI's GPT-4 API. Overall, we use temperature sensitivity $t = 0$ for all generation tasks, except for $t = 0.1$ for state space estimation and $t = 0.3$ for instructor question generation.

Using $30 / 1M input tokens and $60 / 1M output tokens, we break down the cost for each method. TreeInstruct uses an average of 35,000 input tokens and 4,000 output tokens, which adds up to **$1.29 per conversation**. BRIDGE uses an average of 18,000 input and 5,500 output tokens, which adds up to **$0.87 per conversation**. Vanilla uses an average of 31,000 output and 2,200 output tokens, which adds up to **$1.06 per conversation**.

G.2 Mistral and Llama

We run the Mistral-7B-Instruct-0.2 and Llama-3 models locally on 2 NVIDIA-RTX A6000 GPUs. For one pass on a dataset (i.e., 150 problems/-conversations), TreeInstruct takes approximately 4 hours. We use the same temperature settings as the GPT-4 API.

H License

All of the datasets used in this work, including our own, is under the Apache 2.0 License. Our use of existing artifact(s) is consistent with their intended use, specifically for the Socratic Debugging benchmark and in general, programming practice and feedback for the problems used in the MULTI-DEBUG dataset.

I 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. We provide the specific prompts starting from the next page.

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 9: Instructor agent persona prompt

You are a Student writing code to solve the above problem statement (after tag 'problem'), and you have written the below buggy code (after tag 'buggy_code'). You are seeking help from your Instructor help solve your 'buggy_code'. Your role is to answer the questions that the Instructor asks you as if you were an introductory programmer with a beginner's level of coding knowledge.

Table 10: 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'). Assume the Student is a introductory programmer with a beginner's level of coding knowledge.

Table 11: Verifier agent persona prompt

Given the student's buggy code (after tag 'buggy_code'), bug description (after tag 'bug_description'), bug fixes (after tag 'bug_fixes'), and the correct code (after tag 'correct_code') for solving the problem statement (after tag 'problem'), we define the state representation of a set of Instructor-Student interactions as a series of necessary tasks which lead the Student from their 'buggy_code', with bugs described in 'bug_description', to understanding and correcting their conceptual and syntactical mistakes to reach 'correct_code' with the 'bug_fixes'. 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. A NECESSARY task directly addresses at least one of the 'bug_description's and thus, is NOT ALREADY ADDRESSED in 'buggy_code'. In other words, if a task is not successfully completed, the Student will never be able to correct their 'buggy_code' to 'correct_code'. If the student's 'buggy_code' shows that they have already understood and implemented a specific task, DO NOT INCLUDE that task as a state attribute since it is REDUNDANT. The list should be ordered, with earlier attributes/tasks given priority over later ones (e.g., conceptual understanding tasks are a pre-requisite 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 12: Internal Verifier prompt to estimate the state space representation; corresponds to the GenerateState() method in line 1 of Alg 1.

The Student has written code (after tag ’student_code’) to solve the problem (after tag ’problem’) and is answering a question (after tag ’Student’) from the Instructor (after tag ’Instructor’) based on their understanding of the ’problem’ and their ’student_code’. IF the Student suggests a solution to a bug they identify, also consider the following:

Ensure that the Student’s suggestion is isomorphic to any one of the bug fixes mentioned in the provided ’bug_fixes’; if not, then ’answer_has_no_mistakes’ should be "False". A Student’s suggestion is isomorphic to a bug fix if they (1) have the same conclusion or output, (2) share the same underlying logical structure or pattern, and (3) are convertible to each other through a series of logical transformations.

Answer the following questions and within your reasoning, think about how you would answer the "instructor_question" yourself and include this in your "explanation".: answer_addresses_question: <Does the Student’s response (after tag ’Student’) directly answer the Instructor’s question (after tag ’Instructor’)? Output "True or "False"> answer_has_no_mistakes: <Is the Student’s response (after tag ’Student’) to the Instructor’s question (after tag ’Instructor’) logical (no logical errors or mistakes)? Output "True or "False">

Instructor: {Instructor question}
Student: {Student response}
bug_fixes: {bug fixes}
student_code: {student code}

Table 13: Internal Verifier prompt to assess the accuracy of the Student response with respect to the Instructor’s question; corresponds to the VerifyResponse() method in line 6 of Alg 1.

A Student has sufficient understanding of a certain topic (specified at tag "target_understanding") when the responses that they provide to the Instructor (specified in the "conversation_history") would REQUIRE them to comprehend "target_understanding". This can either be demonstrated (1) explicitly, where the Student directly mentions "target_understanding", OR (2) implicitly, where their reasoning is isomorphic to completing the task in "target_understanding". A Student’s reasoning is isomorphic to the "target_understanding" if they (1) have the same conclusion or output, (2) share the same underlying logical structure or pattern, and (3) are convertible to each other through a series of logical transformations.

Based on the Student’s response (after tag ’student_response’) to the Instructor’s question (after tag ’instructor_question’) and the conversation history (after tag ’conversation_history’), do you believe that the Student needed to sufficiently comprehend the "target_understanding" in order to provide their responses (after tag ’Student’ in ’conversation_history’) to the Instructor’s questions (after tag ’Instructor’ in ’conversation_history’) throughout the conversation history? Include specific quotes from the "conversation_history" in your "explanation". Within your reasoning, think about how you would answer the "instructor_question" yourself and include this in your "explanation".

Instructor: {Instructor question}
Student: {Student response}
target_understanding: {target understanding}

Table 14: Internal Verifier prompt to update the state space with respect to a single-turn Instructor-Student interaction; corresponds to the UpdateUnderstanding() method in line 12 of Alg 1.

Are any bug fixes mentioned in the conversation that you have had with the Instructor (under tag "conversation_history")? If no, return "None". If yes, then follow the format below: First, based on your current understanding of the problem (tag "problem") and your conversation with the Instructor, summarize (after tag "bug_summarization") the bugs in the code explicitly mentioned within the "conversation_history" that you believe will revise your buggy code (after tag "buggy_code") to a correct implementation of the "problem" statement. Then, based on this summary, output a list of the explicitly mentioned bug fixes (from "bug_fix_1" to "bug_fix_n", where n is the number of bug fixes to make), each described briefly. An example format/wording of a brief bug fix would be: "Replace ‘i‘ with ‘i+1‘ on line 6." conversation history: {convo history}

Table 15: Instructor to Student prompt that asks the Student to generate a list of bug fixes; corresponds to the GetStudentBugFixes() method in line 17 of Alg 1.

For the problem description given above (after tag ’problem’), you are given two sets of bug fixes (under tags ’suggested_bug_fixes’ and ’correct_bug_fixes’). For each bug fix in ’correct_bug_fixes’, is there at least one bug fix in ’suggested_bug_fixes’ that is isomorphic? Two bug fixes are isomorphic if they (1) have the same conclusion or output, (2) share the same underlying logical structure or pattern, and (3) are convertible to each other through a series of logical transformations. Output "True" or "False" as your answer with an explanation. suggested bug fixes: {student_bf} correct bug fixes: {correct_bf}

Table 16: Internal Verifier prompt check if the Student has suggested all the correct bug fixes that are present in the ground truth set of bug fixes, corresponds to isResolved() in line 4 of Alg. 1.

Based on the student’s current level of understanding, as demonstrated through their conversation history (tag "conversation_history"), what is 1 follow-up question with the same level of depth and difficulty RELATIVE to the ’previous_questions’ that you could ask based on the Student’s explanation that would help them reach the "target_understanding"? Make sure that the question addresses the reasons why the Student got the previous question(s) wrong, as detailed in tag "misunderstanding", such that the Student is more likely to resolve these misunderstandings. You must generate a question such that any correct answer to your question should automatically reflect the "target_understanding" and resolve the "misunderstanding". target_understanding: {target} conversation_history: {conversation history} previous_questions: {previous questions} previous_misunderstanding: {explanations} These questions should help the Student arrive at the answer themselves; do NOT give any direct hints towards the solution (under tag "bug_fixes" and tag "bug_description"). bug_fixes: {bug fixes} bug_descriptions: {bug descriptions}

Table 17: Internal Instructor prompt to generate a sibling question; corresponds to the GenerateSiblingQuestion() method in line 10 of Alg 1.

Based on the student's current level of understanding, as demonstrated through their conversation history (tag "conversation_history"), what is 1 follow-up question with increasing depth and difficulty RELATIVE to the 'previous_questions' that you could ask based on the Student's explanation that would help them reach the "target_understanding"? Make sure that the question addresses the reasons why the Student has not reached the "target_understanding", as detailed in tag "misunderstanding", such that the Student is more likely to resolve these "misunderstanding"s by answering your question. target_understanding: {target} conversation_history: {conversation history} previous_questions: {previous questions} previous_misunderstanding: {explanations} These questions should help the Student arrive at the answer themselves; do NOT give any direct hints towards the solution (under tag "bug_fixes" and tag "bug_description"). bug_fixes: {bug fixes} bug_descriptions: {bug descriptions}

Table 18: Internal Instructor prompt to generate a child question; corresponds to the GenerateChildQuestion() method in line 14 of Alg 1.

Based on the buggy code and the target understanding state (under tag "target_understanding"), what is one question (k=1) that you could ask that would help the Student reach the "target_understanding"? These questions should help the Student arrive at the answer themselves; do NOT give any direct hints towards the solution (after tag 'bug_fixes'). These questions should help the Student arrive at the answer themselves; do NOT give any direct hints towards the solution (under tag "bug_fixes" and tag "bug_description"). target_understanding: {target} bug_fixes: {bug fixes} bug_descriptions: {bug descriptions}

Table 19: Internal Instructor prompt to generate the initial question; corresponds to the GenerateQuestion() method in line 3 of Alg 1.