Washington University in St. Louis

Washington University Open Scholarship

McKelvey School of Engineering Theses & Dissertations

McKelvey School of Engineering

Spring 5-15-2023

An Assistive Interface For Displaying Novice's Code History

Ruiwei Xiao

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Educational Technology Commons, and the Graphics and Human Computer Interfaces Commons

Recommended Citation

Xiao, Ruiwei, "An Assistive Interface For Displaying Novice's Code History" (2023). *McKelvey School of Engineering Theses & Dissertations*. 851.
https://openscholarship.wustl.edu/eng_etds/851

This Thesis is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.

WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Computer Science & Engineering

Thesis Examination Committee:
Caitlin Kelleher, Chair
Jonathan Shidal
Doug Shook

An Assistive Interface For Displaying Novice's Code History

by

Ruiwei Xiao

A thesis presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Master of Science

May 2023
St. Louis, Missouri

© 2023, Ruiwei Xiao

Table of Contents

List of Figures…… iv
List of Tables……v
Acknowledgments……vi
Abstract…… viii
Chapter 1: Introduction……1
Chapter 2: Background……3
Chapter 3: Related Work……5
3.1 TA Training Programs……5
3.2 Automated Code Correction Tools……5
3.3 Computer-Supported Tutoring Tools……6
Chapter 4: Overview of the System……8
4.1 Components……8
4.1.1 Question Information……8
4.1.2 Submission History Snapshots……9
4.1.3 One Submission……10
4.1.4 Concept Table……11
4.2 Usage Scenario……12
Chapter 5: Methodology……14
5.1 Participants……14
5.2 Study Material……14
5.2.1 Interfaces……14
5.2.2 Problem Data……15
5.3 Experiment Design……15
5.3.1 Experiment Process……15
5.3.2 Survey questions……16
5.4 Data Collection……17
Chapter 6: Data Analysis and Results……18
6.1 Justification of Methods Choices……18

6.2 Quantitative Analysis……………………………………………..……………...19
6.2.1 Performance in Evaluating Students' Concepts Use………………………………… 19
6.2.2 Performance in Evaluating Students' Debugging Skills ……………………………21
6.2.3 Participants' Time Spent ……………………………………………………………..23
6.3 Thematic Coding Analysis ……………………………………………………………24
6.3.1 How Code Develops…………………………………………………………………24
6.3.2 Repeated Submissions……………………………………………………………….26
6.3.3 Feedback on the Assistive Interface …………………………………………………26
6.4 Results……………………………………………………………………………….28
Chapter 7: Conclusion…...…………………………………………………………………28
7.1 Discussion…………………………………………………………………………...29
7.1.1 Feedback…..…………………………………………………………………………29
7.2 Future Work…………………………………………………………………………..29
7.2.1 Expanded Study...……………………………………………………………………29
7.2.2 New Approaches to Generate and Represent Code Correction...…………………29
References………………………………………………………………………………..30

List of Figures

Figure 4.1: The Assistive Interface 8
Figure 4.2: The View of Selecting Student's Edits Only........ 10
Figure 4.3: The View of Selecting Correction's Edits Only........ 10
Figure 4.4: The View of Selecting Both Checkboxes........ 10
Figure 5.1: The Baseline Interface 14
Figure 6.1: Code history#6, submission#6........ 25
Figure 6.2: Code history#6, submission#7........ 25
Figure 6.3: A Hidden Bug From Code History #4........ 25

List of Tables

Table 4.1: An Example Concept Table.....12
Table 5.1: Question-Answering Order in Different Groups.....16
Table 6.1: ***Section 1*** ANCOVA Result.....20
Table 6.2: ***Section 1*** repeated measure ANOVA Result.....20
Table 6.3: ***Section 2*** ANCOVA Result.....23
Table 6.4: ***Section 2*** repeated measure ANOVA Result.....22
Table 6.5: ANCOVA on Time Spent.....23
Table 6.6: Repeated Measure ANOVA on Time Spent.....23

Acknowledgments

I would like to take this opportunity to express my gratitude to my advisor, Professor Caitlin Kelleher for her advising on this research. As a researcher, she is knowledgeable and careful, and always uses her knowledge and experience to help me in every step of my research. As an advisor, she is patient and supportive. Her help and guidance make a huge impact on my early research career development.

I also want to say thank you to Yana Malysheva, a Ph.D. student advised by Professor Caitlin Kelleher. She attended every weekly meeting to discuss research ideas, and her previous and current work greatly helped the development of this research.

Finally, I want to thank my family for their support in my education, my life, and my future development.

Ruiwei Xiao

Washington University in St. Louis

May 2023

Dedicated to my parent

ABSTRACT OF THE THESIS

An Assistive Interface For Displaying Novice's Code History

by

Ruiwei Xiao

Master of Science in Computer Science

Washington University in St. Louis, 2023

Professor Caitlin Kelleher, Chair

As Teaching Assistant (TA) programs grow in number and size in introductory CS courses, TAs play a significant role in novice programmers' experience and contribute to their success. However, many TAs are also relative beginners themselves and thus have limited experience in programming and teaching. Thus the effectiveness and consistency of their guidance can vary significantly. To improve interaction quality and assist TAs in providing better support, we examine the difficulties encountered by inexperienced TAs in previous literature and then identify the potential for the high cognitive load as an unaddressed difficulty that may prevent new TAs from initiating effective TA-student interactions. This work aims to build a scaffolding tool to assist with assistant teaching tasks and help them initiate better TA-student interactions. A user study has also been conducted for evaluation purposes. Based on our user study, it appears that our assistive interface achieved significant effects on assisting TAs in evaluating students' debugging skills. When performing this task, 80% of TAs achieved better performance after using the assistive interface, and the accuracy of questions answered using the assistive interface is 22% higher than using the baseline interface. Additionally, the assistive interface is more likely to identify visually minor, student-specific problems to facilitate conversations between TAs and students.

Chapter 1: Introduction

In introductory Computer Science courses, students who encounter difficulties with programming problems often turn to Teaching Assistants (TAs) as their initial source of assistance. As enrollments surge, TAs play a larger role in students' experience and contribute a lot to students' success [1, 2, 3]. Regardless of the overall success of using TAs in introductory CS courses [4, 5], some major complaints about TAs' performance are related to time efficiency [6], lack of common understanding [4], and less interactive teaching methods [5]. These complaints are associated with the high cognitive load that may be challenging to new TAs - they need to complete multiple programming-related (e.g. identifying one student's intention, debugging, etc.) and teaching-related (e.g. correcting concepts the student misunderstood, leading students to find the bugs, etc.) tasks simultaneously in a short period of time, while many of these TAs are also relative beginners themselves and inexperienced in both programming and teaching. Therefore, the effectiveness and consistency of their feedback and guidance can vary significantly.

To address this issue and assist TAs in providing better support, we build an assistive code history visualization interface. This interface aims to reduce the cognitive load for TAs on multiple tasks so that TAs are able to initiate higher-quality conversations in a shorter time. A user test has also been conducted for the evaluation purpose of this tool. The user study results suggest that our assistive interface can help TAs perform better in evaluating students' debugging skills. When performing this task, 80% of TAs achieved better performance after using the assistive interface, and the accuracy of questions answered using the assistive interface is 22% higher than the accuracy of using the baseline interface. Also, the assistive interface is more

likely to reveal visually minor, student-specific problems to facilitate conversations between TAs and students.

Chapter 2: Background

Cognitive load theory is an instructional theory based on our knowledge of human cognition. This theory suggests that our working memory is only able to hold a small amount of information at any given time [4]. In order to help learners process more information under a limited capacity, researchers investigated three different types of cognitive loads and considered whether there are approaches to reduce each of them.

The three types of cognitive loads are intrinsic load, extraneous load, and germane load. Intrinsic load is related to the complexity of the task and the learner's characteristics, and it can be optimized by selecting learning tasks that match the learner's prior knowledge. Extraneous load is related to instructional material efficiency, and appropriate instructional material design helps to avoid overload in this aspect. Germane load helps link newly-obtained information with those already stored in long-term memory, and effective instructional design will potentially be beneficial for the germane load [11]. In the general CS Education domain, when teaching introductory CS concepts to students, researchers designed instructional material to minimize the extraneous and intrinsic loads in order to increase the amount of memory available for the germane load [5]. However, little research related to evaluating and reducing the cognitive load during the assistant teaching process has been conducted.

In our context, we try to reduce the cognitive load by designing an interface that closely matches with assistant teaching tasks and TA's abilities and experience level. Aligning with the typical workflow that TAs use to examine a code history, different representations of students' code history are also designed in our interface. More specifically, when helping a student debug a coding problem, the TA usually starts by reading the problem description, and then the TA looks

at the student's code, trying to understand the student's logic and find out what changes the student made, what are the bugs, and how to connect them to the student's conceptual understanding and debugging skills. After acquiring the needed information, the TA points to a piece of code and initiates the conversation with the student to lead the student to solve all problems together. In this process, considering many of the TAs are beginners themselves, features on our interface provide a suggested correct solution for them and also highlight the bugs and students' edits to make up for TAs' potential incompetence of coding abilities in order to reduce TAs' intrinsic load. By matching every visualization with each task TAs are required to perform, we avoid redundant and irrelevant information for the purpose of reducing TAs' extraneous load.

Chapter 3: Related Works

3.1 TA Training Programs

Many institutions advocate best practices during TA-student interactions by running TA training programs. Some study uses a "Collaborative Learning Framework" (CLF) poster to demonstrate several best practices during peer tutoring [6], while others design curriculums and activities to equip TAs with interpersonal and pedagogical skills [7, 8]. Regardless of the improved experiences and outcomes by following best practices, these training programs are not designed to reduce the cognitive load of the debugging process - peer tutors still need to organize and process the information on their own.

3.2 Automated Code Correction Tools

Abstract Syntax Tree (AST) has been commonly used to power automated code correction tools for generating feedback [9] or next-step hints [2, 6, 26]. Some of these works use a data-driven approach and compare the student's erroneous solution with a set of correct solutions, while some other works such as ITAP [10] utilize AST for path construction to handle states that have not occurred in the data before.

Regardless of the development of these automation tools, the limitation of these tools in understanding students' intent affects the empirical result of helping students. Several works raised concerns with the validation of generated feedback [12, 13, 14]. One previous study found that even the best-performance hint-generation tool performs at least 20% and 10% less than a human tutor in the generation of fully valid and partially valid hints respectively [14]. The more occurrences of such poor-quality hints, the less help students tend to seek [15]. Also, while most of these AST applications emphasize comparison between correct solutions and the

student's code, few of them compare one student's previous (incorrect) submission to the current submission in this student's code history. The code history will offer more context related to one student's intent, concept mastery, and debugging skills so that help-provider are able to give better help and instructions.

In this work, the backend is also powered by automation code correlation algorithms built on AST [16]. However, instead of giving students automatic feedback or hints directly, our class size and TA resources allow us to take a step back and involve humans-in-the-loop to observe the patterns of AST generated fixed and pursue the positive effects brought by one-on-one human tutoring [17].

3.3 Computer-Supported Tutoring Tools

Previous researchers have designed and implemented computer-supported tools to facilitate TA-student interactions. Some of these tools allow course staff to group and search for a piece of code among all submissions from the whole class. Online learning environments such as MOOCs are a common use case for these implementations [16, 17, 29]. An example of these tools is OverCode [29], which uses both static and dynamic analysis to cluster similar solutions. Based on the solution clusters, instructors are also allowed to apply different criteria to further filter and cluster solutions. In this way, instructors can initiate intervention or provide more targeted feedback on common misunderstandings specific to the cluster to which the student belongs.

Another type of TA-assistive tool provides information on one snapshot in depth. For example, AIORT [19] is an interface that utilizes automated feedback techniques to assist TAs in determining if one code submission is on the right track or not. In addition to a yes or no answer to whether this piece of code is moving in the right direction, the interface also provides term

lookups and test cases relevant to this problem. In addition, code visualization tools [20, 21, 22] can also be helpful when TAs walk the student through the debugging process.

Among all these previous works, few of them focus on assisting TAs to understand how an individual student's code develops, and then initiate a conversation with this one student. Pensieve [23] organizes snapshots of student code as they progress through an assignment. Some of these works use IDE events to represent one student's code development procedures in solving one coding problem. TaskTracker-tool [24] tracks code snapshots of scores gained and IDE actions (e.g. edit, run, or paste code) to help TAs or instructors gain deep insight into the process of student programming.

Similar to all these computer-supported tutoring tools, our implementation also attempts to provide support by reducing extraneous cognitive load in various ways (e.g. chunking information, reducing redundancy, etc.). However, our work focuses more on helping TAs gauge the information which is commonly challenging to gain to prepare for a common ground for TA-student conversation.

Chapter 4: Overview of the System

There are four components in this interface. Each component is designed to reveal how this student’s code develops from different perspectives. The main strategies used in the interface to reduce cognitive load are 1) the dual coding approach that represents information in different colors; and 2) the chunking information approach that chunk bug-concept relationships into a table.

Figure 4.1 The Assistive Interface: (A1) problem description; (A2) test cases for this problem; (B) Submission History Snapshots; (C) the selected submission and its closest correction; (D) a concept table that maps required concepts with bugs and student’s edits

4.1 Components

4.1.1 Question Information

The question information section is on the left side of the interface. The name, description, and test cases of this question will be displayed in Figure 4.1(A1), and Figure 4.1(A2). Users can click each test case to view the student’s output and correct output, and the color of these test case buttons indicates whether the code passed (green) the test case or not (red). In response to

the change in code submission selection (see section 4.1.2), the color and output of test cases would also change.

The best use scenario of this feature could be 1) helping TA locate the submission that passed most of the test cases. This version is most likely when the student is close to the correct answer; 2) helping TA find the test case(s) that exist in most of the submissions, these test cases may indicate some problems that the student was unable to solve during the whole process.

4.1.2 Submission History Snapshots

This section uses a sequence of code snapshots each with a green bar on the left side to represent a student's code submission history. The height of these green bars symbolizes how similar the current code is to the closest correct version. There are two checkboxes on top of the snapshot sequence - Student's Edits and Correction's Edits. By selecting "student's edits", users can see the blue highlight that indicates which parts the student changed in the next submission (Figure 4.2); similarly, by selecting "correction's edit", users can see the red highlight that indicates what needs to be changed in order to get the correct answer (Figure 4.3). By selecting both checkboxes, users can see the overlapped area, and that means the student edit is the same as the suggested correction, and this can be interpreted as the student located the buggy place successfully (Figure 4.4). When the user clicks on one of the snapshots, more details about this submission will be displayed underneath.

Our design purpose of this section is to shorten TAs' time in finding out whether the student is moving toward the right track - TAs can observe the height of the green bars, or the area of overlapped highlights to obtain this information.

Figure 4.2 The View of Selecting *Student's Edits* Only

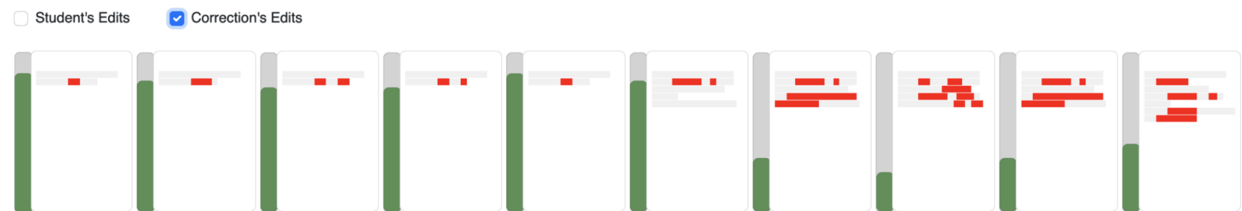

Figure 4.3 The View of Selecting *Correction's Edits* Only

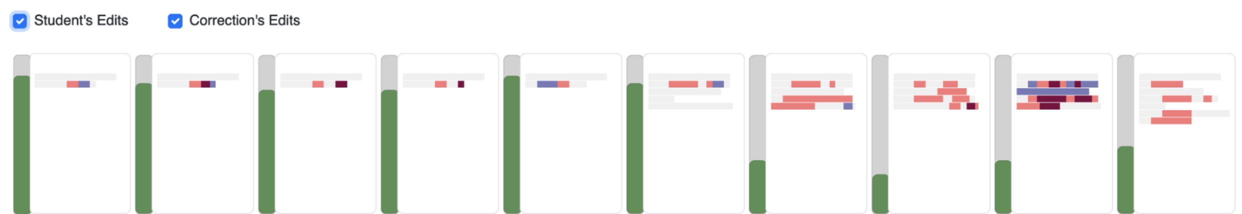

Figure 4.4 The View of Selecting Both Checkboxes

4.1.3 One Submission

After clicking one snapshot, the submission details will be shown in this section. The left column shows the code the student submitted, and the right column shows the closest correct version for the student's submission. The differences between these two submissions are also highlighted.

This section is implemented using the diff2html library [23], which provides a clean code comparison visualization in the frontend. Providing the correct solution on the side has been proven to be helpful in a previous study [4] and in this study - our user test result shows that all participants, regardless of their years of assistant teaching experience, agreed that the correct solutions supported their thinking process. The intention of highlighting every difference between the correct version and the student's version is to aid the process of finding multiple

instances of the same bugs and finding visually minor or positionally connected bugs, which are all identified as challenges for TAs in their debugging process [4].

4.1.4 Concept Table

Lastly, A concept table would show at the bottom. The intention of this design is to reduce TAs' cognitive load in understanding the student intent, debugging skills, and understanding of required concepts. This table will show users the bugs and student edits associated with each concept, as well as how many edits made by this student are correct.

Table 4.1 is an example of a concept table for one submission history. Red, blue, and green highlights imply information related to the current submission. C#1, C#2, and C#3 are highlighted in red, which means there are bugs associated with these three concepts in the current submission. Similarly, from the blue highlights we can tell that student has edited code related to C#1, C#2, and C#4 in the current code. The green highlight implies the edit student made to C#1 successfully solved the bug related to the same concept. The logic to decide whether an edit is correct is: 1) there is a bug and a student's edit on the same concept; 2) the same bug does not exist in the next submission. Only in this case, the green highlight would appear.

For C#1, the student demonstrated good debugging skills by finding and resolving all related bugs; for C#2, the student showed some level of ability to diagnose the problem associated with it but still be struggling to correct it. The student failed on debugging tasks related to C#3 and C#4 by changing the wrong places. In terms of the numbers in the table, the student may already understand C#1 because it only appeared in two submissions and has been resolved. C#2 might be the most confusing concept for this student since it existed in 10 submissions and remained unsolved. The student was likely not to realize the problem related to C#3 and thought the code related to C#4 was buggy. But soon the student realized it was not, since the student changed the

code related to C#4 back in 2 submissions. Therefore, the instructor may need to talk about C#2 and C#3 with the student to help on solving this problem.

Table 4.1 An Example Concept Table

Concept	Bugs	Student's Edits	Correct Edits?	Explanation
C#1	2	1	1	There is at least one bug related to C#1, and the student successfully located and resolved all related bugs.
C#2	10	9	0	There is at least one bug related to C#2, and the student successfully located at least one of them but failed to resolve all related bugs.
C#3	2	0	0	There is at least one bug related to C#3, but the student failed to locate any of them in this submission.
C#4	1	2	1	There is no bug related to C#4, but the student changed the code related to this concept.

4.2 Usage Scenario

In current standard practices, the student will bring one version of the code to the TA (sometimes with the test cases). And there are three main differences made in this work that better assist TA to find more valuable information.

By only looking at one version, it is hard to understand how this student's logic develops. With the code history, TAs can identify salient patterns or changes made by the student, and hence ask students questions such as "Why do you change A to B" to better communicate with the student. In addition, our interface provides TAs with the closest correction along with highlighted differences between the correction and the student's code. This feature makes sure every bug is highlighted so that TAs are less likely to miss certain bugs. Lastly, this work is also able to

connect the bugs and edits that appear in one code history with the required concepts. Therefore, TAs are able to more quickly associate the bugs with certain concepts that are misunderstood by the student and provide feedback on time.

Chapter 5: Methodology

5.1 Participants

We recruited study participants through 1) announcements made by the CS introductory course instructor; 2) a mailing list for current and past undergraduate computer science TAs at Washington University in St. Louis. Participants received a $20 gift card in recognition of their participation in a 1.5-hour-long study session. Among these 10 participants, 6 of them have one semester of TA experience, 4 have more than 4 semesters' experience, and the rest two participants have been TA for 2 semesters and 3 semesters respectively.

5.2 Study Material

5.2.1 Interfaces

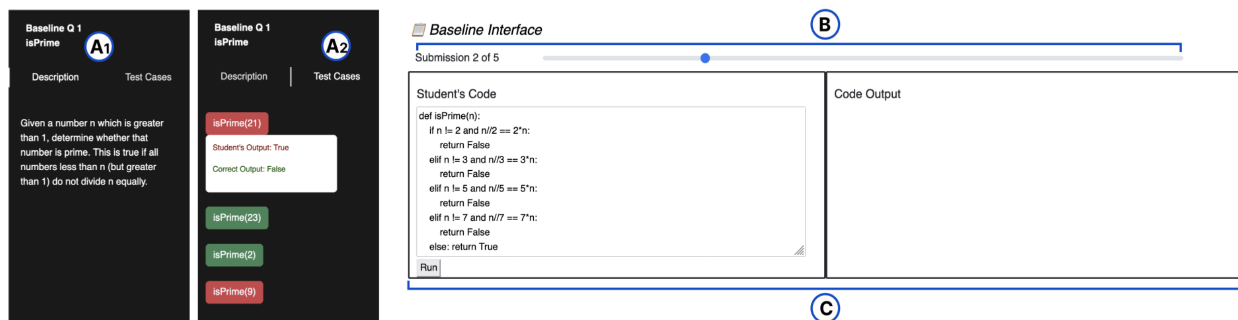

Figure 5.1 The Baseline Interface: (A1) problem description; (A2) test cases for this problem; (B) a slide bar to select submission; (C) the code of the selected submission.

Two interfaces are used in the experiment. In addition to the assistive interface introduced in the last section, another interface being used is the baseline interface (Figure 5.1). The baseline interface only shows the basic information about this problem and the code submission history: the left column shows the problem description and test cases, and on the right side, users are able to select and view the code for a certain submission by changing the scroll bar. Since the

information on the baseline interface is the same as TAs would see in regular student-helping settings, this interface enables us to conduct experiments in the control condition.

5.2.2 Problem Data

To present the code histories on the interfaces, we obtained the problem descriptions, required concepts, test cases, correct output, and the student code submission history from the CSEDM2019-Data-Challenge dataset [26]. We conducted data transforming and manual checks to select code histories with a length greater than 5 and unusual patterns in their code development. The length and unusual patterns increase the complexity of the code histories and thus can be challenging for TAs to interpret students' intentions. And then, a data pipeline has been built to generate 1) a correction for each erroneous code submission [4]; 2) a difference between each submission and its closest correction [4]; 3) the difference between each submission and its next submission; 4) output of each test case for each submission, and 5) mapping 2) and 3) to the required concepts.

5.3 Experiment design

5.3.1 Experiment process

The purpose of the experiment is to examine whether the assistive interface is able to reduce TAs' cognitive load and scaffold TA-student interactions. We conducted a 1.5-hour long one-on-one session with each participant. During the session, the participant is required to use specific interfaces to look at 6 code histories written by novice programmers and then answer survey questions. To mitigate the difference caused by individual ability and interface-using order, we first assigned each participant to one of the four groups. Group settings details are shown in Table 5.1. The notation "Baseline (h1, h2, h3)" means the participant will use the

baseline interface to look at code history and answer questions on code history#1, #2, and #3 subsequently.

Participants first watched a 1-2 minute long demo video to learn about their tasks, survey questions, and how to use the first interface before they started on the first three code histories. After finishing tasks on the first three histories, participants would watch another 1-2 minute demo video on how to use the second interface and then finish tasks on code history 4-6. For each code history, the participant 10-15 minutes to observe the code history and answer survey questions.

Table 5.1. Question-Answering Order in Different Groups

Group	Number of Participants	First Three Code History	Last Three Code History
group#1	3	Baseline (h1, h2, h3)	Assistive (h4, h5, h6)
group#2	2	Assistive (h4, h5, h6)	Baseline (h1, h2, h3)
group#3	3	Assistive (h1, h2, h3)	Baseline (h4, h5, h6)
group#4	2	Baseline (h4, h5, h6)	Assistive (h1, h2, h3)

5.3.2 Survey questions

The survey starts by asking for participant id, group id, and years of TA experience. And then for each of the six code histories, participants need to answer three sets of questions based on their observation of the code history: ***section 1***: How well does the code history owner understand and use each required concept (rating from 1-7); ***section 2***: How well the code history owner does in debugging (rating from 1-7); and ***section 3***: How to initiate the conversation with the code

history owner? All the survey questions aim for evaluating how well the interface scaffolds the participant to understand the development of the code history and initiate the conversation with the code owner.

5.4 Data collection

We collected two types of data from participants: survey answers and screen recordings using both interfaces during the whole experiment. From the data, we are able to extract participants' assessments of each code history owner's understanding of concepts, debugging ability, participants' way of initiating the conversation and question-answering time for these six problems. We also built the ground truth for survey questions in ***section 1*** and ***section 2*** by answering these questions by ourselves and the GPT-4 model [28] and validating the GPT-4 generated justifications of each score manually as a quality control procedure.

Chapter 6: Data Analysis and Results

In this section, we first conduct a statistical analysis using ANCOVA [30] and repeated measure ANOVA [30] on the distances between participants' answers in the survey ***section 1*** and ***section 2*** to the ground truth of each section, as well as the time spent on each code history. Next, we use thematic coding [27] to identify the themes in teaching assistant responses to the questions in ***section 3***. Lastly, we use the analysis result in the first two steps to examine whether the assistive interface facilitates better TA-student interactions by answering three research questions: ***RQ1:*** Does the Assistive Interface help TAs obtain information faster? ***RQ2:*** Does the Assistive Interface help TAs obtain information more accurately? ***RQ3:*** Does the Assistive Interface help TAs understand a student's intent better?

6.1 Justification of Methods Choices

In this study, ANCOVA is used for examining the influence of an independent variable on a dependent variable while removing the effect of the covariate factor. In our experiment settings, we eliminated the variance introduced by different code histories by selecting code histories with similar complexities. Therefore, participants' task performances mainly depend on the interface use and their own experience level. As we want to examine the effect of the interface used on TAs' performance regardless of participants' experience level, we assigned the semesters of TA experience, a self-reported value, as the covariant to represent participants' previous experience, interface used as the independent variable, and task performances (e.g. distance to ground truth, time spent) as the dependent variable. ANCOVA results are used to indicate among 60 tasks completed by 10 participants, whether tasks performed using the assistive interface achieve better performance regardless of participants' experience level.

Repeated measures analysis, by definition, deals with response outcomes measured on the same experimental unit using different settings of within-subject variables [citation here]. The within-subject variable represents the conditions each individual is exposed to. It allows researchers to control participants' differences since each participant serves as their own control. In this experiment, the interface use is the within-subject variable, and we repeated the observing code history task with different interfaces used on each participant. As each participant performed three tasks on each interface, the mean value of task performances on one interface is aggregated to represent this participant's performance on one interface. The repeated measures ANOVA is mainly used to examine whether the assistive interface can improve an individual's task performance.

Thematic coding analysis is a qualitative analysis approach used for identifying common themes and arranging these themes in multiple ways to reveal patterns in the data. Reasons for applying this approach in our experiment are: 1) the human thought process is complex and thus hard to be measured by a quantitative matrix; 2) unexpected patterns and scenarios can be identified to facilitate a more well-rounded evaluation and improvement of the system design.

6.2 Quantitative Analysis

6.2.1 Performance in Evaluating Students' Concepts Use

To compare participants' performance in evaluating students' concepts use using baseline and assistive interfaces, we use the distance from the ground truth to participants' answers in survey ***section 1*** to represent one participant's performance. The lower the distance is, the better this participant performed in this task.

We first performed ANCOVA on ***section 1***'s data by setting the interface used as the independent variable, experience as the co-variant, and distance to ground truth per code history as the

dependent variable. The result is shown in Table 6.1. Although the average distance per code history using the assistive interface (5.37) is lower than which of the baseline interface (6.87), the uncorrected p-values (p-unc) indicate neither the interface used nor previous experiences have a statistically significant effect on the evaluation of students' concept use.

Table 6.1. ***Section 1*** **ANCOVA Result**

Source	SS	DF	F	p-unc	np2
interface	33.750000	1	3.237398	0.077269	0.053744
experience	6.206061	1	0.595303	0.443563	0.010336
Residual	594.227273	57	NaN	NaN	NaN

Next, we examined whether different interface use makes a difference in each individual's performance using repeated measure ANOVA (Table 6.2). While the average distance after using the assistive interface did decrease for 70% of all individuals, the result is not statistically significant.

Table 6.2. ***Section 1*** **repeated measure ANOVA Result**

	F Value	Num DF	Den DF	Pr > F
interface	3.7461	1.0000	9.0000	0.0849

Combining the ANCOVA and repeated measure ANOVA results, we found that ***section 1*** questions answered using the assistive interface are generally closer to the ground truth, and for each participant, using the assistive interface also makes the answers more accurate. Such observation may imply the positive effects of the assistive interface. However, since no statistical significance was observed, the influence caused by the interface used can be minute.

In light of the results of participants' evaluation of students' concept use, it can be explained that the assistive interface did not perform as well as we expected due to the fact that the code history is not closely connected to concepts, but it is also surprising that the accuracy is not correlated with the prior experience of TAs. According to participants' feedback, most of them perceived this experiment and their previous assistant teaching jobs as a debugging process rather than a combination of understanding, diagnosing, and communication processes. Since most of their previous experience does not include the concept evaluation process, the prior experience represented by the length of assistant teaching semesters cannot accurately represent their capability of evaluating students' concept use. In addition, the variation in participants' evaluating standards could be another reason that contributed to the insignificant impacts from both interfaces used and prior experience.

6.2.2 Performance in Evaluating Students' Debugging Skills

Similar to section 6.2.1, we use the distance from the ground truth to participants' answers in survey ***section 2*** to represent one participant's performance score in evaluating code owners' debugging skills. Table 6.3 and Table 6.4 use the same approaches as the previous section to group all data points.

To perform ANCOVA on ***section 2***'s data, we set the interface used as the independent variable, experience as the co-variant, and distance to ground truth per code history as the dependent variable. The result is shown in Table 6.3. The average distance for each code history using the assistive interface and the baseline interface is 4.6 and 5.9 respectively, and the p-unc value of the interface indicates that the assistive interface made a statistically significant difference in participants' performance in evaluating code owners' debugging skills regardless of their previous TA experience.

Table 6.3. ***Section 2*** **ANCOVA Result**

Source	SS	DF	F	p-unc	np2
interface	25.350000	1	4.919416	0.030557	0.079449
experience	4.176136	1	0.810420	0.371784	0.014019
Residual	293.723864	57	NaN	NaN	NaN

We also examined the interfaces' effects on each individual using repeated measure ANOVA (Table 6.4). Based on the fact that 80% of participants achieved lower average distance after using the assistive interface, together with the p-value (0.0423), we can conclude that the assistive interface helps each individual perform better in evaluating students' debugging skills.

Table 6.4. ***Section 2*** **repeated measure ANOVA Result**

	F Value	Num DF	Den DF	Pr > F
interface	5.5896	1.0000	9.0000	0.0423

Comments from participants may explain how the assistive interface improved participants' performance in evaluating debugging skills: "*I can understand the student's intention to change from one version to the next version by using both interfaces, but only by using the assistive interface can I always tell whether this change is expected (in the right direction)*". Moreover, all participants favored the corrections (Figure 4.1(C)) on the side with the highlight feature on the assistive interface, which helped them identify all bugs within a shorter time. In summary, the assistive interface reduced TAs' workload by helping TAs locate bugs and tell whether students' changes are on the right track or not, thus the accuracy of diagnosing code owners' debugging skills was improved both within each individual and among all participants. Participants' prior experience, again, turns out to be less influential on the result. Similar to section 6.2.1, the

variation in participants' evaluating standards may once again affect the measurement of prior experience's effects.

6.2.3 Participants' Time Spent

We also conducted quantitative analysis in comparing two types of users' time spent (Table 6.5 and Table 6.6). Overall speaking, assistive interface users tend to finish their tasks quicker, with an average time of 11.55 minutes and 11.93 minutes per question, but there is no significant difference between the time use of different interface users or difference within each individual.

Table 6.5 ANCOVA on Time Spent

Source	SS	DF	F	p-unc	np2
interface	0.820560	1	0.066653	0.797203	0.001168
experience	83.502235	1	6.782794	0.011717	0.106342
Residual	701.720811	57	NaN	NaN	NaN

Table 6.6 Repeated Measure ANOVA on Time Spent

	F Value	Num DF	Den DF	Pr > F
interface	0.0384	1.0000	9.0000	0.8490

As the result indicates, the assistive interface users would not perform quicker than baseline interface users. One possible explanation could be the experimental settings is not identical to the office hour settings. Participants are required to finish each question in roughly 10 minutes, whereas during office hours, the session ends once the student thinks problem-solved. And participants also need to write down answers instead of talking to students, which may make a difference in time spent. Another cause may be the greater amount of information on the assistive interface compared to the baseline. Participants may stay longer on the assistive interface and try to obtain more data. In addition, the increment of features and information on

the assistive interface can result in a higher learning curve and thus slow down participants' speed.

6.3 Thematic Coding Analysis

We also performed thematic coding analysis on participants' responses in ***section 3*** as our qualitative data analysis strategy. In this section, TAs depicted how to initiate the conversation with the code history owner and the rationale behind it. We mainly use this information to assess participants' understanding of students' intent and their feedback on the assistive interface.

6.3.1 How Code Develops

Two types of codings demonstrated participants' understanding of code development. The first type of phrase is used to summarize what is the student doing. For example, in code history#1, the code owner kept adding ***if*** cases that specifically target passing each test case rather than writing code to handle more general cases. One participant summarized the development of this code history as "to (barely) fit the test cases", which showcases a good understanding of code development. However, most of the code histories are hard to be clearly described in a short sentence. Therefore, such coding only appeared once from an assistive interface user.

A more feasible way to examine participants' understandings is to check the coverage of potential problems in participants' answers. We assume that the more problems a participant mentioned in the answer, the better understanding this participant has. Potential problems include bugs that cause test case failures and bad practices that do not affect the test case results. One example of the "bad practices" is an unnecessary change from submission#6 (Figure 6.1) to #7 (Figure 6.2) in code history#6. Among all collected responses, one participant answered *"I would also ask them about their thoughts on the previous and after iteration to check whether using both ifs is necessary"* to point out the student's bad practice in the if-else structure.

```
1 def howManyEggCartons(eggs):
2     if (eggs // 12) % 12 == 0:
3         return eggs // 12
4     else:
5         return (eggs // 12) + 1
```

Figure 6.1 Code History#6, Submission#6

```
1 def howManyEggCartons(eggs):
2     if (eggs // 12) % 12 == 0:
3         return eggs // 12
4     if (eggs // 12) % 12 != 0:
5         return (eggs // 12) + 1
```

Figure 6.2 Code History#6, Submission#7

Another example is in code history#4, only one participant pointed out the bug in the if condition and designed a test case to walk the student through it. This bug is visually minor and has been deleted (rather than solved) in the later submissions due to approach changes, but it clearly shows this student's limited understanding of the modulus symbol and calculation. Figure 6.3 shows the submission that contains the bug, and the test case the participant suggested is "test case 21 is incorrect for all versions of the code except the final version. The idea of using n != x and n//x == x*(n//x) is incorrect."

```
1 def isPrime(n):
2     if n != 2 and n//2 == 2*(n//2):
3         return False
4     elif n != 3 and n//3 == 3*(n//3):
5         return False
6     elif n != 5 and n//5 == 5*(n//5):
7         return False
8     elif n != 7 and n//7 == 7*(n//7):
9         return False
```

Figure 6.3 A Hidden Bug From Code History #4

While users of both interfaces did a great job in finding bugs that directly affect test case results, participants using the assistive interface outperformed their counterparts using the baseline interface in locating bad practices that did not directly cause the failure in test cases, but reveal students' incompetence in concept use or debugging. This conclusion is drawn from participants'

answers to “What version of the code would you ask the student to explain and why”. 33.33% (10 out of 30) answers from participants using the assistive interface and 13.33% (4 out of 30) answers from participants using the baseline interface choose to talk about the bad practices. This result implies a higher probability for assistive interface users to notice more problems in students’ code development, and hence understand students’ intent better and initiate more informative conversations.

6.3.2 Repeated Submissions

Repeated (but not adjacent) submissions often imply the student’s struggling - the student moves towards a direction for a while and then gave up and reverts to a previous version. In this experiment, 3 participants pointed at submission#3 and submission#5 of code history#2 and asked “*Are these two submissions identical?*” This also happened when participants were working on code history#1 and #6. Since the same submissions in code history locate closer, most participants easily identified them as well as the students’ attempts to solve the off-by-1 issue. However, none of the participants, either using the assistive interface or the baseline interface found the same pattern in code history#3, where the first submission is identical to the 10th (out of 12) submission, and the student was struggling with locating the bug in 9 iterations between these two submissions. Participant’s failure to find identical submissions located far away suggests the need to present the connections between repeated submissions.

6.3.3 Feedback on the Assistive Interface

Feedback on each component of the assistive interface has also been extracted. All participants mentioned that 1) corrections (Figure 4.1(C)) on the side and highlights to show the difference between the student’s code and the correct version are helpful; 2) test cases and corresponding outputs (Figure 4.1(A2)) are helpful, regardless of each participant’s reliance (e.g time spent on

test cases, number of clicks) on test cases varies; 3) comparing to one version of code that students usually bring to the office hour, code history enables TAs to learn about the student's intent and debugging skills.

While two participants think the green bar (within Figure 4.1(B)) in the assistive interface is one of the most frequently used features to observe whether the student is on the right track, two participants mentioned the green bar is confusing, because they intuitively believe it represents how many test cases this submission passed even they know what the green bar stands for.

Novice TAs tend to rely on the concept table (Figure 4.1(D)) more. However, one participant (id: 6) is confused by the concept table. *"I thought the number means the total number of bugs that occurred (but it is actually the number of submissions that contains this kind of bug)."*, and two participants said they did not use the concept table too often due to the learning curve. These comments suggest the need to revise the concept table to provide more clear and more straightforward information.

Some feedback is not expected at the point when the experiment was designed. One participant (with more than 4 semesters of TA experience) only observed the code itself rather than other features. *"I spent most of my time looking at the student's code section and rarely looked elsewhere. Personally, I prefer baseline cuz it's more clear."* Another participant shared their own vision of auto-correction and TA works - this participant suggested using gpt4 to generate corrections and it actually went very well in comparing code, detecting all bugs, and other tasks. *"Maybe (in the future) we don't need TAs anymore."* On the other hand, we need to be wary of the overuse of technology by both students and help providers and design instructions, assessments, and feedback accordingly.

6.4 Results

With the analysis results in sections 6.2 and 6.3, we are able to answer the three research questions.

RQ1: Does the Assistive Interface help TAs obtain information faster?

Although many participants said having the correct solution on the assistive interface allows them to locate bugs faster, the results in section 6.2.3 indicate that using the assistive interface has little improvement on TAs' information processing speed.

RQ2: Does the Assistive Interface help TAs obtain information more accurately?

By using the assistive interface, participants are able to obtain information on students' debugging skills more accurately. Assistive interface users also tend to have better performance in assessing code owners' concept use, yet no statistical significance has been shown in this task.

RQ3: Does the Assistive Interface help TAs understand a student's intent better?

From the thematic code analysis in section 6.3.1, assistive interface users are more likely to initiate a conversation on multiple latent problems in addition to the code lines that caused test case failures directly. The correction comparison and highlighting feature in Figure 4.1 (C) may contribute to the process of locating these problems and making participants think about the logic behind them. However, assistive interface users did not outperform baseline interface users in identifying students' struggles using repeated submission.

Chapter 7: Conclusion

7.1 Discussion

In this work, we build an assistive interface to reduce TAs' cognitive load and prepare them for TA-student interactions with a common ground with the student. The user study results on 10 participants suggest this tool’s positive effects on scaffolding the evaluation of code history owners’ concept use and debugging skills, as well as revealing visually minor, student-specific problems to facilitate conversations between TAs and students. However, this interface failed to accelerate TA’s speed, and the feedback from participants also suggests some flaws in the interface design.

7.1.1 Feedback

Feedback from the user study suggests there is too much information on the interface. Based on the strategies to reduce extraneous cognitive load by removing redundancy, the amount of information in the snapshot view and the concept table should be reduced. For example, we can only keep the green bar in the submission history snapshot section to represent each submission, and we can reorganize the information in the concept table and keep the minimum numbers and colors to map concepts with the student’s code.

7.2 Future Work

7.2.1 Expanded Study

As is shown in Section 6, due to the small number of participants (n=10), it is hard to obtain statistically significant results. Recruiting more participants would contribute to more generalized results.

7.2.2 New Approaches to Generate and Represent Code Correction

One participant proposed to use GPT-4 [26] to generate automated code correction and answers to the survey questions, and the results were promising for novice programmers' code. It would be interesting to evaluate and compare the performances of GPT-4 and traditional models (e.g. AST) on tasks such as automated code correction and debugging. If GPT-4's high performance is consistent, we can think of how to integrate that into tutoring tools and how to organize and represent the generated information to best assist learning and teaching.

References

[1] B. Wilson and S. Shrock, “Contributing to success in an introductory computer science course: A study of twelve factors,” presented at the ACM Sigcse Bulletin, Mar. 2001, pp. 184–188. doi: 10.1145/366413.364581.

[2] E. Roberts and E. Lazowska, “Tsunami or Sea Change? Responding to the Explosion of Student Interest in Computer Science,” Proc. 2014 NCWIT Summit Women IT, 2014.

[3] J. Forbes, D. J. Malan, H. Pon-Barry, S. Reges, and M. Sahami, “Scaling Introductory Courses Using Undergraduate Teaching Assistants,” in Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle Washington USA: ACM, Mar. 2017, pp. 657–658. doi: 10.1145/3017680.3017694.

[4] J. Sweller, P. Ayres, and S. Kalyuga, Cognitive Load Theory. New York, NY: Springer, 2011. doi: 10.1007/978-1-4419-8126-4.

[5] B. B. Morrison, B. Dorn, and M. Guzdial, “Measuring cognitive load in introductory CS: adaptation of an instrument,” in Proceedings of the tenth annual conference on International computing education research, in ICER ’14. New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 131–138. doi: 10.1145/2632320.2632348.

[6] B. A. Kos, “The collaborative learning framework: Scaffolding for untrained peer-to-peer collaboration,” 2017.

[7] F. Muzny and M. D. Shah, “Teaching Assistant Training: An Adjustable Curriculum for Computing Disciplines,” in Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, in SIGCSE 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 430–436. doi: 10.1145/3545945.3569866.

[8] G. Rößling and J. Gölz, “Preparing first-time CS student teaching assistants,” in Proceedings of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education, in ITiCSE 2018. New York, NY, USA: Association for Computing Machinery, Jul. 2018, p. 376. doi: 10.1145/3197091.3205829.

[9] R. Singh, S. Gulwani, and A. Solar-Lezama, “Automated feedback generation for introductory programming assignments,” in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, in PLDI ’13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 15–26. doi: 10.1145/2491956.2462195.

[10] K. Rivers and K. R. Koedinger, “Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor,” Int. J. Artif. Intell. Educ., vol. 27, no. 1, pp. 37–64, Mar. 2017, doi: 10.1007/s40593-015-0070-z.

[11] J. Leppink, F. Paas, C. P. M. Van der Vleuten, T. Van Gog, and J. J. G. Van Merriënboer, “Development of an instrument for measuring different types of cognitive load,” Behav. Res. Methods, vol. 45, no. 4, pp. 1058–1072, Dec. 2013, doi: 10.3758/s13428-013-0334-1.

[12] C. Watson, F. W. B. Li, and J. L. Godwin, “BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair,” in Advances in Web-Based Learning - ICWL 2012, E. Popescu, Q. Li, R. Klamma, H. Leung, and M. Specht, Eds., in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, pp. 228–239. doi: 10.1007/978-3-642-33642-3_25.

[13] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart, “The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces.” arXiv, Jun. 30, 2018. doi: 10.48550/arXiv.1708.06564.

[14] T. W. Price et al., “A Comparison of the Quality of Data-Driven Programming Hint Generation Algorithms,” Int. J. Artif. Intell. Educ., vol. 29, no. 3, pp. 368–395, Aug. 2019, doi: 10.1007/s40593-019-00177-z.

[15] T. W. Price, R. Zhi, and T. Barnes, “Hint Generation Under Uncertainty: The Effect of Hint Quality on Help-Seeking Behavior,” in Artificial Intelligence in Education, E. André, R. Baker, X. Hu, Ma. M. T. Rodrigo, and B. du Boulay, Eds., in Lecture Notes in Computer Science, vol. 10331. Cham: Springer International Publishing, 2017, pp. 311–322. doi: 10.1007/978-3-319-61425-0_26.

[16] Y. Malysheva and C. Kelleher, “An Algorithm for Generating Explainable Corrections to Student Code,” in Proceedings of the 22nd Koli Calling International Conference on Computing Education Research, 2022, pp. 1–11.

[17] M. Minnes, C. Alvarado, and L. Porter, “Lightweight Techniques to Support Students in Large Classes,” in Proceedings of the 49th ACM Technical Symposium on Computer Science Education, Baltimore Maryland USA: ACM, Feb. 2018, pp. 122–127. doi: 10.1145/3159450.3159601.

[18] A. Nguyen, C. Piech, J. Huang, and L. Guibas, “Codewebs: Scalable homework search for massive open online programming courses,” presented at the WWW 2014 - Proceedings of the 23rd International Conference on World Wide Web, Apr. 2014, pp. 491–502. doi: 10.1145/2566486.2568023.

[19] M. Q. Feldman, Y. Wang, W. E. Byrd, F. Guimbretière, and E. Andersen, “Towards answering ‘Am I on the right track?’ automatically using program synthesis,” in Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E, in SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 13–24. doi: 10.1145/3358711.3361626.

[20] H. Kang and P. J. Guo, “Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations,” in Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, in UIST ’17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 737–745. doi: 10.1145/3126594.3126632.

[21] P. J. Guo, “Online python tutor: embeddable web-based program visualization for cs education,” in Proceeding of the 44th ACM technical symposium on Computer science education, in SIGCSE ’13. New York, NY, USA: Association for Computing Machinery, Mar. 2013, pp. 579–584. doi: 10.1145/2445196.2445368.

[22] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari, “Visualizing programs with Jeliot 3,” Proc. Work. Conf. Adv. Vis. Interfaces, pp. 373–376, May 2004, doi: 10.1145/989863.989928.

[23] L. Yan, A. Hu, and C. Piech, “Pensieve: Feedback on Coding Process for Novices,” in Proceedings of the 50th ACM Technical Symposium on Computer Science Education, in SIGCSE ’19. New York, NY, USA: Association for Computing Machinery, Feb. 2019, pp. 253–259. doi: 10.1145/3287324.3287483.

[24] E. Lyulina, A. Birillo, V. Kovalenko, and T. Bryksin, “TaskTracker-tool: A Toolkit for Tracking of Code Snapshots and Activity Data During Solution of Programming Tasks,” in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, in SIGCSE ’21. New York, NY, USA: Association for Computing Machinery, Mar. 2021, pp. 495–501. doi: 10.1145/3408877.3432534.

[25] R. Fernandes (rtfpessoa), “diff2html,” diff2html. https://diff2html.xyz (accessed Apr. 14, 2023).

[26] T. Price, “The CSEDM 2019 Data Challenge.” Sep. 10, 2022. Accessed: Apr. 16, 2023. [Online]. Available: https://github.com/thomaswp/CSEDM2019-Data-Challenge

[27] R. E. Boyatzis, Transforming Qualitative Information: Thematic Analysis and Code Development. SAGE, 1998.

[28] “GPT-4.” https://openai.com/research/gpt-4 (accessed Apr. 17, 2023).

[29] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, “OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale,” ACM Trans. Comput.-Hum. Interact., vol. 22, no. 2, p. 7:1-7:35, Mar. 2015, doi: 10.1145/2699751.

[30] T. D. Wickens and G. Keppel, Design and analysis: A researcher’s handbook. Pearson Prentice-Hall Upper Saddle River, NJ, 2004.