Using Adaptive Parsons Problems to Scaffold Write-Code Problems

Xinying Hou
xyhou@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Barbara Jane Ericson
barbarer@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

Xu Wang
xwanghci@umich.edu
University of Michigan
Ann Arbor, Michigan, United States

ABSTRACT

In this paper, we explore using Parsons problems to scaffold novice programmers who are struggling while solving write-code problems. Parsons problems, in which students put mixed-up code blocks in order, can be created quickly and already serve thousands of students while other types of programming support methods are expensive to develop or do not scale. We conducted two studies in which novices were given equivalent Parsons problems as optional scaffolding while solving write-code problems. We investigated when, why, and how students used the Parsons problems as well as their perceptions of the benefits and challenges. A think-aloud observational study with 11 undergraduate students showed that students utilized the Parsons problem before writing a solution to get ideas about where to start; during writing a solution when they were stuck; and after writing a solution to debug errors and look for better strategies. Semi-structured interviews with the same 11 undergraduate students provided evidence that using Parsons problems to scaffold write-code problems helped students to reduce the difficulty, reduce the problem completion time, learn problem-solving strategies, and refine their programming knowledge. However, some students found them less useful if the Parsons solution did not match their approach or if they did not understand the solution. We then conducted a between-subjects classroom study with 81 undergraduate students to investigate the effects on learning. We found that students who received Parsons problems as scaffolding during write-code problems spent significantly less time solving those problems. However, there was no significant learning gain in either condition from pretest to posttest. We also discuss the design implications of our findings.

CCS CONCEPTS

• **Applied computing** → **Education**; **Interactive learning environments**; **Computer-assisted instruction**.

KEYWORDS

Parsons Problems, Introductory Programming, Scaffolding, Help-Seeking, Code Writing

ICER '22, August 7–11, 2022, Lugano and Virtual Event, Switzerland

ACM ISBN 978-1-4503-9194-8/22/08…$15.00
https://doi.org/10.1145/3501385.3543977

ACM Reference Format:
Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2022. Using Adaptive Parsons Problems to Scaffold Write-Code Problems. In *Proceedings of the 2022 ACM Conference on International Computing Education Research V.1 (ICER 2022), August 7–10, 2022, Lugano and Virtual Event, Switzerland.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3501385.3543977

1 INTRODUCTION

One of the primary goals for introductory programming classes is to develop skill in writing code [42, 80, 81]. However, a growing number of studies suggest that writing code can be prohibitively challenging [63] for novice programmers for various reasons, such as the gap between theoretical concepts and practice [65] and high cognitive load [73]. Students spend many frustrating hours trying to figure out why their program fails to compile or does not produce the expected output [6]. Such frustration is one reason for the high drop-out and failure rates in introductory programming courses [77].

To tackle this issue, prior work has investigated different scaffolding approaches to assist students. The term "scaffolding" is used to describe the assistance given to students so that they can complete a task that is otherwise out of reach [16]. When learning a subject matter, one-on-one tutoring has been found to be more effective than typical classroom instruction with just one instructor [9]. However, it would be too costly to provide one-on-one tutoring in undergraduate programming courses due to their huge size [27]. Computer-based scaffolding techniques could be scaled to reach thousands of students [56]. One notable approach is timely and personalized programming hints from intelligent tutoring systems (ITS) [57]. Nevertheless, ITS have not been widely adopted in higher education due to the time and cost of developing such systems [3, 37]. To reduce the cost, ITS researchers have explored using data-driven technologies to generate hints automatically [59, 62]. Despite previous research demonstrating the possibility of employing these automated hints to aid students when solving write-code problems [32], there are still several barriers to scaling the use of those techniques. First, the quality of automatically generated hints varies significantly, and low-quality hints might prevent students from seeking help [57]. Also, existing systems provide help based on the student's current code, which may not be sufficient for novices who have difficulty determining how to begin and therefore require more comprehensive help [46, 63]. In addition, hints generated by one system are hard to implement or transfer to other systems [32]. As a result, there is still a need for a more scalable, low-cost, easy-to-implement, and high-quality solution that will allow students of various levels to succeed on write-code problems.

Parsons problems, an increasingly popular programming exercise that requires students to place mixed-up code blocks in the

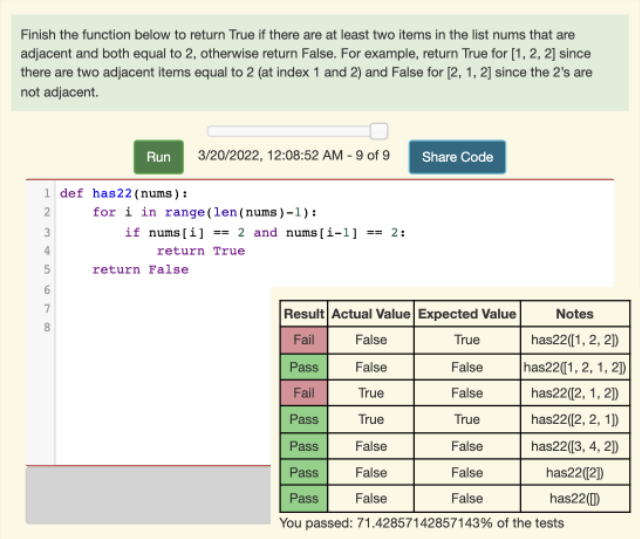

Figure 1: Screenshot of a write-code problem with unit test results

correct order [53], could potentially scaffold novices who are struggling while writing code from scratch. Although Parsons problems have been used as practice and summative assessment questions [19], to our knowledge, no prior studies have investigated the use of Parsons problems to scaffold write-code problems. Nevertheless, learners could benefit from Parsons problems in a variety of ways. For instance, Parsons problems can decrease the required cognitive load by constraining the problem space and providing prewritten code pieces yet still exercise students' higher-level thinking skills [30]. Also, compared to worked examples that provide passive guidelines and do not promote active learning [15], Parsons problems allow more interactivity and most students find them engaging [19, 53]. These results indicate that Parsons problems have the potential to provide effective scaffolding for write-code problems.

This research examined the impact of using Parsons problems to scaffold students while they solve write-code problems. We conducted our studies in *Runestone*, a free e-book platform. In *Runestone*, students can write and execute code and receive immediate feedback from unit test results and error messages (Figure 1). Our studies added an equivalent optional Parsons problem to each write-code problem. Students could display and solve the equivalent Parsons problem in a preview window (Figure 2), but were still required to solve the write-code problem. Students could not copy and paste the Parsons problem solution to the write-code problem, they had to at least retype the solution. Our research questions were as follows.

- RQ1: When, why, and how do students use Parsons problems to scaffold write-code problems?
- RQ2: What are the perceived benefits and challenges to using Parsons problems to scaffold write-code problems?
- RQ3: What is the effect of using Parsons problems as scaffolding to write-code problems on learning?

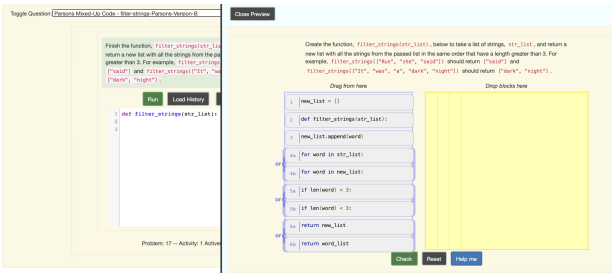

Figure 2: Screenshot of using a Parsons problem to scaffold a write-code problem

2 RELATED WORK

2.1 Parsons problems

In the original Parsons problems, Parsons and Haden provided students with a problem description and a set of drag-and-drop code fragments [53]. Each code fragment was made up of one or more lines, and some of the fragments contained incorrect code, also called distractors. To complete a problem, students chose the correct code fragments and arranged them in the correct order. They reported that most (82%) of students thought this type of problem was useful for learning [53].

Subsequent research has produced a range of Parsons problem varieties that differ by dimension, how to fill in the code line, distractor and feedback. For instance, in one-dimensional Parsons problems [53], code blocks must be organized in the right vertical sequence, while in two-dimensional Parsons problems, the blocks must additionally be appropriately indented horizontally [34]. In terms of filling in the code line, Ihantola et al. introduced MobileParsons, a mobile phone-based 2D Parsons problem that allows users to choose a piece of code from a list of options to complete the code [33]. Later, Weinman et al. invented faded Parsons problems, and in this variation, students must use valid expressions to fill in blanks in code lines and rearrange them to create a proper program [79]. Distractors can also be added to make the problems more challenging [18, 28, 53]. Distractor blocks typically include syntactic or semantic flaws. There are two types of display for distractor blocks: paired distractors, which are displayed adjacent to the correct blocks, and unpaired distractors, which are randomly mixed in with the correct blocks. Previous research has demonstrated that visually matched distractors make Parsons problems easier to solve than unpaired distractors [18]. Parsons problems provide immediate feedback in one of two ways. Block-based feedback highlights the blocks that need to be relocated or replaced [23, 53], while execution-based feedback [79] displays the results from executing the code, which often includes unit test results. In this research, we used two-dimensional Parsons problems with paired distractors (A1 in Figure 4) and block-based feedback (Figure 3).

Parsons problems have been explored for both practice [21, 25, 53] and summative assessment [13, 18, 19]. Ericson et al. found that more students attempted to solve practice Parsons problems than nearby multiple-choice questions in an interactive ebook [23]. Most of the students solved the Parsons problems in one or two attempts but some students took an unexpectedly large number of attempts

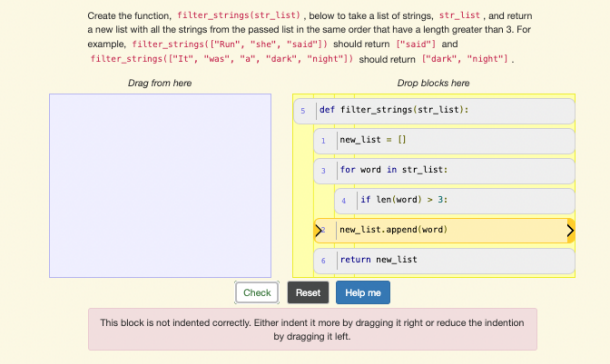

Figure 3: Screenshot of block-based feedback in a Parsons problem

to solve them and some gave up and never solved them [23]. Ericson et al. also provided evidence that students could solve Parsons problems significantly faster than either the equivalent write-code or fix-code problems with similar learning gains from pretest to posttest [22, 24]. Zhi et al. found that solving blocks-based Parsons problems improved learning efficiency on lab assignments without reducing performance on subsequent homework or programming assignments [83]. Morrison et al. also reported that Parsons problems were more sensitive than write-code problems for assessing students' learning gains [50]. While prior studies have explored Parsons problems as a type of practice or exam question, no previous research has looked into Parsons problems as a type of scaffolding for write-code problems.

Researchers have also explored changing the difficulty of a Parsons problem based on student performance to maximize learning and reduce frustration. Kumar uses student performance on a previous Parsons problem to select the next Parsons problem from a pool of possible Parsons problems [43]. Ericson et al. uses two types of adaptation for Parsons problems: intra-problem and inter-problem adaptation [21]. Intra-problem adaptation reduces the difficulty of the current Parsons problem, and inter-problem adaptation affects the difficulty of the next problem based on students' performance on the current problem [22]. Ericson et al. found that learners were nearly twice as likely to correctly solve adaptive Parsons problems than non-adaptive Parsons problems [21]. We used intra-problem adaptation in this research (A2 in Figure 4).

2.2 Scaffolding Write-Code Problems

The term "scaffolding" was coined by Bruner and colleagues to describe the process of providing support structures that enable students to learn a subject or skill that is beyond their ability [10]. Vygotsky and Cole argued that learning happens when the learner is given a task that is just beyond their ability to accomplish without assistance [76]. Appropriate scaffolding will equip learners with sufficient knowledge and skills to perform the task independently. In traditional educational settings, scaffolding usually comes from a human tutor, such as a teacher or advanced peer. For example, Bloom demonstrated that one-on-one human tutoring helps students improve their learning by two standard deviations over typical classroom instruction with a single teacher for 30 students [9]. However, in courses with a high student-to-teacher ratio, like many intro-level CS courses, this type of support would be too costly [12]. Pair programming, an example of peer-facilitated scaffolding, has also been proven to be beneficial [48]. Nevertheless, working with pairs is not convenient for out-of-class assignments, and disengagement is likely to arise due to social pressure from peers, unwelcome interruptions, and time pressure [55].

Therefore, researchers have investigated computer-assisted scaffolding techniques to support code development to address these issues. As Hmelo and Guzdial mentioned, software-enabled scaffolding can provide the assistance required for students to succeed at learning-by-doing tasks [31]. One line of research focuses on using intelligent tutoring systems (ITS) to provide formative elaborated feedback and hints [61]. In traditional ITS, authors need to spend a long time modeling the domain knowledge and tagging possible states with hint messages. This method, however, does not work for more open-ended and unstructured problem-solving environments, such as writing code [61]. Therefore, recent hint generation techniques have tried to use past student code submission data to generate next-step hints automatically [36, 45]. For example, Rivers built ITAP, which employed a three-stage process to generate next-step hints for student code submissions [61]. Her initial analysis found that students with hints spent less time practicing but achieved the same learning outcomes. Nevertheless, the inconsistency in the quality of the automated hints is still a problem, affecting students' trust in these systems and future help-seeking behaviors [57]. Furthermore, automated hints rarely contain comprehensive guidance, such as examples, that might be leveraged to overcome the "design barriers" experienced by beginner programmers [38]. As a result, by using low-level hints before starting to program a task, some novices experienced inefficient help-seeking outcomes [46].

Therefore, we investigated a more comprehensive scaffolding method by providing the equivalent adaptive Parsons problem for a write-code problem. We hypothesized that Parsons problems would be useful in scaffolding students who are having difficulty writing code from scratch because Parsons problems let students apply problem-solving skills while limiting irrelevant cognitive load [18, 71] and keep students engaged in learning [19, 53]. We next discuss cognitive load theory and how Parsons problems may reduce the cognitive load required for learning.

2.3 Cognitive load theory

According to cognitive load theory, the human cognitive architecture is made up of numerous memory stores, including a restricted working memory and an unlimited long-term memory [64]. Working memory is limited in terms of capacity and duration, especially when processing new information. Cognitive load refers to the working memory resources necessary when learning new information [69]. The original cognitive load theory distinguished three types of cognitive load: intrinsic, extraneous, and germane cognitive load [68]. However, this last category has evolved and gone through some conceptual adjustments recently [20, 52]. In Sweller's updated formulation, germane cognitive load is not an independent source of cognitive load, and the total cognitive load is determined

by the total element interactivity generated by intrinsic and extraneous cognitive load [67]. Taking another step further, Kalyuga suggested a new dual intrinsic/extraneous framework and eliminated the concept of germane load as a distinct type of cognitive load [35].

After this reconceptualization, only intrinsic and extrinsic cognitive load are distinguished as core cognitive load categories [70]. Intrinsic cognitive load relates to the material's inherent difficulty, which is mediated by the learner's prior knowledge [70]. Instructional strategies like segmenting and pre-training can be applied to manage intrinsic cognitive load due to over-complex content [47]. Extraneous cognitive load is determined by how the instructional information is delivered and what the learner is expected to perform [70]. Extraneous load can be reduced by dealing with typical instructional elements that may cause extraneous load, such as coherence, signaling, and redundancy [11]. Germane processing refers to the actual working memory resources committed to dealing with intrinsic cognitive load [70], and tends to play a minor role in the new model [20].

Computer programming is a highly cognitive skill that requires mastering multiple competencies and is recognized as being inherently difficult to learn, making cognitive load theory one of the most popular theories in computing education research [7]. Previous studies have adopted a wide range of instructional recommendations provided by cognitive load theory to programming learning. Among those effects, the usage of examples is extremely popular in introductory programming courses [7]. Worked examples demonstrate an expert's comprehensive solution to a problem, allowing students to learn how to solve problems before they can write correct code independently [1]. They are commonly used for novice learners to reduce the cognitive load [67]. However, one potential disadvantage is that worked examples have limited interactivity and do not actively engage students [70]. To address this issue, Vieira et al. attempted to combine worked examples with self-explanation prompts using code comments to attract learners' attention [75]. Other scholars recommend the use of completion problems, which provide partial solutions that students need to finish [72]. Parsons problems are a type of completion problem.

Parsons problems can be suitable scaffolding for write-code problems since they provide the necessary support for novices when they start writing code from scratch [30] while requiring students to pay active attention [15] and provide desirable difficulties compared to worked examples [70]. Desirable difficulties refer to learning challenges that, although appearing to present challenges for the learner and slow down the acquisition process, actually promote long-term retention and transfer [8]. Compared to worked examples and step-by-step hints, Parsons problems provide additional practice opportunities and may create desirable difficulties that result in more robust learning, as shown in Harms et al. [29].

3 STUDY 1: WITHIN-SUBJECTS THINK-ALOUD STUDY

The purpose of this study was to answer RQ1 and RQ2. We conducted a think-aloud study as it enabled information extraction regarding behavior and thoughts during programming problem completion. We first conducted a pilot think-aloud study with 7 undergraduate students in summer 2021 on the same six write-code problems we used in Study 1. Students thought Parsons problems were beneficial as a type of scaffold for write-code problems in general. They also provided suggestions on how to improve our experimental materials. Specifically, some students reported that having separate variable names for Parsons problems and write-code problems was confusing; therefore, we adjusted them to be the same. In addition, some students struggled to solve the Parsons problems since they were not adaptive, so we modified the problems to be adaptive (A2 in Figure 4). In fall 2021, we conducted a formal think-aloud study with 11 undergraduate students at a large public research university in the northern United States.

3.1 Method

3.1.1 Participants. We recruited participants from an introductory Python programming course at a public research university during fall 2021. Students who registered for this course received a recruitment announcement via Slack. Given that we were particularly interested in understanding the impact of using Parsons problems to scaffold novice programmers, we required that participants had less than six months of experience programming in Python. Also, following the IRB requirements, all of the participants had to be at least 18 to enroll in the study. Students who met the criteria and were willing to participate scheduled a one-hour appointment with the researcher. After finishing the study, each participant received a $25 Amazon gift card. Eleven participants participated in the study. Ten (91%) of the 11 participants were aged between 18 to 20, and one (9%) was 28. Eight (73%) of the 11 participants were female, and three (27%) were male. All participants had less than six months of Python coding experience, and for 10 (91%) of 11 participants, this introductory Python course was the most advanced programming course they had taken. Only one participant had taken a high-school CS course in C over two years ago.

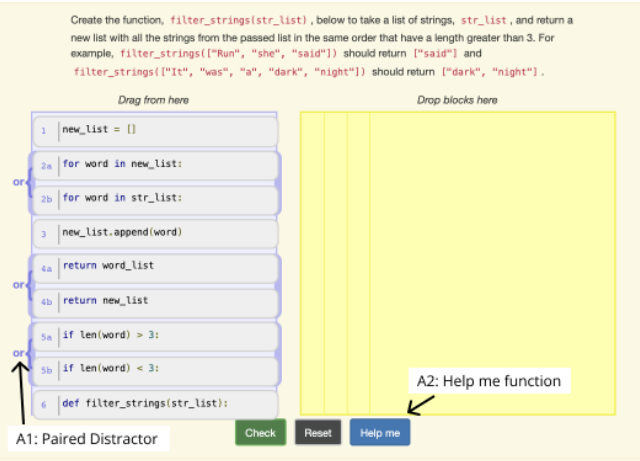

Figure 4: Screenshot of the Adaptive Parsons problem interface

3.1.2 Procedure. Each think-aloud study was conducted virtually through Zoom and lasted 60 minutes. The study started after checking participants' age and obtaining verbal consent to record. During

the session, participants started with a sample problem with verbal instructions to make them comfortable with our online environment (Figure 2). The instruction covered all of the features needed in later practice and explained the relationship between the write-code problem and Parsons problem. Participants could play around with this sample problem and ask questions. Next, we provided a brief think-aloud training with an example to help participants understand what they needed to do during the practice. After finishing the onboarding tasks, we muted ourselves and let the participants finish the six write-code problems. Each participant had 45 minutes to complete the six problems, and they were allowed to skip one or stop at any time. Following the whole think-aloud session, we conducted a 5-10 minute semi-structured interview to ask about their experience with using Parsons problems to scaffold write-code problems (e.g. "Tell me about a time when you tried to use a Parsons problem to help you solve a write-code problem and found it helpful/not helpful, and why?") and suggestions for future improvement (e.g. "What suggestions do you have for improving the interface?"). At the end of the session, participants filled out a demographic survey about their gender, age, and ethnicity.

3.1.3 Materials. This think-aloud study used a free and interactive ebook on *Runestone*. Since one of the goals of the study was to gather students' opinions on Parsons problems as scaffolding, each participant received six write-code problems with an equivalent two-dimensional adaptive Parsons problem as optional scaffolding.

Write-code Problems We sourced the write-code problems from an intermediate Python programming course at the same public research university. We categorized the problems into three difficulty levels and chose two problems from each level (easy, medium, and hard). Our purpose was to make sure that participants with different skill levels all had a chance to use a Parsons problem to scaffold a write-code problem.

Parsons problems as scaffolding One recent study found that a Parsons problem with an unusual solution increased students' cognitive load [30]. To address this problem, we generated students' most common solutions by clustering student-written code from previous semesters using Abstract Syntax Tree software, and then used the most common student solution to create the equivalent Parsons problem (Figure 4). We also wondered if it would be better to use distractors or not in the equivalent Parsons problems. So we inserted paired distractors into three of the six Parsons problems (A1 in Figure 4) and placed them interleaved across problems. The distractor blocks were created by experts based on common syntax or logic misunderstandings [53]. Two versions (A and B) were created where the only difference was if A contained distractors then B did not and vice versa as shown in Table 1. Participants were randomly assigned to either version. Additionally, a "Help Me" button was provided at the bottom of each Parsons problem to assist students who struggled while solving the Parsons problem (A2 in Figure 4). This button triggered the intra-problem adaptation which either removed a distractor or combined two blocks into one if the learner had made at least three attempts to solve the problem and had more than three blocks left in the solution.

Table 1: Distractor (D) Locations by Version

Version	Distractor (D) Location
A	D - No D - D - No D - D - No D
B	No D - D - No D - D - No D - D

3.2 Data Analysis

We recorded the screen using Zoom for all of the think-aloud sessions and transcribed the recordings. We analyzed our data using *ATLAS.ti*. To answer our research questions, we devised a coding scheme that focused on when, why, and how students used Parsons problems, as well as the perceived benefits and challenges. To create the coding scheme for the "when" part, we applied a deductive approach [49] and identified four representative stages of programming development from existing literature: problem understanding, solution planning, solution implementation, and implemented solution evaluation [17, 44]. For other sub-dimensions, we used an inductive method to generate the initial coding scheme. Then two researchers independently coded two transcripts, met to address any disagreements and refined the coding scheme iteratively. After finalizing the coding scheme, two researchers independently coded four transcripts (36% of the data), and reached an intercoder reliability score of 0.84 (alpha values > 0.80 are considered reliable [40, 78]) using Krippendorff's alpha [41], then one researcher coded the rest of the transcripts. The full code scheme can be found at https://bit.ly/3t7YpN9.

Relevant codes were grouped into themes to describe our results in relation to the research questions. When reporting our findings, we utilize (explanation) to clarify missing information in student quotes and [behavior] to indicate student behaviors.

3.3 Findings

3.3.1 When and why do students use Parsons problems to scaffold write-code problems (RQ1). In this section, we present our findings on when and why students used Parsons problems to scaffold write-code problems. We found that they opened the Parsons problem at three different stages: planning a solution (36%), implementing a solution (64%), and debugging a solution (91%). The percentages are based on the number of students who opened the Parsons problem at least once during that stage. The reasons for employing Parsons problems ("why") vary according to the stage ("when").

All eleven students (100%) in our study solved at least one Parsons problem. Some completed the Parsons problem based on the block-based feedback [30], while others relied on the intra-problem adaptation (triggered by clicking "Help Me"). P3 described this process as *"Then at some point, I was like, I don't know how to solve this. So I clicked the Help Me button, and then I got rid of one of them (Parsons blocks)."* In addition, we also noticed some students gamed the system when solving Parsons problems, where they guessed or tested various block combinations to find the correct solution. Following Baker et al., gaming means abusing the software to complete problems without having to learn [4]. We detected it mainly based on participants' verbal descriptions during the think-aloud. For example, P10 expressed how she got the right solution, *"*[reorganized some blocks] ... *so I guess I'll check this even though I don't think it's right* [clicked the check button and the solution was correct] ... *Oh,*

okay." This behavior might have negatively impacted their learning, as they did not understand the rationale behind the correct solution. We will discuss it more in 5.1.

Planning a solution: After reading the problem description, four students (36%) opened the Parsons problem because they did not know how to begin. For example, P3 stated, *"I don't know how to start here, so Parsons thing ...* [opened the Parsons problem]." And P8 was confounded when trying to tackle specific requirements in the problem description, *"I don't know how to do that adjacent part ...* [opened the Parsons problem]." Students were motivated to use the Parsons problem to help them plan their initial solution. Given that Parsons problems are constructed from a correct answer, they can be a valuable resource to start students on the correct path. Like P8 mentioned, she opened a Parsons problem once to *"check how many lines to expect ... to get an idea of how long it should be."*

Implementing a solution: Seven students (64%) chose to open the Parsons problem while they were implementing a solution. In this case, students often had at least a partial solution in mind and had already written some code. Five (71%) of the 7 students opened the Parsons problem at this stage because they were stuck. For instance, P8 wrote some code and said *"If this is an even number* [typed some code and then deleted it] *... well let me just go look at that (Parsons problem). This is ridiculous."* Two (29%) of the 7 students had difficulty translating the solution into source code using their existing Python knowledge [44]. For example, after writing 8 lines of code, and changing line 9 twice, P4 said *"Ok ... I am not sure how to skip this next number so I am gonna look at the Parsons."* Similarly, P10 explained her solution correctly verbally before writing code, but paused while writing and said *"I don't know exactly how to do that. Like, I don't think you can do ... so I'm just gonna switch to Parsons just to make sure."*

Debugging a solution: After writing a complete solution, ten students (91%) referred to the Parsons problem to correct their unit test errors for at least one problem. All ten students switched to the Parsons problem when they failed to resolve their errors. Some students already knew where their errors were before opening the Parsons problem, but could not solve them. For example, P6 changed a number first in the code while saying *"Maybe minus 1 ... let's try that ...* [checked the code but still failed unit tests] *... no"*, so she opened the Parsons problem to help her debug. Similarly, P4 diagnosed the errors correctly, but she said *"I forgot how to do the index function so I'm gonna check the Parsons."* On the other hand, some students looked at Parsons problems when they had unit test errors but did not know why the code was wrong. For instance, P11 said *"I don't know why it's not working"* when only passing 50% of the unit tests, and opened up the Parsons problem.

3.3.2 How students used Parsons problems to scaffold write-code problems (RQ1). Recall that even if students solved the Parsons problem they still had to solve the write-code problem. They could not just copy and paste the Parsons solution, they had to at least retype it. This section will focus on how students used the Parsons problem to help them finish the write-code problem. We wondered if they would just scan the Parsons problem and not try to solve it; would they only partially solve the problem until they figured out what they needed and then finish writing the code; would they solve the Parsons problem and just retype the Parsons solution (replacing their original code); or would they solve the Parsons problem and use that solution to modify their code.

Scan Parsons Problem: Because Parsons problems already contain correct code fragments, four students (36%) were able to gain information by scanning the mixed-up blocks without moving any of them. For example, P7 opened up the Parsons problem after diagnosing his syntax error and said *"Oh I don't remember the method name so check the Parsons code* [opened Parsons problem] *... so let's see ...* [found that method in one block] *... so let's go back ...* [changed his code, ran and passed all unit tests]." In another example, P3 also opened a Parsons problem after getting unit test errors and caught her syntax mistakes when comparing the mixed-up blocks with her own code, *"Oh my god, I was close* [scanned through all the blocks, and pointed to one block] *... oh the word (the variable name) ...* [closed Parsons window, modified append to append(word), ran and passed all the unit tests]." In this situation, students were able to fix their errors by looking over the mixed-up blocks, thus they didn't need to attempt to solve the Parsons problem.

Attempt Parsons Problem: Two students (18%) were able to debug their errors by only moving a few blocks rather than solving the entire Parsons problem. The drag-and-drop process allowed them to gain insight and return to work on the write-code problem. In particular, P6 opened the Parsons problem twice when planning a solution, and after dragging several blocks, she got a basic idea of how to achieve the solution and started to type in the write-code problem. Similarly, P4 opened the Parsons problem to debug, and after dragging two blocks, she saw one block and said *"I used the wrong function, let's use module ...* [modified code in write-code problem, checked and passed all the unit tests]."

Solve and replace their code: Six students (55%) deleted their own code in the write-code problem and retyped the Parsons solution in several problems. Four students chose to keep the Parsons problem window open while retyping the code to match the Parsons solution. During this process, some students self-explained the Parsons solution. For example, after getting a Parsons solution, P8 typed it in the write-code problem and said, *"let me copy it ... What does the minus one do (-1 in range(len(var)-1))?"* To answer her own question, she did some tests in her code such as removing the number. After receiving an error that said "list index out of range", she understood the usage and said, *"Oh okay, so maybe that's why it was here."* Two students wrote the Parsons solution in the write-code problem after closing the Parsons problem window. Before exiting the Parsons problem, they both explained to themselves correctly why the mixed-up blocks were placed together.

Solve and modify their code: While some students replaced their code with the Parsons solution, we found evidence that other students chose to apply what they had learned to refine their existing code. Students tended to use this approach when their primary goal was to debug their solution. Specifically, five students (45%) first examined how the Parsons solution achieved the goal, then went back to their code to refine it based on what they had learned from the Parsons solution. For example, when reflecting on how they used the Parsons problem solution in write-code problems, P5 said *"I had a general idea of how to solve it, but I was having an error message, after using (the) Parsons problem, I was able to figure out where I messed up and I was able to fix my own code."* In addition, to build a stronger connection between the solution and the written

draft, it was common for students to compare their code with the Parsons solution side by side. For instance, after understanding how the Parsons solution worked, P1 compared it with his own code and explained *"for each number if the number (explaining the Parsons solution) ... oh the one after equals to two (realizing his error) ... so I'll go back to this* [backed to change code in the write-code problem]."

3.3.3 The benefits of using Parsons problems to scaffold write-code problems (RQ2). To better understand student perceptions of the benefits and challenges of using Parsons problems to scaffold write-code problems, we conducted semi-structured interviews with the 11 undergraduate students at the end of the think-aloud sessions. First, we will present the perceived benefits.

Reduce difficulty and completion time: Six students (55%) stated that the Parsons problem aided them in lowering the difficulty of the problem and speeding up the problem-solving process. For example, P5 told us that *"It kind of, because it doesn't just give you the answer right off the bat. They gave me like a good start to it. And then from there, I was able to figure (it) out easily."* Parsons problems also reduced the debugging time. For example, P11 stated, *"I just used the Parsons problems to check a small mistake I did ... And I found that instead of divide, I should do (another method)."* Another student said Parsons problems are preferable to searching on Google as *"there's more than one way to solve some coding problems (so it takes time to find), and it's a little frustrating, especially when your teachers want one method."*

Learn problem-solving strategies: Most students (91%) found Parsons problems helpful to gain a clearer understanding of how to create a correct solution. Some students obtained new strategies, for example, P4 explained that *"It got me to think how the these steps are like this, like the reasons behind it and why I should do it like this, it made me think about the process of the steps that I have to take to solve them."* For those who already had a plan for how to solve the problem, working on Parsons problems helped them to develop more effective strategies. Like P2 described, *"When I was using a for loop, and then I saw that the Parsons problem used a while loop. It helps me realize that would be a better solution to that problem."*

Refine and extend existing programming knowledge: A Parsons solution contains information about programming language syntax and semantics. Most students (91%) mentioned that they clarified misunderstandings by using Parsons problems. As P3 said, *"But once I like, looked up the Parsons thing, I knew right away, first of all, I got this number wrong. Like I understood why because it was a simple mistake."* In addition, Parsons solutions serve as good examples to teach new programming language syntax. For instance, P7 explained how Parsons problem taught him syntax knowledge, *"I don't remember the Python syntax. So I use a Parsons problem to view its syntax, then (I) get a better idea of how the code should be written."*

Prompt to think more deeply: When asked about the helpfulness of Parsons problems as a scaffolding tool, P4 compared it with providing an answer directly, and claimed that *"I really enjoyed it (Parsons problem) because the process made me think, it is going to be useful when I solve a different problem that is similar, rather than just like giving the answers, it really makes me think."* More students (45%) shared something similar when asked about the use of distractors blocks in Parsons problems. Despite the fact that distractors made the problem more challenging, they also pushed students to think and learn more. As P8 stated, *"I feel like I don't want everything to be easy for me because I want to learn and so I think this one (the one with distractors) was good."* Students also gained a deeper understanding by distinguishing between the distractors and correct blocks. Like P11 said, *"... it (the one with distractors) did help because it showed how a small difference will not work. And I feel like for other ones (the one without distractors) ... then you don't think about like if you did it a different way it wouldn't work."*

3.3.4 The challenges of using Parsons problems to scaffold write-code problems (RQ2). While there is evidence that Parsons problems provided a wide range of help to scaffold students' code writing, some participants encountered challenges during this process.

Have difficulty solving the Parsons problem: Four students (36%) faced this challenge during learning. Given that we used the most common solution to create the Parsons problem, it was possible that students who had a different approach in mind would experience a higher cognitive load. For example, P9 skipped one question and stated, *"I was thinking differently on how to approach it, so it's just the Parsons problem (solution) didn't seem reasonable to me."* Three students (27%) also complained about the use of distractors as they made it easier to get stuck, as P3 explained *"I prefer the ones without the distractors. Because a couple of times I used the wrong choice, so that wasn't very helpful. And that kind of led me astray a little bit. So it'd be easier just to have one option that is necessary in the solution."*

Have difficulty understanding the Parsons solution or adaptation process: Four students (36%) reported this as a challenge. As previously mentioned, students might game (intentionally or unintentionally) the system to achieve the correct solution, so they did not gain a proper understanding of the solution and thus felt that they had gained very little from solving the Parsons problem. For instance, when looking at the Parsons solution she created, P8 said, *"What am I doing? I mean, I kind of understood it but not really. Because I don't know what the slash slash (floor division) means."* Sometimes the intra-problem adaptation confused the students, as they did not understand the intention or what to do after the adaptation. As P3 explained, *"All it did was eliminate an option, it just felt like a guessing game rather than the actually learning ... And that didn't like help me quite understand. I was still stuck in the same problem that I was beforehand."* P6 shared the same feeling, and said, *"I tried to use the "Help Me" button, but it just kind of confused me more ... it took it (a block) away from my code, but it still said I needed more like lines of code."*

Help Avoidance: This refers to situations in which students did not use the help features even when they were struggling. We encountered two types of help avoidance: avoiding the use of the Parsons problem and avoiding the use of the intra-problem adaptation (triggered by clicking the "Help Me" button). Only one student (P7) clearly expressed his avoidance of Parsons problems as *"(it) isn't really a hint, it's like the answer."* He would prefer more high-level guidance on the logic instead of a Parsons problem with solution blocks. Two students (18%) avoided using the "Help Me" button due to a desire for independence [58]. As P10 explained, *"I*

feel like I'd rather try to figure it out myself first. So I didn't really want the answer to be kind of given to me more easily."

4 STUDY 2: BETWEEN-SUBJECTS CLASSROOM STUDY

After getting evidence that students found Parsons problems helpful while solving write-code problems in Study 1, we investigated the effect on learning in an authentic classroom environment. We conducted the study at a large public research university in the northern United States in the winter-spring semester of 2022. All participants were enrolled in a data-oriented programming course, which was the second required Python course for the university's information science majors. This course covered topics including Python basic data structures (list, tuple, and dict), object-oriented programming, debugging, unit testing, web scraping, regular expressions, HTML, XML, JSON, working with APIs, working with databases, and Matplotlib.

4.1 Methods

4.1.1 Participants and Procedure. The classroom study was conducted during the 80-minute lecture period in week three of the class; students who did not attend the lecture were allowed to finish by the end of the day. Ninety-six students participated in this study. Students were randomly assigned to one of two conditions: Parsons-Help condition (PH) and No-Help condition (NH). Similar to Study 1, students in the Parsons-Help (PH) condition received a text-entry write-code interface with an equivalent two-dimensional adaptive Parsons problem as scaffolding (Figure 2), while the No-Help (NH) group only had the write-code interface (Figure 1). Students took a timed pretest before solving the practice problems and a timed posttest immediately after the practice, each for 20 minutes. Students were instructed to spend 25 minutes solving the five practice problems. After the study, the data from 15 students was removed since they had not completed all of the materials or did not follow the instructions. Our final sample included 81 students (43 students in the PH condition and 38 students in the NH condition).

4.1.2 Materials. Study 2 used five of the six problems from study 1. The first problem was removed since most students in Study 1 solved it easily without needing the Parsons problem. The course instructor designed the pretest and isomorphic posttest questions to measure students' understanding of related topics. Three types of questions were included in the pretest and posttest: multiple-choice questions, Parsons problems, and write-code problems. Each test had four multiple-choice questions (2.5 points for each, 10 points in total), one Parsons problem (10 points), and two write-code problems (10 points for each, 20 points in total). Parsons problems were scored by the number of lines in the correct location with the correct indentation. Write-code problems were scored based on the percentage of unit tests that passed. There were a total of 40 possible points per test.

4.2 Results

The distribution of the pretest score and the pre-posttest learning gain did not meet the normality assumption, so we conducted a group of Mann–Whitney U tests to compare the difference between the two groups [51]. Given that these were non-parametric tests, we utilized common-language effect size (CLES) as the effect size, using a brute-force version of the formula given by Vargha and Delaney [74]. There was no significant difference between the PH group and the NH group in pretest performance (Table 2), which suggests the randomization of our experiment was successful. However, there was a ceiling effect in the pretest, with both groups scoring highly. We then compared students' learning gains by question type, as seen in Table 3. Results indicated that there was no significant difference between the two groups in their pre-posttest learning gain. However, there was no significant learning gain from pretest to posttest. This experiment was conducted in week three, and the material was on Python basics. The material appears to have been too easy for these students based on the high pretest score medians (Table 2). Study 1 had been conducted on students from the prerequisite course who had less experience than these students.

We then compared the students' completion time between the two conditions. This was the time to complete the five practice problems. We first tested the relationship between completion time and pretest, and there was no significant correlation between completion time and pretest score, r = -0.03, p = 0.773. We then applied a Mann–Whitney U test to compare the completion time between the two conditions, as the completion time data was not normally distributed. Our result found that the 43 participants in the PH condition (Mdn = 19.42 mins) had a significantly shorter completion time versus the 38 participants in the NH condition (Mdn = 23.94 mins), U = 504.5, p = 0.003, CLES = 0.31. We also examined the students' practice outcomes between the two conditions. Practice outcome was the percentage of unit tests students passed across the five practice problems (e.g., 0.7 means that the student passed 70% of the unit tests in the five practice problems). We found that there was no significant difference in the practice outcomes between conditions, U = 900.0, p = 0.420, CLES = 0.55, even though students in the PH condition had higher practice outcomes (Mdn = 0.91) than students in the NH condition (Mdn = 0.80).

Overall, students in the Parsons problem as scaffolding condition had a 19% reduction in their completion time, from a median value of 23.94 minutes to 19.42 minutes. Despite spending less time on practice, students achieved an equal level of pretest (Table 2) to posttest (Table 4) performance in both conditions. This suggests that utilizing Parsons problems as scaffolding could help students achieve equivalent outcomes (posttest scores) in significantly less practice time. However, the median learning gain from pretest to posttest was zero for both groups, which may mean that our experiment materials were too fundamental for these students. The high median pretest scores also indicate that the students had good knowledge of the content they were tested on already.

5 DISCUSSION AND DESIGN SUGGESTIONS

In this study, we investigated the use of Parsons problems to scaffold write-code problems. We conducted two studies to evaluate the effectiveness of our approach. Our think-aloud study (Study 1) demonstrated that Parsons problems helped students solve the write-code problems, learn new problem-solving strategies, and refine their programming knowledge. The classroom study (Study 2) indicated that students with Parsons problems as scaffolding used

Table 2: Pretest in Parsons-Help (PH) group vs. No-Help (NH) group

Category (Max score)	Pretest (*Mdn*, *M (SD)* in score)				Mann-Whitney U		
	PH group (n = 43)		NH group (n = 38)		*U*	*p*-value	CLES
Multiple-Choice question (10)	10.0	8.6 (2.06)	7.5	8.09 (2.21)	931.5	0.230	0.57
Parsons problem (10)	10.0	9.46 (1.4)	10.0	8.35 (2.97)	959.0	0.082	0.59
Write-code problem (20)	18.18	14.03 (7.6)	18.18	15.08 (7.07)	777.5	0.702	0.48

Table 3: Pre-posttest learning gain in Parsons-Help (PH) group vs. No-Help (NH) group

Category	Learning gain (min, *Mdn*, max, *M (SD)* in score)								Mann-Whitney U		
	PH group (n = 43)				NH group (n = 38)				*U*	*p*-value	CLES
Multiple-Choice question	-5	0	5	-0.12 (2.11)	-2.5	0	5	0.33 (1.66)	741.5	0.391	0.45
Parsons problem	-8.57	0	4.29	-1.21 (3.01)	-8.33	0	8.57	-0.47 (3.84)	811.5	0.959	0.50
Write-code problem	-5.0	0	20	0.72 (5.39)	-10.0	0	10	-0.76 (3.91)	828.5	0.914	0.51

significantly less time to complete practice problems compared to those who did not. However, there were no significant learning gains from pretest to posttest in either condition, which might indicate that our experiment materials were too simple for participants in Study 2. The high median pretest scores provide evidence that these students already had a good grasp of the tested concepts. We will discuss our results for RQ1 and RQ2 in this section, and RQ3 in the next section.

5.1 RQ1: Help-seeking behaviors in writing code

We answered the "when" and "why" parts in RQ1 by revealing three representative scenarios in Study 1, where students used Parsons problems to scaffold their code writing. In general, those scenarios are consistent with the help-seeking strategies identified in the literature in non-programming domains [2], such as using help when students do not know how to begin and when debugging. Our findings also surfaced specific help-seeking behaviors for write-code problems. For example, our study revealed an important scenario where students asked for help when they had a solution in mind but had difficulty translating it into code [44]. In this case, Parsons problems helped students translate the algorithm from plain English to code.

When it comes to the "how" part of RQ1, we found that students interacted with the Parsons problems in four different ways. First, in "Scan Parsons Problem" and "Attempt Parsons Problem", students referred to Parsons problems to correct syntax errors, consistent with the Glossary mechanism in ITS [2]. This helps students quickly get information about syntax and apply it to the write-code problem. Second, in "Solve and replace their code", Parsons problems were used as a differentiated instruction tool [26] so that beginners with less prior knowledge were not overwhelmed by the open-ended write-code problem. In addition, in "Solve and modify their code", students finished solving the Parsons problem first and used it as a worked example to facilitate further understanding [66].

However, as mentioned in the challenge part, we observed some "Help Avoidance" in Study 1. A small group of students avoided utilizing the "Help Me" function or even the Parsons problem when they were stuck, which has previously been seen in other programming scaffolding research [46, 82]. To address this problem, we plan to use prompts to encourage students to seek help once the system detects many failed attempts. In addition, we also noticed gaming behaviors when students were completing the Parsons problems, where they tried to get the problems right without paying active attention. Previous research suggested two possible reasons for such behaviors when students interacted with intelligent tutoring systems: (1) the primary goal of the students is to perform rather than learn, and (2) a lack of ability and actively avoiding challenges [5]. Future work should further investigate the underlying reasons of gaming behaviors, and help students to avoid them, such as increasing students' self-efficacy through the right level of scaffolding.

5.2 RQ2: Opportunities for personalized Parsons problems to scaffold writing code

Semi-structured interviews in Study 1 revealed various benefits of utilizing Parsons problems as scaffolding for students' performance and learning, as well as challenges of using Parsons problems as scaffolding in our current design (RQ2). One significant difference between Parsons problems and writing code from scratch is that Parsons problems constrain the problem space [71]. Parsons problems break down a complex write-code problem into more manageable tasks [24], which is one possible explanation for the benefit of reducing difficulty and completion time. Based on the challenges and benefits we discovered in our studies, we propose three design ideas to improve the future practice of using Parsons problems as scaffolding.

5.2.1 Personalize the Parsons problem based on the student's solution. The first design idea is inspired by the challenge that students had difficulty solving the Parsons problems and that students benefited from learning problem-solving strategies. In our study, we utilized the most common solution from student-written code to

Table 4: Posttest in Parsons-Help (PH) group vs. No-Help (NH) group

Category (Max score)	Posttest (*Mdn*, *M (SD)* in score)			
	PH group (n = 43)		NH group (n = 38)	
Multiple-Choice question (10)	10.0	8.49 (2.06)	10	8.42 (2.06)
Parsons problem (10)	10.0	8.25 (1.4)	10.0	7.88 (1.4)
Write-code problem (20)	17.5	14.75 (7.6)	16.25	14.32 (7.6)

generate the Parsons problems, which could differ from the solution path the current student was following. Under this circumstance, some students were able to distinguish and examine both approaches and learn more problem-solving strategies. This is consistent with a previous study [54] that found having students view distinct code in parallel helped them achieve better learning of procedural knowledge and develop higher flexibility in problem-solving. Nevertheless, this may create a higher cognitive load for other students and become a challenge. For example, some students failed to solve the Parsons problem or did not find it helpful, as the Parsons problem used a different strategy than their approach. This suggests that personalized Parsons problems (e.g., ones that are closest to the student's current problem-solving strategy) or personalized feedback (e.g., explanations of the connections between the Parsons blocks and the student code) are needed to better support students and further decrease the unnecessary cognitive load.

5.2.2 Provide additional support on Parsons solution comprehension. The second design idea is motivated by the challenge that students had difficulty understanding the Parsons solution or the adaptation process and the benefit that some students used Parsons problems to refine and extend their prior knowledge. For example, some students self-explained while solving the Parsons problem or reflected on the difference between the Parsons problem solution and their own incorrect solution. These students are more likely to identify inadequacies in their prior knowledge, correct misinterpretation, and demonstrate conceptual change and learning from Parsons problems [44]. Nevertheless, other students found it challenging to understand the Parsons solution due to a lack of prior knowledge [14]. This result implies that we could provide further support for beginners to comprehend the Parsons problem solution and reflect on the solution. This suggestion is consistent with previous research, which discovered that instructional explanations can be beneficial under certain conditions because they effectively support knowledge-construction activities [60]. Our participants also provided similar recommendations, as P8 suggested, *"I wish there was like an explanation about what's like [pointed to one block in Parsons solution]."* In addition, recall that the intra-problem adaptation confused some students as it removed or combined blocks without any explanation. Therefore, adding more explanation could increase the effectiveness of the current adaptation process.

5.2.3 Dynamically adjust the difficulty of the Parsons problem. The third design idea is to augment the benefit of Parsons scaffolding in prompting students to think more deeply and address the challenge that students might have difficulty solving the Parsons problems we provide. Our findings revealed that some students valued the opportunity to think more deeply by solving the Parsons problems, which is consistent with previous results on the desirable difficulty introduced by adaptive Parsons problems [22]. Instead of receiving a passive "bottom-out" hint, solving Parsons problems and recognizing common mistakes from paired distractors could improve students' ability to identify errors. On the other hand, some students became more perplexed specifically because of the distractors and had difficulty solving the Parsons problems.

Inspired by the conflicting opinions on distractors and requirements of high-level guidance, we need to increase the adaptivity of Parsons problems to provide more personalized and targeted scaffolding when novices are writing code. In other words, Parsons problems can be constructed dynamically depending on the student's problem-solving status when seeking assistance. Specifically, for those who already have an idea of what to do, skeleton-style Parsons problems can be provided, such as those that only contain subgoal labels [24]. In this way, students may compare their thought processes with the Parsons problem skeleton to decide on next-step goals. For students with less experience, we could remove distractors or combine some blocks in advance to provide a simpler Parsons problem. Testing adaptive Parsons problems at different difficulty levels provides further research opportunities to address the "assistance dilemma" in determining the appropriate amount of support for novice programmers [39].

6 LIMITATIONS AND FUTURE WORK

For RQ3, although we found in Study 2 that students who received Parsons problems as scaffolding saved significant practice time, we did not observe a significant difference in learning between the two conditions. One potential reason is that, despite the fact that Study 1 and Study 2 used identical learning materials, students in Study 2 were recruited from a more advanced Python course (usually the second Python course an information science major would take) compared to Study 1. Students were expected to have more prior programming knowledge. There might be a ceiling effect where students were already competent in the topics we covered. Another limitation of this work is that we performed the studies at only one public university in the United States. We might see different help-seeking scenarios and scaffolding effects with other demographic groups and contexts such as MOOCs. Future work needs to explore diverse demographic groups, additional topics, and programming languages.

Our goal is to employ Parsons problems as a scalable and effective scaffolding tool for write-code problems for students with different levels of ability and/or knowledge. We are excited to pursue future work based on the design suggestions mentioned above. First, more research is needed to fully understand the effects of using Parsons problems as scaffolding for write-code problems for diverse student

groups, contexts, topics, and programming languages. Second, we plan to provide personalized scaffolding by generating a Parsons problem that is closest to the student's incorrect solution. Third, we plan to collect log data that records how students write code and seek help on the platform to better understand their behavior.

7 CONCLUSION

In this work, we presented two studies on using Parsons problems to scaffold write-code problems. Study 1 identified three representative scenarios, four use cases, and reported student perceptions on the benefits and challenges of using Parsons problems as scaffolding. Our findings showed that this approach can help students refine and extend existing knowledge, reduce the difficulty of the problem, reduce the problem completion time, learn problem-solving strategies, and reflect on their code writing process. In Study 2, we found that using Parsons problems to scaffold write-code problems could significantly save students' practice time. Moving forward, we plan to improve our design and increase the adaptivity of the scaffolding to enhance student learning.

ACKNOWLEDGMENTS

We thank Zihan Wu who contributed to the qualitative analyses and the reviewers who provided valuable suggestions that improved this paper. Some of the funding for this research came from the National Science Foundation award DUE-2143028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Siti-Soraya Abdul-Rahman and Benedict Du Boulay. 2014. Learning programming via worked-examples: Relation of learning styles to cognitive load. *Computers in Human Behavior* 30 (2014), 286–298.

[2] Vincent Aleven, Bruce McLaren, Ido Roll, and Kenneth Koedinger. 2006. Toward meta-cognitive tutoring: A model of help seeking with a Cognitive Tutor. *International Journal of Artificial Intelligence in Education* 16, 2 (2006), 101–128.

[3] Vincent Aleven, Bruce M Mclaren, Jonathan Sewall, and Kenneth R Koedinger. 2009. A new paradigm for intelligent tutoring systems: Example-tracing tutors. *International Journal of Artificial Intelligence in Education* 19, 2 (2009), 105–154.

[4] Ryan Baker, Jason Walonoski, Neil Heffernan, Ido Roll, Albert Corbett, and Kenneth Koedinger. 2008. Why students engage in "gaming the system" behavior in interactive learning environments. *Journal of Interactive Learning Research* 19, 2 (2008), 185–224.

[5] Ryan Shaun Baker, Albert T Corbett, Kenneth R Koedinger, and Angela Z Wagner. 2004. Off-task behavior in the cognitive tutor classroom: when students" game the system". In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 383–390.

[6] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. When life and learning do not fit: Challenges of workload and communication in introductory computer science online. *ACM Transactions on Computing Education (TOCE)* 12, 4 (2012), 1–38.

[7] João Henrique Berssanette and Antonio Carlos de Francisco. 2021. Cognitive Load Theory in the Context of Teaching and Learning Computer Programming: A Systematic Literature Review. *IEEE Transactions on Education* (2021).

[8] Robert A Bjork. 1994. Memory and meta-memory considerations in the training of human beings. *Metacognition: Knowing about knowing* 185, 7.2 (1994).

[9] Benjamin S Bloom. 1984. The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational researcher* 13, 6 (1984), 4–16.

[10] Jerome Seymour Bruner et al. 1966. *Toward a theory of instruction*. Vol. 59. Harvard University Press.

[11] Ünal Çakiroğlu and Dilara Arzugül Aksoy. 2017. Exploring extraneous cognitive load in an instructional process via the web conferencing system. *Behaviour & Information Technology* 36, 7 (2017), 713–725.

[12] Yan Chen, Jaylin Herskovitz, Gabriel Matute, April Wang, Sang Won Lee, Walter S Lasecki, and Steve Oney. 2020. EdCode: Towards Personalized Support at Scale for Remote Assistance in CS Education. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–5.

[13] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating Coding Ability Without Writing any Code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 123–128.

[14] Michelene TH Chi, Miriam Bassok, Matthew W Lewis, Peter Reimann, and Robert Glaser. 1989. Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science* 13, 2 (1989), 145–182.

[15] Michelene TH Chi and Ruth Wylie. 2014. The ICAP framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist* 49, 4 (2014), 219–243.

[16] Elizabeth A Davis and Naomi Miyake. 2018. Explorations of scaffolding in complex classroom systems. In *The journal of the learning sciences*. Psychology Press, 265–272.

[17] Fadi P Deek, Murray Turoff, and James A McHugh. 1999. A common model for problem solving and program development. *IEEE Transactions on Education* 42, 4 (1999), 331–336.

[18] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. 2008. Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop on computing education research*. 113–124.

[19] Yuemeng Du, Andrew Luxton-Reilly, and Paul Denny. 2020. A review of research on Parsons problems. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*. 195–202.

[20] Rodrigo Duran, Albina Zavgorodniaia, and Juha Sorva. 2022. Cognitive Load Theory in Computing Education Research: A Review. *ACM Transactions on Computing Education (TOCE)* (2022).

[21] Barbara Ericson, Austin McCall, and Kathryn Cunningham. 2019. Investigating the affect and effect of adaptive parsons problems. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.

[22] Barbara J Ericson, James D Foley, and Jochen Rick. 2018. Evaluating the efficiency and effectiveness of adaptive parsons problems. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 60–68.

[23] Barbara J Ericson, Mark J Guzdial, and Briana B Morrison. 2015. Analysis of interactive features designed to enhance learning in an ebook. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. 169–178.

[24] Barbara J Ericson, Lauren E Margulieux, and Jochen Rick. 2017. Solving parsons problems versus fixing and writing code. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*. 20–29.

[25] G Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. A comparison of different types of learning activities in a mobile Python tutor. (2017).

[26] Kathy Gibbs and Wendi Beamish. 2020. Differentiated instruction: A programming tool for inclusion. In *Inclusive theory and practice in special education*. IGI Global, 174–191.

[27] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.

[28] Kyle James Harms, Jason Chen, and Caitlin L Kelleher. 2016. Distractors in Parsons problems decrease learning efficiency for young novice programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. 241–250.

[29] Kyle J Harms, Noah Rowlett, and Caitlin Kelleher. 2015. Enabling independent learning of programming concepts through programming completion puzzles. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 271–279.

[30] Carl C Haynes and Barbara J Ericson. 2021. Problem-Solving Efficiency and Cognitive Load for Adaptive Parsons Problems vs. Writing the Equivalent Code. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[31] Cindy E Hmelo and Mark Guzdial. 1996. Of black and glass boxes: Scaffolding for doing and learning. (1996).

[32] Minh Ho Chi and Syed Mustapha SMFD. 2018. Automated Data-Driven Hint Generation in Intelligent Tutoring Systems for Code-Writing: On the Road of Future Research. *International Journal of Emerging Technologies in Learning (iJET)* 3, 9 (2018), 174–189.

[33] Petri Ihantola, Juha Helminen, and Ville Karavirta. 2013. How to study programming on mobile touch devices: interactive Python code exercises. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. 51–58.

[34] Petri Ihantola and Ville Karavirta. 2011. Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10, 2 (2011), 119–132.

[35] Slava Kalyuga. 2011. Cognitive load theory: How many types of load does it really need? *Educational Psychology Review* 23, 1 (2011), 1–19.

[36] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2016. Towards a systematic review of automated feedback generation for programming exercises. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*. 41–46.

[37] Hassan Khosravi, Kirsty Kitto, and Joseph Jay Williams. 2019. Ripple: a crowdsourced adaptive platform for recommendation of learning activities. *arXiv preprint arXiv:1910.05522* (2019).

[38] Amy J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *2004 IEEE Symposium on Visual Languages-Human Centric Computing*. IEEE, 199–206.

[39] Kenneth R Koedinger and Vincent Aleven. 2007. Exploring the assistance dilemma in experiments with cognitive tutors. *Educational Psychology Review* 19, 3 (2007), 239–264.

[40] Klaus Krippendorff. 2004. Reliability in content analysis: Some common misconceptions and recommendations. *Human communication research* 30, 3 (2004), 411–433.

[41] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability. (2011).

[42] Amruth N Kumar. 2013. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 183–188.

[43] Amruth N Kumar. 2018. Epplets: A tool for solving parsons puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 527–532.

[44] Dastyni Loksa, Amy J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, problem solving, and self-awareness: Effects of explicit guidance. In *Proceedings of the 2016 CHI conference on human factors in computing systems*. 1449–1461.

[45] Mariam MAHDAOUI, NOUH Said, My Seddiq ELKASMI ALAOUI, and Mounir SADIQ. 2022. Comparative study between automatic hint generation approaches in Intelligent Programming Tutors. *Procedia Computer Science* 198 (2022), 391–396.

[46] Samiha Marwan, Anay Dombe, and Thomas W Price. 2020. Unproductive help-seeking in programming: What it is and how to address it. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 54–60.

[47] Richard E Mayer and Roxana Moreno. 2003. Nine ways to reduce cognitive load in multimedia learning. *Educational psychologist* 38, 1 (2003), 43–52.

[48] Charlie McDowell, Linda Werner, Heather E Bullock, and Julian Fernald. 2003. The impact of pair programming on student performance, perception and persistence. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 602–607.

[49] Matthew Miles, Michael Huberman, and J Saldana. 2013. SAGE: Qualitative data analysis: A methods sourcebook.

[50] Briana B Morrison, Lauren E Margulieux, Barbara Ericson, and Mark Guzdial. 2016. Subgoals help students solve Parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. 42–47.

[51] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology* 4, 1 (2008), 13–20.

[52] Fred Paas, Tamara Gog, and John Sweller. 2010. Cognitive Load Theory: New Conceptualizations, Specifications, and Integrated Research Perspectives. *Educational Psychology Review* 22 (06 2010), 115–121. https://doi.org/10.1007/s10648-010-9133-8

[53] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.

[54] Elizabeth Patitsas, Michelle Craig, and Steve Easterbrook. 2013. Comparing and contrasting different algorithms leads to increased student learning. In *Proceedings of the ninth annual international ACM conference on International computing education research*. 145–152.

[55] Laura Plonka, Helen Sharp, and Janet Van Der Linden. 2012. Disengagement in pair programming: does it matter?. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 496–506.

[56] James Prather, Raymond Pettit, Brett A Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci. 2019. First things first: Providing metacognitive scaffolding for interpreting problem prompts. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 531–537.

[57] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29, 3 (2019), 368–395.

[58] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors influencing students' help-seeking behavior while programming with human and computer tutors. In *Proceedings of the 2017 ACM Conference on international computing education research*. 127–135.

[59] Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017. Hint generation under uncertainty: The effect of hint quality on help-seeking behavior. In *International conference on artificial intelligence in education*. Springer, 311–322.

[60] Alexander Renkl. 2002. Worked-out examples: Instructional explanations support learning by self-explanations. *Learning and instruction* 12, 5 (2002), 529–556.

[61] Kelly Rivers. 2017. *Automated data-driven hint generation for learning programming*. Ph.D. Dissertation. Carnegie Mellon University.

[62] Kelly Rivers and Kenneth R Koedinger. 2014. Automating hint generation with solution space path construction. In *International Conference on Intelligent Tutoring Systems*. Springer, 329–339.

[63] Kelly Rivers and Kenneth R Koedinger. 2017. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (2017), 37–64.

[64] Wolfgang Schnotz and Christian Kürschner. 2007. A reconsideration of cognitive load theory. *Educational psychology review* 19, 4 (2007), 469–508.

[65] Sue Sentance and Andrew Csizmadia. 2017. Computing in the curriculum: Challenges and strategies from a teacher's perspective. *Education and Information Technologies* 22, 2 (2017), 469–495.

[66] Leonardo Silva, António José Mendes, Anabela Gomes, and Gabriel Fortes Cavalcanti de Macêdo. 2021. Regulation of learning interventions in programming education: A systematic literature review and guideline proposition. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 647–653.

[67] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational psychology review* 22, 2 (2010), 123–138.

[68] John Sweller. 2011. Cognitive load theory. In *Psychology of learning and motivation*. Vol. 55. Elsevier, 37–76.

[69] John Sweller and Paul Chandler. 1994. Why some material is difficult to learn. *Cognition and instruction* 12, 3 (1994), 185–233.

[70] John Sweller, Jeroen JG van Merriënboer, and Fred Paas. 2019. Cognitive architecture and instructional design: 20 years later. *Educational Psychology Review* 31, 2 (2019), 261–292.

[71] Jeroen JG Van Merriënboer. 1990. Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of educational computing research* 6, 3 (1990), 265–285.

[72] Jeroen JG Van Merriënboer and Marcel BM De Croock. 1992. Strategies for computer-based programming instruction: Program completion vs. program generation. *Journal of Educational Computing Research* 8, 3 (1992), 365–394.

[73] Jeroen JG Van Merrienboer and Hein PM Krammer. 1987. Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science* 16, 3 (1987), 251–285.

[74] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[75] Camilo Vieira, Alejandra J Magana, Michael L Falk, and R Edwin Garcia. 2017. Writing in-code comments to self-explain in computational science and engineering education. *ACM Transactions on Computing Education (TOCE)* 17, 4 (2017), 1–21.

[76] Lev Semenovich Vygotsky and Michael Cole. 1978. *Mind in society: Development of higher psychological processes*. Harvard university press.

[77] Christopher Watson and Frederick WB Li. 2014. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. 39–44.

[78] Field M Watts and Solaire A Finkenstaedt-Quinn. 2021. The current state of methods for establishing reliability in qualitative chemistry education research articles. *Chemistry Education Research and Practice* 22, 3 (2021), 565–578.

[79] Nathaniel Weinman, Armando Fox, and Marti A Hearst. 2021. Improving Instruction of Programming Patterns with Faded Parsons Problems. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–4.

[80] Jacqueline Whalley and Nadia Kasto. 2014. How difficult are novice code writing tasks? A software metrics approach. In *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*. 105–112.

[81] Benjamin Xie, Dastyni Loksa, Greg L Nelson, Matthew J Davidson, Dongsheng Dong, Harrison Kwik, Alex Hui Tan, Leanne Hwa, Min Li, and Amy J Ko. 2019. A theory of instruction for introductory programming skills. *Computer Science Education* 29, 2-3 (2019), 205–253.

[82] Rui Zhi, Min Chi, Tiffany Barnes, and Thomas W Price. 2019. Evaluating the effectiveness of parsons problems for block-based programming. In *Proceedings of the 2019 ACM Conference on International Computing Education Research*. 51–59.

[83] Rui Zhi, Thomas W Price, Samiha Marwan, Alexandra Milliken, Tiffany Barnes, and Min Chi. 2019. Exploring the impact of worked examples in a novice programming environment. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 98–104.