A Review of Research on Parsons Problems

Yuemeng Du
ydu541@aucklanduni.ac.nz
University of Auckland
Auckland, New Zealand

Andrew Luxton-Reilly
andrew@cs.auckland.ac.nz
University of Auckland
Auckland, New Zealand

Paul Denny
paul@cs.auckland.ac.nz
University of Auckland
Auckland, New Zealand

ABSTRACT

Parsons problems are a type of programming exercise where students rearrange jumbled code blocks of a solution program back into its original form. It is usually implemented as a complement or alternative to traditional programming exercises like code-tracing and code-writing. This paper reviews the existing literature on the Parsons problem in introductory CS education. We find that the flexible nature of the design of Parsons problems has led to many variants, and these have been continuously refined to better address student needs. However, the effectiveness of Parsons problems, both as a question type and as a learning tool in CS education, remains uncertain due to a lack of replicated research in the field.

KEYWORDS

Parsons problem, Parsons puzzle, programming exercise

ACM Reference Format:
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 (ACE'20), February 3–7, 2020, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3373165.3373187

1 INTRODUCTION

Introductory programming courses must necessarily teach students the syntax of programming and mechanical learning to a certain extent. At many universities, this is achieved through exercises that involve code-writing. Research have suggested that reliance on traditional programming exercises often decrease student engagement and motivation [4], which directly influence student performance. Furthermore, tasks such as code-writing can often end up being challenging and time-intensive to students in ways unintended by the instructors. A potential solution to this is to provide a novel way for students to learn to program, one that imposes less cognitive load.

To this end, in recent years, there has been a trend of increased interest from researchers in studying the use of block-based programming in beginner CS classrooms. This is partially due to the general perception that block-based programming exercises are easier than many traditional programming tasks [41]. Notable initiatives that use the question type include the CS Principles Course [1] and the Exploring Computer Science Curriculum [17]. However, these are mainly informal programming courses that target younger (high school) students, serving as preparation for future university study.

At university level, block-based programming exercises are less frequently implemented. Nevertheless, the Parsons problem, a drag-and-drop style, block-based program construction exercise, has been a notable and consistent presence in introductory CS university courses. Unlike traditional code-writing exercises, students are supplied with code fragments that are already written, which relieves a considerable amount of cognitive load. Furthermore, Parsons problems can be easily implemented in an online or mobile environment, making it easier for students to engage in learning in real-time.

A survey of existing literature shows that, while there is existing research on the adoption of Parsons problems, the results are scattered and concern themselves with different aspects of CS education. This paper is a review of the literature surrounding Parsons problems in introductory CS education at the university level. It aims to provide an overview of how the Parsons problem has evolved over the years, both in terms of the question type itself and how it is implemented. The following research questions are addressed in subsequent sections:

(1) Why are Parsons problems studied in CS education?
(2) What are the features of Parsons problems?
(3) How are Parsons problems used in CS education?

2 METHODOLOGY

We conducted a systematic review of the literature using the guidelines proposed by [26]. The main steps include the identification of research questions, selection of primary studies, assessment of quality of the studies, data extraction and synthesis.

To identify primary studies, we conducted a search of the ACM digital library using the search term:

> "parsons problem" OR "parson's problem" OR "parsons puzzle" OR "parson's puzzle" or "parsons programming" OR "parson's programming"

The ACM search automatically includes plurals, so it was not necessary to distinguish between "puzzle" and "puzzles". This search identified 29 potential papers. Subsequently, we searched IEEE Xplore using the same search phrase resulting in 245 papers. Finally, we searched Google Scholar using the same search phrase and identified 51 potential papers.

The title and abstract of each paper was examined to determine the relevance. The papers in the ACM digital library were all relevant, which is unusual in a systematic review, but likely resulted from the very specific terms that were used both in the search and in the literature to refer to a particular named activity. Papers in Google Scholar were also all relevant, but those in the IEEE reflected

ACE'20, February 3–7, 2020, Melbourne, VIC, Australia

ACM ISBN 978-1-4503-7686-0/20/02…$15.00
https://doi.org/10.1145/3373165.3373187

a wide range of other topics and only 3 were identified as being relevant. Duplicates were removed at this stage.

To assess the quality of the work, we read the full text. We rejected any paper that was not peer reviewed, and any publication that was 2 pages or less in length (e.g., abstracts, lightning talks, panels).

Finally, we identified 34 primary studies that were of sufficient quality.

3 OVERVIEW

Figure 1 shows the primary studies according to their year of publication. We note that the original paper defining the Parsons problem was published in 2006; the most recent paper reviewed was published in 2019. As an overall trend, we observe a small but increasing amount of published literature contributing to the Parsons problem over the years, since the problem's conception.

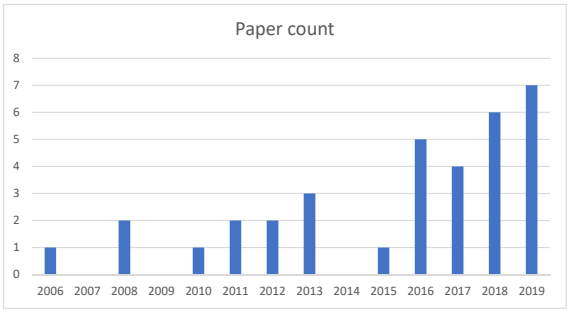

Figure 1: Amount of Parsons problems papers published by year.

4 WHY ARE PARSONS PROBLEMS STUDIED IN CS EDUCATION?

This section examines the reported motivations behind using Parsons problems as a question type for introductory CS education. Parsons and Haden [36] designed the original problem according to the following principles: maximising student engagement, separating logic from syntax, providing immediate feedback, and modelling good design. Existing literature seem to suggest that these principles have been achieved to a satisfiable extent, as they are commonly reported among the motivations for studying Parsons problems.

4.1 Identifying Student Difficulties

In common programming exercises, it is often difficult to separate syntactical units from the logical context in which they occur. As a result, students' responses to these exercises frequently make it unclear to instructors what specific concepts they are struggling with. Parsons problems give instructors the potential to test programming concepts, be it syntax or logic, separately; therefore, student responses from Parsons problems have potential to be more meaningful.

Denny et al. [6] were interested in whether Parsons problems could provide them with information on student performance that code-tracing and open-ended code-writing questions cannot provide, in a paper-based exam setting.

Helminen et al. [22] were aware of the importance of identifying student misconceptions and correcting common student errors in the early stages of programming study. They used interaction data traces on Parsons problems to identify common student errors during problem solving.

4.2 Providing Immediate Feedback

A crucial factor to maintaining student engagement and improving student performance in introductory CS courses is to provide continuous, timely feedback [3]. This can be a particular challenge since many first-year CS1 courses have large cohorts. Online Parsons problems allow students to request immediate feedback on their solution and is therefore of interest to researchers.

Helminen et al. [21] were interested in automated feedback and investigated the effects of line-based vs execution-based feedback in the context of online Parsons problems.

Ericson et al. [9] implemented Parsons problem in their study of interactive e-books as a form of student exercise that provides immediate feedback to the user.

4.3 Improving Student Engagement

Student engagement is important to effective education. While active engagement in the form programming exercises is a major component of traditional CS courses, research have shown that reliance on traditional tasks alone contributes to students dropping out [4]. Parsons problems provide an alternative incentive for students to engage in active learning.

Ericson et al. [10] reports that one of the reasons they chose to compare Parsons problems with tradition exercises was that evidence suggests that Parsons problems are engaging to students. This finding is supported in other studies in the form of student feedback and collected data [6, 7, 9, 36].

4.4 Reducing Cognitive Load

Introductory programming courses often expect students to complete challenging, high cognitive load tasks such as code-writing. According to cognitive load theory, the cognitive load of complex tasks often needs to be reduced for learning to occur [39].

Due to the part-complete nature of the Parsons problem, Ericson et al. [10] studies the problem as a lower cognitive load alternative to code-writing, and evaluates respective performances.

Garcia et al. [15] chose to apply Parsons problems as a development tool in designing planning skills, since they found the problems were shown to reduce cognitive load with positive outcomes.

5 WHAT ARE THE FEATURES OF PARSONS PROBLEMS?

This section examines literature on the design of the Parsons problem, and addresses the outcome of its many variants; we aim to

provide an overview of the problem's development. The usage of the types of Parsons variants in the studies mentioned below is summarised in Table 1.

Parsons and Haden [36] describe a Parsons problem as a drag-and-drop style coding exercise where "in each problem context, the student is given a selection of (randomly mixed) code blocks, some subset of which comprise the problem solution". The student is required to correctly choose and order the code blocks to form the problem solution; after each attempt automated feedback may be requested on their answer. The setup of Parsons problems allows great flexibility in its design, which perhaps most naturally lead to variations in *scaffolding, distractors and feedback.*

5.1 Pre-scaffolded vs Student-scaffold

Parsons problems that are pre-scaffolded allow students to focus only on the ordering of code blocks. This can be achieved by: a) providing a 'skeleton outline' of the problem solution, such as using braces in Java [6]; or more commonly, b) providing correctly formatted code blocks by default [8, 34].

Intuitively, Parsons problems that require student scaffolding are harder. Denny et al. [6] compared paper-based pre-scaffolded Parsons problems with their code-equivalent counterparts that required student scaffolding; almost all students reported the latter to be more difficult.

5.1.1 Two-dimensional Parsons problems. Ihantola and Karavirta [24] introduced a class of student-scaffold Parsons problems named two-dimensional Parsons problems (2D Parsons), and also found them to be harder than pre-scaffolded problems. Based on Python, where indentation defines scaffolding, 2D Parsons require students to both correctly order and indent code blocks. Existing literature suggests that this has since become the standard template for implementing student-scaffolded Parsons problems [9, 10, 22, 25].

5.2 Distractors

In Parsons problems, distractors are code blocks that are closely related to, but not a part of the problem solution [10]; typically, they contain syntax or logic errors. Researchers generally agree that distractors effectively illustrate common programming misconceptions to introductory programming undergraduate students [10, 25, 36]. Particularly, in Parsons problems, they have the potential to highlight specific concepts a student is struggling with [36], something instructors often find difficult to determine when examining student work.

Distractors frequently feature in implementations of Parsons problems [6, 10],[8, 9, 24]. While the initial implementation by Parsons and Haden [36] included distractors, Denny et al. [6] expanded on the idea to assess more distractor types. Part I of their study involved CS2 undergraduates working through 5 Parsons variants with the same problem context; the variants included 3 distractor levels: *no* distractors, *paired* distractors, and *jumbled* distractors. In the paired level, distractors were explicitly paired with their corresponding correct alternatives (binary-choice indicator); in the jumbled level, all code blocks were mixed with no indicators. Think-aloud sessions and subsequent student interview feedback revealed jumbled distractors to be somewhat 'overwhelming' and is especially exacerbated when combined with student scaffolding. Some students had the false impression that multiple solutions were available; some tried to guess instructor intentions rather than problem solve. They conclude that the extra cognitive load imposed by jumbled distractors appear to be unrelated to programming. While the study used a range of Parsons variants, and the students were closely observed, it only included 13 participants from a single North American university, an evidently small and possibly unrepresentative sample base.

Later studies following Denny et al. [6] have scarcely implemented a student-scaffolded, jumbled-distractor Parsons variant, with the exception of Ihantola and Karavirta [24], but they tested on senior teachers and assistants who were experienced in the topic domain. This may be due to the reluctance of researchers to impose redundant cognitive load on students; others may fear the variant is too difficult. However, the use of paired distractors and jumbled distractors with scaffolding is common.

5.3 Line-based Feedback

Parsons and Haden's [36] initial line-based feedback system was basic; code blocks in the wrong absolute positions were highlighted. Ihantola and Karavirta [24] sought to improve line-based feedback by designing the js-parsons tool for 2D Parsons problems. It offered the following features: the first code block in the wrong absolute position is highlighted in red (preceding correct blocks are in green); if code ordering is correct, the first incorrectly indented block is labelled in red; the background colour is green or red, depending on whether the student solution has the correct number of blocks.

This system had its weaknesses. In follow-up work, Helminen et al. [22] comment that the median usage of feedback was sparingly low; yet some students used the trial-and-error approach to solve the exercise and requested excessive feedback. Others showed episodes of no progress and were stuck in solution loops (i.e., they traversed through and resubmitted a series of solution states that they have already tested previously) [22, 25]. To amend this, Karavirta et al. [25] updated the feedback system in their mobile js-parsons tool. To avoid students changing correctly ordered code blocks, feedback is provided for partial solutions. An algorithm finds the longest common sequence (LCS) between the student solution and the model solution; code blocks outside the LCS are highlighted and only those can be moved. Subsections 3.3.1 and 3.3.2 discuss further improvements. Note that the statistics provided are estimations applied over a dataset of student interaction traces from a previous study [22]; the findings of the paper [25] are speculative since the updated feedback system has not been tested on students.

5.3.1 Feedback on looping behaviour. If the student requests feedback on a solution state that has been requested before, they are informed of their looping behaviour. A hint tab offers feedback on the current state, and on the state they moved onto the last time they were here. They claim that 14.7% to 23.4% of students from the dataset (who exhibit looping behaviour) will be benefited [25]. However, no experiment data is available to support if, and how much student behaviour would change.

5.3.2 Feedback penalty. When the student requests feedback, the application checks the number of times the student has done so

Table 1: Summary of the types of Parsons variants used.

Authors	Year	Distractors	Scaffolding	Feedback
Parsons and Haden [36]	2006	jumbled	pre	absolute line-based
Denny et al. [6]	2008	jumbled + paired	pre + student	—
Ihantola and Karavirta [24]	2011	jumbled	student	absolute line-based
Helminen et al. [22]	2012	none	student	relative line-based
Karavirta et al. [25]	2012	paired	student	relative line-based
Helminen et al. [21]	2013	none	student	relative line-based + execution-based
Ericson et al. [9]	2015	none	student	relative line-based
Morrison et al. [34]	2016	none	pre	—
Ericson et al. [10]	2017	paired	student	relative line-based
Ericson et al. [8]	2018	jumbled + paired	pre	relative line-based

Table 2: Summary of existing tools available for Parsons problems.

Name	Year	Distractor support	Indentation support	Languages	Feedback
Hot Potatoes [36]	2006	yes	no	BASIC	absolute line-based
CORT [16]	2007	yes	yes	BASIC	execution-based; copy and paste to interpreter
VILLE [37]	2007	no	no	independent	execution-based; students can see line-by-line visualization
js-parsons [25]	2011	yes	yes	Python	relative line-based
Epplets [27]	2018	yes	yes	C+, C#, Java	relative line-based; every student action logged on feedback panel

in the last 45s; if this exceeds the limit (a summative scheme is used), the feedback button is deactivated for a period of 15s to 45s. An estimated 21% to 30% of students from the dataset would have received a first penalty [25]. Again, there is no data to show how student behaviour (particularly for trial-and-error students) would change after the penalty.

5.4 Execution-based Feedback

Following from their previous work, Helminen et al. [21] suggest that line-based feedback has several drawbacks. Firstly, it may encourage trial-and-error behaviour; secondly, it requires that the Parsons problem have a *unique* problem solution, which significantly limits the size and types of questions that can be represented.

Execution-based feedback is a potential alternative. To compare its effectiveness to line-based feedback, Helminen et al. conducted a study on students from a CS1 university course (N=445), where students were split into two groups that differed in feedback type for a subset (of 2) of the Parsons problems they were given. Using interaction traces data on one of the problems, they found no significant difference on the number of feedback requests between the two groups. However, the execution group had many long solution sequences; they requested feedback less frequently and required more time, on average, to complete the problem [21]. In terms of quality, upon requesting feedback, the execution group had significantly fewer states where the code failed to execute, and contained fewer errors.

Though the results seem to suggest that execution-based feedback increases problem difficulty and encourages students to think more carefully on their solution, the study has its limitations. The Parsons problem studied was a short (6 blocks) 2D Parsons problems regarding loop structure. We are unaware of the effect of feedback type in longer, more complex problems, where execution-based feedback may prove to be beneficial by forcing students to focus on program structure instead of fragmented details. Furthermore, it is unsure whether effects of feedback type may vary on other problem topics.

5.5 Other Variants

Briefly acknowledged here are some infrequently implemented variants of the Parsons problem. These variants differ from the setup of traditional Parsons problems through altered features such as supplying part-complete (instead of fully completed) code blocks that require student input for completion [23]; and supplying surrounding context code so the Parsons problem actually completes a subsection of the program (instead of being the entire program itself) [16].

A potential reason why Parsons variants are implemented infrequently may be that their alterations partially defeat the purpose of using the problems in the first place, as discussed in section 4. Particularly, this may be the case for examination questions. For example, allowing students to complete code blocks can limit the ability of an instructor to test for specific programming concepts; embedding parsons problems in a larger problem context may induce unnecessary cognitive load, they may also test skills that are better suited to other question types (e.g., code-writing).

5.6 Tools

The tools used for implementing Parsons problems are not a focus of this review; however, Table 2 provides an overview of the available software and their functionality with respect to the design elements discussed previously. However, it is worth noting that Parsons problems are increasingly being incorporated into learning resources such as textbooks [9], online tutors [11, 12], integrated development environments [37], web-based apps [27–29], mobile devices [23], mobile games [35] and as a form of block-based programming [42].

6 HOW ARE PARSONS PROBLEMS USED IN CS EDUCATION?

After discussing research focused on the design of the Parsons problem itself, we discuss its place within the broader realm of CS education. Parsons problems offer an alternative question type to traditional programming questions such as code-tracing. This section examines existing literature that investigates the use of Parsons problems in introductory CS education, what their findings are, and provides an evaluation of the findings. Table 3 summarises the implementation methods of the papers discussed below.

6.1 As a Paper-based Exam Question

Studies in this area mostly concerns how well student performance on Parsons problems correlate with performance on other exam question types; the underlying assumption here is that a stronger correlation likely implies that a similar set of skills or knowledge is involved (and therefore, is assessed during exams).

As a part of the BRACElet project on improving teaching to novice programmers, Lopez et al. [33] conducted a study investigating the relationships between reading, tracing and writing code. Data was taken from a paper-based final exam of a CS1 Java programming course, where 38 of 78 students agreed to give data (this was claimed to be a representative sample). One Parsons problem that examined code sequencing ability was included; results were surprising – students achieved an impressive average percentage mark of 83% (4.9 out of 6), which ranked 6th highest among the 13 total exam questions. Among the questions ranked higher, 4 were entry-level questions. Statistical analysis suggested that Parsons problems may require skills slightly lower than code-tracing. However, they acknowledged that the problem used (no distractors, pre-scaffolded) may have been too easy; it is possible some students simply applied shallow heuristics to complete the question. We think this casts serious doubt on the validity of the finding.

Extending the work of Lopez et al., Lister et al. [31] replicated the study with 7 final exam datasets across different introductory programming courses, 5 of which included Parsons problems, of those 2 are relevant to this review. Dataset PA (N=330) is from a CS1 course teaching C language for engineering students. Data analysis of student performance shows statistically significant correlation coefficient between Parsons and code-tracing (0.542); and even higher for Parsons and code writing (0.702). However, the exam included just one Parsons problem and it is unsure whether the results were somewhat incidental. Dataset PM included students (N=76) from the second half of a full-year introductory programming course teaching Pascal, some of which are not novices. The 18-question exam included 4 Parsons problems. Pairwise analysis of results revealed no particular highlight between Parsons problems and code-writing, tracing or explaining questions respectively.

Denny et al. [6] studied the final exam of the summer school CS1 course (N=74) at the University of Auckland in 2008 . The exam included 11 questions, one each of 2D Parsons problem, code-writing and code-tracing; all 3 were on iterating through array structures. Data analysis revealed significant correlation between Parsons and code-writing (0.53), but none between Parsons and code-tracing. However, it was noted the latter was particularly difficult on the occasion: a step required in the problem appeared to confuse many students, causing them to focus on a single line (out of 3) in the loop structure presented. Furthermore, the ceiling effect poses a considerable threat to validity, as a great number of students scored perfectly on all 3 questions (particularly on the Parsons Problem).

6.2 As a Tool for Student Learning

6.2.1 A more efficient learning tool? Ericson et al. [10] conducted a study comparing the effectiveness of Parsons problems on student learning with both code-fixing and code-writing. Students were selected from a Python-based CS1 course at an US university and split into 3 groups. The study was non-compulsory; students received 2.5 extra credits for attending each of the 2 sessions. After filtering out unusable data, 135 and 82 student datasets were collected from the sessions respectively. The study was a between subjects design that included one pretest and two posttests (the first immediately after, and the second a week later). The independent variable was the practice condition – interleaved worked examples and practice problems; this was one of Parsons, code-fixing or code-writing. In posttest measurements, they found that students across the groups experienced performance gains. No practice condition outperformed the others; in fact, no interactions were found between the practice condition and posttest questions of that specific condition. However, for every practice problem, the Parsons group had significantly lower completion time, on average, than the other groups. They conclude that the findings support the hypotheses that 1) Parsons problems are a more efficient practice form; and 2) would lead to comparable learning performance, when measured against fixing code errors or writing the equivalent code.

However, the study had a simple flaw: there was no control group, so it remains unsure whether student performance gain is from the practice condition or simply from having completed the pretest. The immediate posttest was identical to the pretest; the second posttest was isomorphic; learning gains may well have been from answering the same or similar questions with correctness feedback.

Ericson et al. [8] followed up on the previous study to compare the effectiveness of adaptive and non-adaptive Parsons problems with writing code. Implementation remained identical to the previous study, except a control group that solved off-task activities (instead of the practice condition) was added. It was found that *only* the adaptive Parsons group performed statistically significantly better than the off-task control group; the other groups show no difference. Furthermore, no analysis on the second posttest was

Table 3: Summary of the context used in studies discussed in section 6

Authors	Year	Course Description	Reporting Data Student Count	Language	Format
Lopez et al. [33]	2008	CS1	38	Java	paper-based exam
Denny et al. [6]	2008	CS1 summer school	74	Java	paper-based exam
Lister et al. [31] PA	2010	CS1 for engineering students	330	C	paper-based exam
Lister et al. [31] PM	2010	2nd half of full-year CS1	76	Pascal	paper-based exam
Ericson et al. [10]	2017	CS1	135 (82)	Python	2 voluntary sessions worth 2.5 extra credits each
Ericson et al. [8]	2018	CS1	—	Python	2 voluntary sessions worth 2.5 extra credits each
Garcia et al. [15]	2018	CS1 procedural programming	120	—	2 voluntary Parsons exercise accompanying assignment

provided and we do not know if the practice conditions had effect on information retention.

6.2.2 A tool for learning program design? Garcia et al. [15] conducted a preliminary study on students from a CS1 procedural programming course to investigate whether Parsons problems can serve as a scaffolding tool to help students learn program design. Students were issued an averaging problem assignment and an optional Parsons problem containing associated tasks. Out of the students (N=120) who attempted the Parsons problem, those that had minimal passes (N=9) on their first interaction scored higher (93.7%) on the assignment than the class average (86.9%); those that did not complete the problem (N=11) scored much lower (67.6%), though this was skewed downwards by 2 students who failed to submit assignments and thereby received 0 marks. No conclusions were drawn; more investigation, such as student feedback, may help to understand whether Parsons problems were beneficial to the design process, or whether its performance were simply an indication of student ability.

6.3 Other Uses

Briefly acknowledged here are papers where Parsons problems were not directly studied, but served as a tool for their research objective. Morrison et al. [34] added subgoal labels to Parsons problems to investigate whether performance gains from subgoal labelling is transferrable onto a different question type. Ericson et al. [9] used Parsons problems as a feature in a study of the effectiveness of interactive e-books.

7 DISCUSSION

This section discusses the observations and potential implications of the literature review findings presented above.

We examined the progression of the Parsons problem in terms of features. It is evident that through the years, student feedback provided by (web-based) Parsons problems have improved. This is demonstrated by the replacement of absolute line-based feedback by relative line-based feedback since 2012, and the existence of increasingly functional and powerful tools to support Parsons problems. Furthermore, researchers have explored execution-based feedback as a means to provide better guidance or encourage deeper learning. We also note that certain combinations of the Parsons problem seem to be no longer in use due to perceived difficulty.

The role of Parsons problems in CS education has been explored in two primary aspects. One of these aspects is the usefulness of Parsons problems as an exam type question. The answer to how Parsons problems relates to traditional question types such as code-writing and code-tracing is not entirely conclusive. While an earlier study suggested that a Parsons problem requires skills slightly lower than that of code-tracing [33], later studies have suggested that it has more in common with code-writing [6], or simply shown no meaningful correlation to other question types [31]. Common threats to validity within these studies include potential ceiling effects of student scores on certain question types; furthermore, the exams often offered just one Parsons problem to extract data from. We are aware of the difficulties in designing exam questions; however, studies in the future should aim to address these threats to validity in the hopes of reaching a conclusion that can more justified.

The second aspect concerns whether Parsons problems is a more efficient but equally effective learning tool, when compared to traditional programming exercises like code-writing and code-fixing. Likewise, the results remain inconclusive. Certain studies so far have had considerable design flaws that affected the validity of the data [10]; others were unable to draw fully-supported conclusions [8]. While it is safe to say Parsons problems are more time-efficient, we are unsure whether it brings about the same learning gain; nor do we have enough evidence to know its impact on knowledge retention. Future work requires the refinement of study design to bring about more meaningful findings, for example, by shifting more focus onto examining isomorphic posttests.

As reflected above, in general, a primary limitation surrounding current research into Parsons problems is the lack of replication and further investigation into existing work. Furthermore, there is an arguably intuitive assumption in current research that if student performance on Parsons correlates strongly with performance on another question type (in particular, code-writing), both question types should be testing a similar underlying set of skills. To this review's knowledge, there is insufficient evidence that exists to back up this claim. This leaves an area open for future work, possibly through the form of think-aloud sessions.

Finally, the research field of using Parsons problems as a tool for learning more abstract programming skills, such as programming design, is very much an open one. We hope to see more papers that investigate a diverse range of uses for Parsons problems in the near future.

7.1 Research Limitations

We use a fairly narrow definition of the Parsons problem: a drag-and-drop style coding exercise that uses *complete* code blocks to form self-contained programs. Papers reporting on Parsons variants outside of this definition are not examined by this review.

Additionally, some of the papers in this review were a part of the BRACElet project [31, 33] which involved many datasets. In these studies, Parsons problems only featured insignificantly (e.g., as a question type for one of the datasets); we are unable to obtain the specific experiment conditions and data to critically analyse the findings reported by these papers.

8 CONCLUSION

Our literature review shows that since its conception, there has been some continuous interest from researchers in CS education on the Parsons problem. Researchers have identified a number of desirable features in Parsons problems that can improve traditional programming courses.

The design of Parsons problems itself has been refined over the years [6, 24, 36], and continues to be evolving in order to better cater to student needs and enhance student learning[8].

REFERENCES

[1] Owen Astrachan and Amy Briggs. 2012. The CS principles project. *ACM Inroads* 3, 2 (2012), 38–42.

[2] A. T. M. Golam Bari, Alessio Gaspar, R. Paul Wiegand, Jennifer L. Albert, Anthony Bucci, and Amruth N. Kumar. 2019. EvoParsons: design, implementation and preliminary evaluation of evolutionary Parsons puzzle. *Genetic Programming and Evolvable Machines* 20, 2 (01 Jun 2019), 213–244. https://doi.org/10.1007/s10710-019-09343-7

[3] Theresa Beaubouef and John Mason. 2005. Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin* 37, 2 (2005), 103–106.

[4] 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. *Trans. Comput. Educ.* 12, 4, Article 15 (Nov. 2012), 38 pages. https://doi.org/10.1145/2382564.2382567

[5] Aparna Chirumamilla and Guttorm Sindre. 2019. E-Assessment in Programming Courses: Towards a Digital Ecosystem Supporting Diverse Needs?. In *Digital Transformation for a Sustainable Society in the 21st Century*, Ilias O. Pappas, Patrick Mikalef, Yogesh K. Dwivedi, Letizia Jaccheri, John Krogstie, and Matti Mäntymäki (Eds.). Springer International Publishing, Cham, 585–596.

[6] 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 (ICER '08)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/1404520.1404532

[7] Darina Dicheva and Austin Hodge. 2018. Active Learning Through Game Play in a Data Structures Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 834–839. https://doi.org/10.1145/3159450.3159605

[8] 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 (ICER '18)*. ACM, New York, NY, USA, 60–68. https://doi.org/10.1145/3230977.3231000

[9] 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 (ICER '15)*. ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/2787622.2787731

[10] 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 (Koli Calling '17)*. ACM, New York, NY, USA, 20–29. https://doi.org/10.1145/3141880.3141895

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

[12] GV Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2018. Supporting Novices and Advanced Students in Acquiring Multiple Coding Skills. (2018).

[13] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2018. Adaptive Problem Selection in a Mobile Python Tutor. In *Adjunct Publication of the 26th Conference on User Modeling, Adaptation and Personalization (UMAP '18)*. ACM, New York, NY, USA, 269–274. https://doi.org/10.1145/3213586.3225235

[14] Rita Garcia. 2017. Codification Pedagogy for Introductory Courses. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. ACM, New York, NY, USA, 263–264. https://doi.org/10.1145/3105726.3105727

[15] Rita Garcia, Katrina Falkner, and Rebecca Vivian. 2018. Scaffolding the Design Process Using Parsons Problems. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research (Koli Calling '18)*. ACM, New York, NY, USA, Article 26, 2 pages. https://doi.org/10.1145/3279720.3279746

[16] Stuart Garner. 2007. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Issues in Informing Science & Information Technology* 4 (2007).

[17] Joanna Goode, Gail Chapman, and Jane Margolis. 2012. Beyond curriculum: the exploring computer science program. *ACM Inroads* 3, 2 (2012), 47–53.

[18] Kyle J. Harms. 2017. *Code Puzzle Completion Problems in Support of Learning Programming Independently*. Ph.D. Dissertation. https://search.proquest.com/docview/1889856556?accountid=8424

[19] K. J. Harms, E. Balzuweit, J. Chen, and C. Kelleher. 2016. Learning programming from tutorials and code puzzles: Children's perceptions of value. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 59–67. https://doi.org/10.1109/VLHCC.2016.7739665

[20] 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 (ICER '16)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2960310.2960314

[21] J. Helminen, P. Ihantola, V. Karavirta, and S. Alaoutinen. 2013. How Do Students Solve Parsons Programming Problems? — Execution-Based vs. Line-Based Feedback. In *2013 Learning and Teaching in Computing and Engineering*. 55–61. https://doi.org/10.1109/LaTiCE.2013.26

[22] Juha Helminen, Petri Ihantola, Ville Karavirta, and Lauri Malmi. 2012. How Do Students Solve Parsons Programming Problems?: An Analysis of Interaction Traces. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. ACM, New York, NY, USA, 119–126. https://doi.org/10.1145/2361276.2361300

[23] 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 (Koli Calling '13)*. ACM, New York, NY, USA, 51–58. https://doi.org/10.1145/2526968.2526974

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

[25] Ville Karavirta, Juha Helminen, and Petri Ihantola. 2012. A Mobile Learning Application for Parsons Problems with Automatic Feedback. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research (Koli Calling '12)*. ACM, New York, NY, USA, 11–18. https://doi.org/10.1145/2401796.2401798

[26] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report EBSE 2007-001. Keele University and Durham University Joint Report. http://www.dur.ac.uk/ebse/resources/Systematic-reviews-5-8.pdf

[27] Amruth N. Kumar. 2018. Epplets: A Tool for Solving Parsons Puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. ACM, New York, NY, USA, 527–532. https://doi.org/10.1145/3159450.3159576

[28] Amruth N. Kumar. 2019. Helping Students Solve Parsons Puzzles Better. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '19)*. ACM, New York, NY, USA, 65–70. https://doi.org/10.1145/3304221.3319735

[29] Amruth N. Kumar. 2019. Mnemonic Variable Names in Parsons Puzzles. In *Proceedings of the ACM Conference on Global Computing Education (CompEd '19)*. ACM, New York, NY, USA, 120–126. https://doi.org/10.1145/3300115.3309509

[30] Amruth N. Kumar. 2019. Representing and Evaluating Strategies for Solving Parsons Puzzles. In *Intelligent Tutoring Systems*, Andre Coy, Yugo Hayashi, and Maiga Chang (Eds.). Springer International Publishing, Cham, 193–203.

[31] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, et al. 2010.

Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[32] Erno Lokkila, Erkki Kaila, Rolf LindÃŠn, Mikko-Jussi Laakso, and Erkki Sutinen. 2016. Developing a Technology Enhanced CS0 Course for Engineering Students. *International Association for Development of the Information Society* (2016).

[33] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[34] 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 (SIGCSE '16)*. ACM, New York, NY, USA, 42–47. https://doi.org/10.1145/2839509.2844617

[35] Solomon Sunday Oyelere, Jarkko Suhonen, and Teemu H. Laine. 2017. Integrating Parson's Programming Puzzles into a Game-based Mobile Learning Application. In *Proceedings of the 17th Koli Calling International Conference on Computing Education Research (Koli Calling '17)*. ACM, New York, NY, USA, 158–162. https://doi.org/10.1145/3141880.3141882

[36] 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 (ACE '06)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. http://dl.acm.org.ezproxy.auckland.ac.nz/citation.cfm?id=1151869.1151890

[37] Teemu Rajala, Mikko-Jussi Laakso, Erkki Kaila, and Tapio Salakoski. 2007. VILLE: a language-independent program visualization tool. In *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research-Volume 88*. Australian Computer Society, Inc., 151–159.

[38] Teemu Sirkiä. 2016. Combining Parson's Problems with Program Visualization in CS1 Context. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research (Koli Calling '16)*. ACM, New York, NY, USA, 155–159. https://doi.org/10.1145/2999541.2999558

[39] John Sweller. 2010. Cognitive load theory: Recent theoretical advances. (2010).

[40] Anderson Thomas, Troy Stopera, Pablo Frank-Bolton, and Rahul Simha. 2019. Stochastic Tree-Based Generation of Program-Tracing Practice Questions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 91–97. https://doi.org/10.1145/3287324.3287492

[41] David Weintrop and Uri Wilensky. 2015. To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-based Programming. In *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15)*. ACM, New York, NY, USA, 199–208. https://doi.org/10.1145/2771839.2771860

[42] 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 (ICER '19)*. ACM, New York, NY, USA, 51–59. https://doi.org/10.1145/3291279.3339419