# Scaffolding Novices: Analyzing When and How Parsons Problems Impact Novice Programming in an Integrated Science Assignment


Benyamin Tabarsi
North Carolina State University
Raleigh, North Carolina, USA
btaghiz@ncsu.edu

Heidi Reichert
North Carolina State University
Raleigh, North Carolina, USA
hreiche@ncsu.edu

Nicholas Lytle
Georgia Institute of Technology
Atlanta, North Carolina, USA
nlytle3@gatech.edu

Veronica Catete
North Carolina State University
Raleigh, North Carolina, USA
vmcatete@ncsu.edu

Tiffany Barnes
North Carolina State University
Raleigh, North Carolina, USA
tmbarnes@ncsu.edu



## ABSTRACT

**Background and Context.** The importance of CS to 21st-century life and work has made it important to find ways to integrate learning CS and programming into the regular school day. However, learning CS is difficult, so teachers integrating programming need effective strategies to scaffold the learning. In this study, we analyze students' log data and apply a novel technique to compare Parsons Problems with from-scratch programming in a middle school science class.

**Objectives.** Our research questions aimed to investigate whether, how, and when Parsons Problems improve learning efficiency for a programming exercise within science, utilizing log data analysis and an automated progress detector (SPD).

**Method.** We conducted a study on 199 students in a 6th-grade science course, divided into two groups: one engaged with Parsons problems, and the other, a control group, worked on the same programming task without scaffolding. Then, we analyzed differences in performance and coding characteristics between the groups. We also adopted an innovative application of SPD to gain a better understanding of how and when Parsons problems helped students make more progress on the coding task, with an objective measure of final student grades.

**Findings.** The experimental group, with scaffolding through Parsons Problems, achieved significantly higher grades, spent significantly less time programming, and toggled less between block category tabs. Interestingly, they ran their code more frequently compared to the control group. The SPD analysis revealed that the experimental group made significantly higher progress in all four quartiles of their coding time.

**Implications.** Our findings suggest that Parsons problems can improve learning efficiency by enhancing novices' learning experience without negatively impacting their performance or grades, which is especially important when programming is integrated into K12 courses.




*ICER '24 Vol. 1, August 13–15, 2024, Melbourne, VIC, Australia*



ACM ISBN 979-8-4007-0475-8/24/08
https://doi.org/10.1145/3632620.3671110



## CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; **K-12 education**; • **Human-centered computing** → *Empirical studies in HCI*.

## KEYWORDS

Novice Programming, Block-based Programming, Data-driven Methods, Parsons Problems, Progress

**ACM Reference Format:**
Benyamin Tabarsi, Heidi Reichert, Nicholas Lytle, Veronica Catete, and Tiffany Barnes. 2024. Scaffolding Novices: Analyzing When and How Parsons Problems Impact Novice Programming in an Integrated Science Assignment. In *ACM Conference on International Computing Education Research V.1 (ICER '24 Vol. 1), August 13–15, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3632620.3671110


## 1 INTRODUCTION

Because of its importance to 21st-century life and work, computer science (CS) has become increasingly common in K-12 schools in non-CS classes [55]. At the secondary level, there has been increased attention on strategies for students learning CS concepts, particularly in teaching programming and algorithmic thinking [25, 45]. Several research efforts have studied pedagogical techniques and curriculum design for teaching programming in non-CS classes [5, 33, 41]. Learning programming during non-CS classes poses unique challenges, including courses' constrained time, varying levels of teacher proficiency, diverse student experience levels, and the inherent complexity of programming tasks and environments.

Scaffolding is one approach to help students learn more in a shorter amount of time by providing support for students as they learn. Scaffolding can take many forms, but in the context of programming, it often looks like starter code and Parsons problems [42]. This provides students with a subset or all of the correct code, albeit in a disorganized order [50], before they begin working on a project. Prior research has shown that this form of scaffolding can reduce cognitive load [16, 73]. The end result of reducing cognitive load is similar (or improved) performance but with less time spent on tasks [73].

There are a number of strategies for evaluating scaffolding's effectiveness, ranging from comparing assessment scores to timing task completion rates [59, 65]. However, assessing an intervention in an integrated programming lesson can be difficult for teachers and may harm student learning when some students have scaffolding and others do not. One approach to address this gap is analyzing students' coding log data or trace logs. Trace log analysis can help us understand the effectiveness of an intervention method at each step of a student's work, potentially pinpointing where the method has the most effect on a student. It can also define the rate at which a student attains progress or proximity to correctness, as well as observe characteristics in programming that may be associated with certain traits. For instance, the number of times a student runs their program may be associated with debugging or tinkering [12]. Yet despite trace log analysis's promise for understanding exactly *how* students progress through their programming, this strategy is still relatively novel.

In this study, we explore the influence of scaffolding on students' programming skills during integrated classroom programming activities, contributing to the growing body of research on scaffolding, and specifically Parsons problems, for secondary students. Specifically, we compare the outcomes between two groups: one provided with scaffolding through Parsons problems, and another without such support. Our focus is on sixth-grade students engaging in programming through a science class activity in a block-based programming environment (BBPE). We aim to look at whether, how, and when Parsons problems improve learning efficiency for programming exercises, using three specific research questions:

- **RQ1:** How do students' programming behaviors change with and without scaffolding through Parsons problems?
- **RQ2:** How do Parsons problems impact students' scores on the specific programming activity?
- **RQ3:** How can an automatic progress detector assess the impact of Parsons problems on students' programming performance over time?

The research questions reflect three techniques that constitute a novel, triangulated method to gain a better understanding of scaffolding impacts on learning by looking at what students do (RQ1), their final code (RQ2), and their progress along the way (RQ3).

## 2 RELATED WORKS

### 2.1 Scaffolding in Programming

Scaffolding in education was initially defined by Wood, Bruner, and Ross, with emphasis on how teachers and tutors help students learn by providing support as they acquire skills and master tasks [69]. Scaffolding can trace at least some of its roots to Vygotsky's zone of proximal development, which states that there is a space between what a student knows and what they can possibly learn, which may be supported externally through aid [9]. In recent years, redefinitions of scaffolding by Holton and Clarke have emphasized that scaffolding is an "act of teaching that i) supports the immediate construction of knowledge by the learner; and (ii) provides the basis for the future independent learning of the individual" [29]; this situates metacognition within the scaffolding framework, and more strongly connects learning theories like the zone of proximal development to the scaffolding process. Scaffolding has been shown to improve learning outcomes at the application level but has less impact on the principles level [3, 48]. In other words, scaffolding may not improve how much a student learns but rather how well they can carry out learned tasks.

Programming is challenging for many learners [22, 30] for various reasons, such as the high distance between theory and practice and the intense cognitive load [30]. As a result, scaffolding can be very useful for supporting programming learners [44]. A myriad of papers have mentioned the impacts of scaffolding on reducing cognitive load and enhancing engagement [6, 35, 36], which have been connected to improved learning outcomes [26, 37]. Some research also suggests that creating scaffolding can be challenging for teachers. Simons, Klein, and Brush found in a case study of a sixth-grade teacher that creating resources for problem-based learning was frustrating [60]; Ertmer and Simons followed this up with a study on how teachers themselves can be scaffolded with regard to implementing problem-based learning in their classrooms [21]. It seems reasonable, then, that providing existing scaffolding systems that have been assessed for efficacy can be beneficial for instructors.

Prior research has shown that there are both benefits and challenges in scaffolding for programming contexts specifically. Basu et al. identified several areas where scaffolding could improve learning in programming, including domain knowledge, modularity, and even reusing code [2]. Scaffolding can be beneficial in improving pair-programming [71], self-regulation [40], and metacognition [61], the latter two of which constitute cognitive control, which has been found to be significantly tied to improved learning outcomes [52]. It has also been tied to students spending more time on programming, as they then spend less time setting up to program [34]. In a BBP (block-based programming) game context, Tikva and Tambouris found that giving hints to students who were programming resulted in improved learning outcomes for the scaffolded population [64]. Zhi et al. created a scaffolding system to provide scaffolded self-explanation prompts and found that while the system did not impact students' learning, it did reduce students' intrinsic cognitive load and slightly reduce the amount of time they had to spend on tasks [73]. Ultimately, scaffolding can play a critical role in programming contexts by reducing the difficulties students face and improving their learning outcomes.

### 2.2 Parsons Problems for Scaffolding

Scaffolding the programming process appears in many different forms. Lytle et al. define several categories of scaffolding in BBP, including worked examples, incomplete code, buggy code, and Parsons problems; their findings suggest that scaffolding coding activities can be particularly helpful for novice programmers [42]. Parsons problems, also known as Parsons puzzles, were first introduced by Parsons and Haden in 2006 [51]. Their idea, which was employed by many instructors and researchers afterward, was to present students with a set of code fragments, which students had to choose from and arrange to build the final solution.

Ericson et al., in a systematic literature review, note that Parsons problems, by exposing students to worked examples (i.e., an

expert's solution to a problem [8]), reduce cognitive load for students, allowing for students (especially novices) to learn with less effort [17]. A study by Ericson et al. in 2017 confirms that solving Parsons problems takes less time than other code-related exercises (such as fixing code errors), while also not negatively impacting students' learning performances or retention of knowledge [19]. Du, Luxton-Reilly, and Denny identified in another systematic literature review that Parsons problems, in general, have been used to identify student difficulties, provide immediate feedback, improve student engagement, and reduce cognitive load [14]. Hou, Ericson, and Wang found that Parsons problems helped to benefit students with low self-efficacy, making them more likely to complete a given task than if provided no scaffolding [31]. Parsons problems have also been correlated with improved code writing scores [10].

However, Parsons problems may not be the end-all of learning programming; for instance, some evidence points to Parsons problems potentially being solved by students using syntactic heuristics, allowing them to solve the problems without actually understanding the solutions [68]. Some, including Ericson et al. in 2023, have called for an increase in replication studies to corroborate existing findings and create better variations of Parsons problems to address learning concerns [20]. We note here that most research on Parsons problems has been conducted at the post-secondary level and with text-based languages, with less research focused on Parsons problems' efficacy at the secondary level or within the context of block-based programming. Zhi et al. studied Parsons problems in the BBP language, Snap*!*, and found that students in a CS0 course (i.e., novices) spent half as much time to complete a Parsons problem than an equivalent programming problem, while also demonstrating similar performance on subsequent assignments [72]; in other words, Parsons problems were seemingly effective in a BBP context. Harms, Chen, and Kelleher found that, for students aged 10-15 working in a BBPE, distractors in Parsons problems (i.e., extra unrelated statements) increased cognitive load, which decreased students' completion rates and increased the time they spent on the tasks [27].

## 2.3 Analysis of Trace Log Data

Log analysis involves examining trace log data (i.e., records of sequences of events or actions within a system) [1]. Previous studies have utilized log analysis in order to gain insights into programming behaviors and processes, like predicting student performance [23, 47, 67], potential dropouts [49], student self-perception [24], achievement goals [7], and novices' programming strategies [39, 66, 70]. In one study to understand novice programming behaviors, Dong et al. defined and classified students' tinkering behaviors in block-based programming assignments by analyzing students' coding traces to determine patterns representing different forms of tinkering [12]. Kong and Pollock [38] have created a tool that logs students' interactions in the programming environment of the BBP language Scratch. They analyzed students' logs both manually and using sequential pattern mining algorithms to identify students' tinkering behaviors and different characteristics of students' final code.

The analysis of trace logs can be beneficial in assessing the effects of scaffolding interventions, particularly when triangulated with other quantitative and qualitative techniques. Dever et al. used log files to understand the impact of scaffolding (as done through a game-based learning environment) on self-regulated learning [11]. Jennings and Muldner extracted data from log files (e.g., number of attempts per problem, problem time, etc.) in order to assess the impacts of a new scaffolding system for students learning how to code trace [32]. Siadaty, Gasevic, and Hatala found that a trace-based protocol was beneficial in assessing the effects of scaffolding on self-regulated learning processes [58]. Trace log analysis has also been used in some studies assessing Parsons problems; Ericson, McCall, and Cunningham used log file data to demonstrate that adaptive Parsons problems were more likely to be correctly solved than non-adaptive Parsons problems [15], although the log analysis in this study was relegated to assessing the final result of programming and not the entire programming process (i.e., whether the solution was correct, rather than any elements of the process of programming). These studies suggest that log data, either alone or triangulated against other sources of data like pre- and post-intervention tests, can be an effective source of data for quantitatively evaluating the impacts and effectiveness of scaffolding.

The use of this data requires an environment that actually *collects* this data. For this paper and its context, we focus here on BBP environments. As mentioned earlier, Kong and Pollock developed such a tool for Scratch, but this is not the only BBP language that can have its trace logs evaluated. Price et al. introduced iSnap, an extension to the Snap*!* programming environment that, among other features, collects programming logs [53]; this includes records of a person running their code or dragging out a block. Dong et al. proposed a set of data-driven detectors that, when provided with logs of students' programs, could automatically define thresholds for progress and struggle during programming [13]. In this study, they evaluated these detectors via expert review, comparing when the detectors noted a period of struggle against when an expert would intervene. Tabarsi et al. further evaluated this system by comparing the defined struggle and progress moments against novice programmers' think-aloud data [62]. They found that the detectors had a high accuracy rate in predicting progression moments. We consequently conclude that trace logs can be used to understand students' programming behaviors and performance (for example, by quantifying students' progress so as to compare the results of interventions), as we do below.

# 3 METHODOLOGY

In this section, we first introduce the participants and the context in which the study was conducted. Next, we outline the steps taken to prepare our data for analysis. Subsequently, we elaborate on the use of SPD to identify moments of struggle and progress. Finally, we delve into the data analysis phase, discussing the various metrics employed.

## 3.1 Participants and Context

For this study, we analyzed the programming log data of 199 students enrolled in a middle school in the southeastern United States.[1] The students programmed in a BBPE called Cellular [33]. Cellular extends the capabilities of Snap*!* (which itself is an extension of

[1]The study was covered by IRB 18022 at NC State University.

the programming language Scratch [28]) for modeling agent-based scenarios within a grid of cells [42]. This data was collected on the last day of a four-day computational thinking lesson, "Food Webs," within five sixth-grade science classes.

Given that participant demographics were not gathered when the study was conducted, we present the demographic profile of the school in Table 1. This is based on the assumption that the classroom demographics mirror those of the school.

**Table 1: Demographic Profile of the School Population for the 2019-2020 Academic Year (The Basis for Our Sample Population)**

| Demographic Category | | Count | Percentage |
|---|---|---|---|
| Gender | Female | 371 | 44.4% |
| | Male | 464 | 55.6% |
| Race | American Indian | 0 | 0.0% |
| | Asian | 52 | 6.2% |
| | Black | 173 | 20.7% |
| | Hispanic | 188 | 22.5 % |
| | Pacific Islander | 1 | 0.1% |
| | More than One Race | 41 | 4.9 % |
| | White | 380 | 45.5% |

The activity followed a multi-day paradigm in which students worked with and developed code in the BBPE. On the first day, students learned Food Webs terminology and components and practiced writing pseudocode for Food Webs agents' behaviors, such as eating and moving. On the second day, students were given code that already programmed sunlight's effect on the "Plant" agent as a producer in Cellular. The third day shifted focus to coding the "Bunny" agent as a primary consumer. The final day culminated with students coding the "Fox" agent as a secondary consumer and experimenting with altering their code to observe the changes in the simulation. Our study focuses on the fourth day, as it had a uniform pedagogical structure for all students.

This study employed a between-subjects, controlled quasi-experimental design. It was conducted across five classes on the same day, categorizing them into experimental (three classes) and control (two classes) groups. Specifically, the experimental classes were presented with Parsons problems, and the control classes proceeded without this intervention. No demographic data was used in creating the splits in order to ensure balance.

As shown in Figure 2, all students were given a starter code for the Fox agent to place the agent on the screen and center it within a particular cell. The students in the control group did not receive any further scaffolding for the Fox implementation. The students in the experimental group were given all the blocks they needed to build the Fox agent code to accomplish the assigned tasks, a technique often referred to as a Parsons Problem, as discussed in Section 2.2. The final Fox agent code solution for both groups is given in Figure 1. The only difference was in the scaffolding they *did* (experimental) or *did not* (control) receive in the starter code.

As illustrated in Figure 1, students were asked to start with making a local variable named "Energy" for tracking the energy of the Fox agent. Subsequently, students had to add a move block inside a forever loop to mimic the movement of a Fox roaming through a biome. They were also supposed to decrease the Fox's energy after each move to replicate energy loss. Finally, they were supposed to implement three conditions: 1) they had to define a condition for times when the Fox energy was low, and the fox would then eat an animal to regain energy; 2) the Fox was supposed to die if the energy value became negative; and 3) after a certain amount of time had passed, the Fox would 'reproduce.' The abstraction of energy and the translation of its flow into an algorithm are both computational thinking concepts. Of 199 participants, we had 106 working on the experimental assignment and 93 on the control. The starter code for both groups and the difference between them is highlighted in Figure 2.

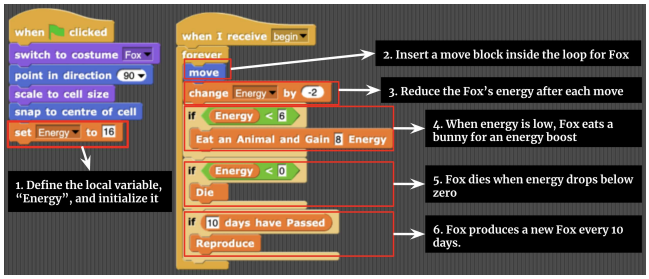



**Figure 1: Fox's Final Code with Assigned Tasks for Students**

## 3.2 Data Preparation

Cellular utilizes the iSnap trace log system, developed by Price et al. [53], to log student activities during block-based programming. The purpose of this trace log system was to be able to record the actions students took within the programming environment and to be able to use the trace log to create a series of snapshots about student code and generate hints for future students writing code to solve the same programming problem. As part of this process, a similarity metric was developed to compare student code to existing previous correct student solutions, and that was then extended by Dong et al. to create the SPD [13], which we discuss in more detail in section 3.3.

Following initial data cleaning steps on the log data, we extracted each student's last code entry due to the absence of separate student submissions. Afterward, we made minor adjustments to make the Cellular submissions compatible with SPD. Finally, one of the researchers defined the rubric given in Table 2 and scored all students' programs. The rubric items correspond to the correct construction of a program. Overall, the rubric is a performance metric that measures the extent to which students were able to implement the algorithm for energy flow into their own block-based programs. Consequently, we primarily measured computational thinking outcomes. When the scoring was completed, we combined the log data and corresponding scores into a dataset.

## 3.3 SPD

All students completed their activity in Cellular, which logs student interactions (e.g., creating a project, adding a sprite, or clicking on the run button). The rubric scores and log data were input into

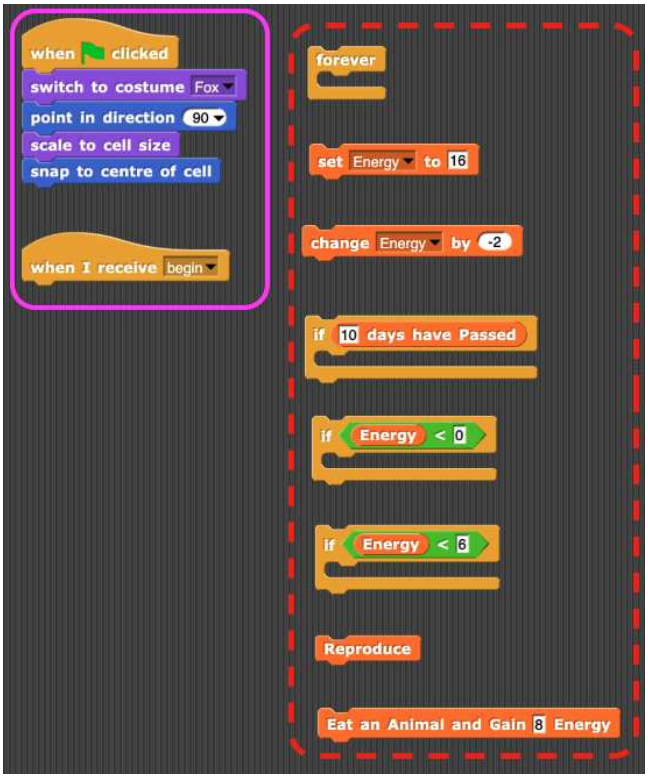



**Figure 2: Starter Code for Fox Agent: The pink box (left) represents the starter code for all students, and the red dashed box (right) features the Parsons Problem exclusive to the experimental group.**

**Table 2: Rubric Items for Grading the Activity**

| # | Rubric Items |
|---|---|
| 1 | Move Block is inside forever |
| 2 | Energy value is set outside Loop |
| 3 | Energy value is decreased inside Loop |
| 4 | If Energy < Positive Eat Animal and Gain Positive Energy |
| 5 | Code related to rubric #4 is in correct location |
| 6 | If Energy < 0 Die |
| 7 | Code related to rubric #6 is in correct location |
| 8 | If N Days have Passed, Fox agent should Reproduce |
| 9 | Code related to rubric #8 is in the correct location |

the SPD[2]. SPD offers several options, including SourceCheck [54], Similarity Score, and tree edit distance [74], to compare student log data with correct, final solutions and measure their progress.

A key novelty of SPD is its adaptation of SourceCheck's mapping cost to the similarity score to find the closest solution and break students' coding time into struggle or progress periods. To achieve this, SPD measures how the proximity of the student's code to the nearest correct submission improves over time. This is important for understanding a student's coding path, as it allows for quantifying the proximity to correctness after each code edit. However, our log data was not fully compatible with SPD's original configuration and similarity score due to the differences between Snap*!* and Cellular.

We examined several metrics and found tree edit distance effective in finding the closest solution for Cellular code. Tree edit distance is a measure of the minimum number of node insertions, deletions, and relabelings required to transform one tree into the other [63, 74]. Inspired by Dong et al. definition of absolute progress [13], we define a metric named "minimum distance" $m(t)$, which quantifies the shortest distance attained from the closest correct solution up to any given point. Given $d(t)$ as measured tree edit distance at each logged action time $t$, $m(t)$ is computed as

$$m(t) = \min(m(t-1), d(t)) \tag{1}$$

where $d(1) = m(1)$, serving as the baseline distance for the subsequent comparisons. For ease of reference, we will refer to this metric as the "SPD score" throughout the remainder of the paper. Also, while the specific value is not of importance in this study, it is essential to know a lower number indicates less distance from a correct solution within this SPD configuration.

SPD can leverage correct code implementations from both students' solutions and instructor-crafted examples. However, the use of a large quantity of student-derived work is preferable, as it offers a closer representation of the diverse strategies students might employ. Since we did not have access to implementations from previous students, we evaluated the final state of our current students' code to measure their progress.

## 3.4 Data Analysis

In the initial phase of our analysis, we used the SPD score in each student's log data to determine whether they were working on the Parsons Problem or not. Then, we defined a list of per-student metrics to gain insight into each student's coding path, enabling us to make comparisons as follows:

- *Score.* The rubric-based score for each student.
- *Active time.* Duration of a student's coding time, excluding periods of inactivity lasting more than one minute. For periods of inactivity longer than one minute, only the first 60 seconds are counted as active time.
- *Idle time.* The sum of a student's idle durations exceeding one minute but excluding gaps of more than 5 minutes.
- *Number of runs.* The number of times the student ran their program.
- *Number of consecutive runs.* The number of times the student ran their program consecutively two times or more without any action in between.
- *Number of category changes.* The number of instances the student switched between block categories.
- *Number of consecutive category changes.* The frequency of the two or more consecutive block category switches by a student without any action in between.
- *SPD Score.* A metric indicating the minimum code tree edit distance between the student's code and the closest correct solution [13, 54], at each code log generation.

[2]SPD was originally developed by Dong et al. [13] to be used as an indicator for when students programming in Snap*!* might need an intervention based on their progress.

We conducted the Shapiro-Wilk test to assess the normality of the distribution of all metrics for both experimental and control groups independently. All but one of the metrics were non-normal in both conditions, while the number of runs was normal for just one. Consequently, the Mann-Whitney U test, suitable for analyzing non-normal distributions, was applied across all comparisons to determine the significance of differences between the study's two independent groups. We ran this test with a two-tailed approach to identify any significant differences, irrespective of direction. Furthermore, since we conducted a number of tests, we performed Benjamini-Hochberg correction [4] with a significance level ($\alpha$) of 0.05 for tests of significance. The results of comparisons are presented in Table 3.

We demonstrated all metrics (except for score and time) at four distinct phases throughout each student's active programming period: 1) at the first quartile, 2) at the midpoint, 3) at the third quartile, and 4) upon completion. By presenting each metric by time quartiles, we aimed to characterize students' coding behavior (what they did within each group) over the course of their programming. To avoid inflating the risk of false discoveries from excessive comparisons, we did not test the statistical significance of each metric across time quartiles for all metrics. However, we made an exception for the SPD score, as we believed that verifying the trend of students' progress was essential for assessing the effectiveness of the Parsons problem throughout the programming time.

## 4 RESULTS

This section describes the results corresponding to metrics specified in Section 3.4. Table 3 provides a comparison of each metric for the experimental and control groups by the average, median, and the related research question.

**Active Time.** The Mann-Whitney U test revealed that active time differences were statistically significant in the two groups. ($U = 6402.5$, $p = 5.12e\text{-}4$). As demonstrated in Table 3, the control group spent a median of 12.72 minutes, and an average of 12.47 minutes of active time, while the control group spent a median of 13.92 and an average of 13.75 minutes. These results suggest that the Parsons Problem significantly affects active programming time among students.

**Idle Time.** The difference between the two groups' distribution of idle times was not statistically significant ($U = 4302.5$, $p = 0.12$). The experimental group had an average idle time of 2.52 and a median of 1.98 minutes, while the control group had an average of 2.03 and a median of 1.67 minutes, which shows a minimal difference in students' tendency to disengage among the two groups.

**Number of Runs.** The Mann-Whitney U test results suggest that the difference between the experimental and control groups was significant ($U = 3292.5$, $p = 1.14e\text{-}4$). According to the data presented in Table 3, the experimental group had an average of 7.92 runs and a median of 8 runs, while the control group had an average of 5.6 and a median of 5. Further investigation of the frequency at which students ran their code across each programming time quartile, as shown in Figure 3, indicated nuanced differences between the coding paths of the two groups.

In the initial three quartiles, the experimental group had a higher frequency of runs compared to the control group, which may indicate more testing on the experimental group side. However, this trend reverses in the final quartile, where the experimental group's frequency of code execution diminishes, falling below that of the control group.

The control group demonstrated a persistent increase in the number of runs from the first through the last quartiles, with a steep rise from the second to the third quartile. This surge contrasts with the experimental group's most notable increase, which occurred earlier, from the first to the second quartile. These findings outline a divergent trajectory in the programming behaviors of the two groups.

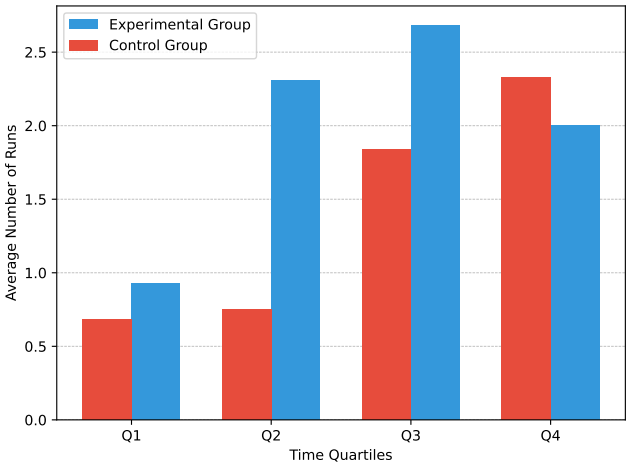



**Figure 3: Average Number of Runs Across Time Quartiles by Group**

**Number of Consecutive Runs.** Our test demonstrated the significance of the difference between the experimental and control groups for this metric ($U = 4111$, $p = 0.03$). The experimental group ran their code two times or more, with an average of 1.05, while the control group did so with an average of 0.71 times. The medians for the two groups were also distinct: 1 for the experimental group and 0 for the control group. A closer examination of consecutive runs within each quartile of programming time in Figure 4 reinforces the results associated with the "Number of Runs" metric.

While the overall incidence of consecutive runs was inherently lower compared to the number of runs, the trend almost mirrors that of the previously discussed metric with one notable exception. In the second quartile, the average number of consecutive runs demonstrated by the control group decreased. This difference contrasts with the control group's consistent rise observed in the number of runs, which also resulted in a greater disparity between the two groups during this quartile.

**Number of Category Changes.** The difference between the number of category changes in the experimental group and the control group was statistically significant ($U = 8984$, $p = 1.41e\text{-}22$). As shown in Table 3, the average number of category changes in the experimental group, 6.9, was much lower than in the control group,

| Metric | Group | Mean | Median | Mann-Whitney U | Corrected P-Value |
|---|---|---|---|---|---|
| *Research Question 1: Students' Programming Behaviors With and Without Scaffolding through Parsons Problems* | | | | | |
| Active Time (minutes) | E | 12.47 | 12.72 | 6402.5 | **5.12e-4** |
| | C | 13.75 | 13.92 | | |
| Idle Time (minutes) | E | 2.52 | 1.98 | 4302.5 | 0.12 |
| | C | 2.03 | 1.67 | | |
| # of Runs | E | 7.92 | 8.0 | 3292.5 | **1.14e-4** |
| | C | 5.6 | 5.0 | | |
| # of Consecutive Runs | E | 1.05 | 1.0 | 4111 | **0.03** |
| | C | 0.71 | 0 | | |
| # of Category Changes | E | 6.9 | 4 | 8984 | **1.41e-22** |
| | C | 24.27 | 22 | | |
| # of Consecutive Category Changes | E | 1.19 | 1.0 | 8363 | **4.01e-17** |
| | C | 3.88 | 4 | | |
| *Research Question 2: Parsons Problems' Impact on Students' Scores* | | | | | |
| Score | E | 78% | 89% | 3791.5 | **5.5e-3** |
| | C | 64% | 78% | | |

**Table 3: Summary of Descriptive Statistics and Statistical Test for Metrics by Group**

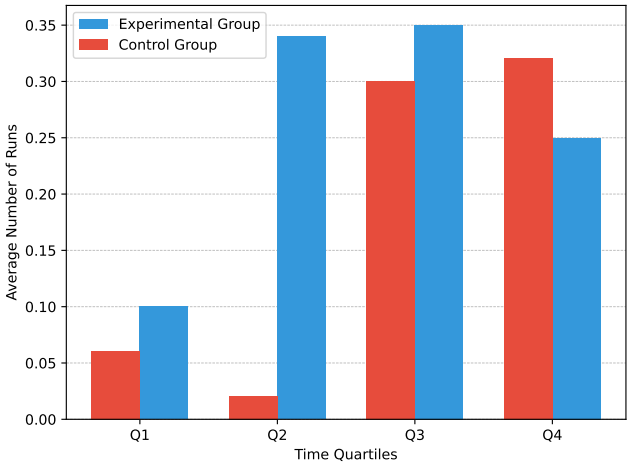



**Figure 4: Average Number of Consecutive Runs Across Time Quartiles by Group**

24.27. This difference is further reflected in the median values, where the experimental group had a median of 4 category changes, contrasting sharply with the median of 22 in the control group.

To further analyze this substantial difference, we drew the box plot of category change, illustrated in Figure 5(a). This plot demonstrates a noticeably wider spread of clicking on category tabs within the control group, as evidenced by the interquartile range (IQR) of 16, which suggests a broader variance in novices' patterns of changing block categories. Contrarily, the experimental group exhibited a narrow spread (*IQR = 7.0*), suggesting a more homogeneous pattern of category changing among novices on the Parsons problem.

Looking into the diagram of the average number of category changes within each time quartile in Figure 5(b) reveals different patterns among the two groups. While the numbers in the control group are constantly at least two times higher across all quartiles, their busiest period seems to be the second quartile, which is followed by two lower values in the subsequent quartiles. In contrast, the experimental group exhibits a decrease in this behavior from the beginning to the end.

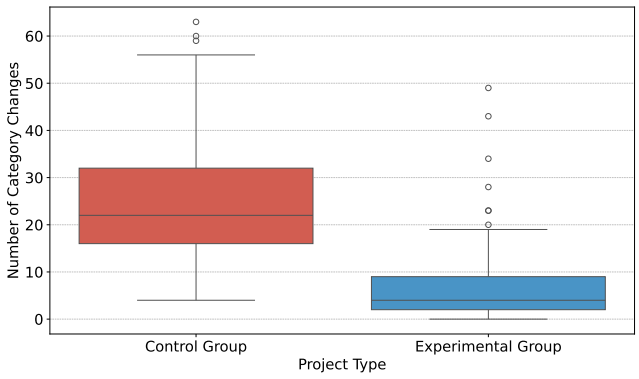
In conjunction with the results associated with the number of runs, we can see that while running seems to be the predominant action of the experimental group from the beginning through the third quartile, the control group is less engaged with running in the first two quartiles and is most changing block categories.

**Number of Consecutive Category Changes.** The distribution of consecutive category changes in the experimental group had a statistically significant *(U = 8363, p = 4.01e-17)* difference from the control group. Table 3 demonstrates that the average number of two or more consecutive block category changes without intermediate actions in the experimental group, 1.19, was lower than that of the control group, 3.88, although the contrast is not as stark as the previous metric. The median of 1 in the experimental group compared to 4 in the control group further confirms students' fewer switches between categories.

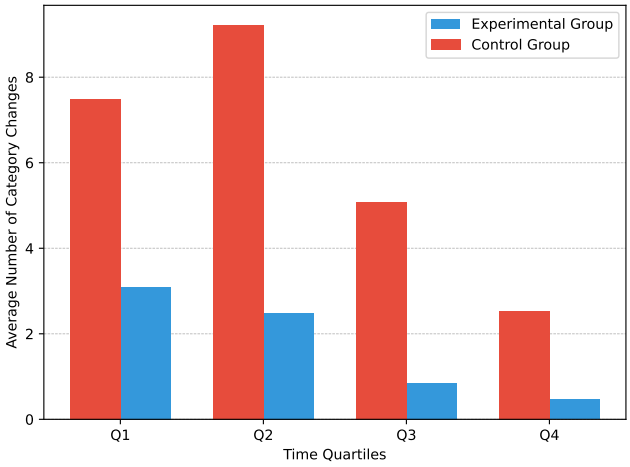
As illustrated in Figure 6, novices' pattern of consecutive category changing across each programming time quartile is not much different from the trend of category switching. This similarity corroborates the observed differences in category-changing behavior between the groups.

**Score.** Our test results indicated a significant difference in the distributions of the two groups (*U = 3791.5, p = 5.5e-3*). The difference is reflected in a higher average score of 78% in the experimental group compared to 64% in the control group. Moreover, the median score in the experimental group was 89% as opposed to 78% in the control group, further indicating the two groups' differences.

The score distribution of students in experimental and control projects is visualized in Figure 7. Scores are divided into five bins on the x-axis, and the percentage of students within each project type is depicted on the y-axis. In the first four bins, the number of students in the experimental group is lower than their other counterparts. However, the trend reverses in the final score bin,




(a)




(b)

**Figure 5: Distribution of Category Changes based on Project Type**

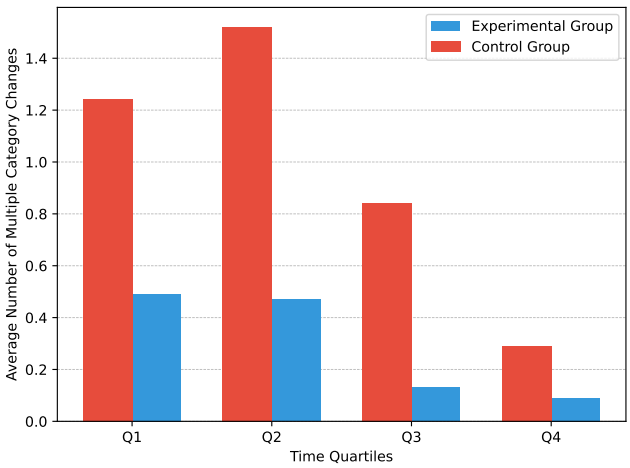



**Figure 6: Average Number of Consecutive Category Changes Across Time Quartiles by Group**

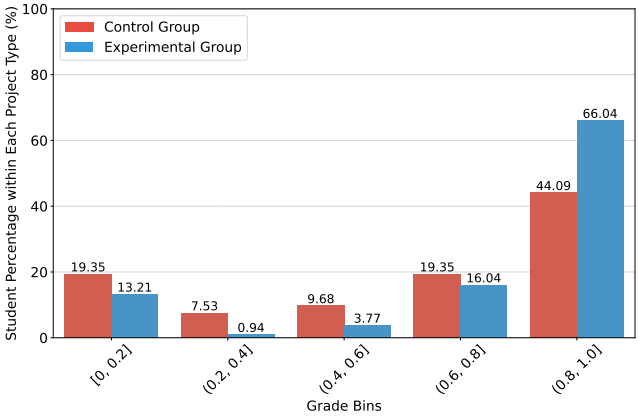



**Figure 7: Score Distribution of Students in Experimental and Control Groups Across Score Bins.**

where over 66% of the experimental group scored 80-100%, but only 44.1% of the control group scored in this range.

**SPD Score**. We examined students' distance from a correct solution at four quartiles to see if the differences between the experimental and control groups were significant. The Mann-Whitney U test indicates that the differences were statistically significant in all four quartiles of time. The results of these tests are demonstrated in Table 4.

To effectively illustrate students' progress in their programming session, Figure 8 displays a graph plotting the average SPD score of students across eight points of their active coding. These intervals include the four quartiles of active coding time (as highlighted by vertical lines) and four midpoints that lie in the middle of each quartile.

Starting the analysis from the first quartile, as opposed to the onset of the programming activity, effectively excludes the initial decline in distance score due to importing code. To ensure a thorough understanding, the initial distance score of the control group after importing the starter code stood at 28, in contrast to 24 for the experimental group. As illustrated in Figure 8, the four-unit gap between the groups grows until the second quartile and then diminishes, ultimately resulting in the narrowest gap by the end of the programming period. The steeper decline in the SPD score of experimental group scores during the first two quartiles implies the positive impact of the Parsons Problem on students' progress toward a valid solution.

## 5 DISCUSSION

### 5.1 RQ1: Students' Programming Behaviors With and Without Scaffolding through Parsons Problems

As outlined in Table 3, we used six metrics to compare the coding behaviors of students engaged with the experimental activity compared to the control activity. Our analysis revealed that students in the experimental group spent less time while also achieving better scores on the activity. This suggests that students who received

| Metric | Group | Mean | Median | Mann-Whitney U | Corrected P-Value |
|---|---|---|---|---|---|
| SPD Score in 1st Time Quartile | E | 14.07 | 15.5 | 7810 | **3.51e-12** |
| | C | 22.39 | 25 | | |
| SPD Score in 2nd Time Quartile | E | 8.42 | 7 | 6926.5 | **2.19e-06** |
| | C | 14.83 | 15 | | |
| SPD Score in 3rd Time Quartile | E | 6.58 | 5 | 5971.5 | **0.01** |
| | C | 10.04 | 8 | | |
| SPD Score in 4th Time Quartile | E | 4.74 | 0 | 5841 | **0.02** |
| | C | 7.44 | 4 | | |

**Table 4: Summary of Descriptive Statistics and Mann-Whitney U Test Results for SPD Score in Different Quartiles of Active Programming Time by Group**

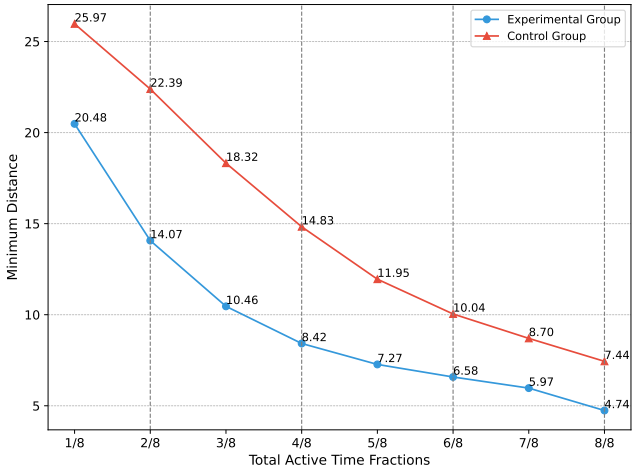



**Figure 8: The Average of Students' SPD Score throughout Active Coding Time in Experimental and Control Groups.**

scaffolding through Parsons problems had a more efficient programming experience. Idle time, however, was consistent across groups, averaging 2 to 3 minutes within the session, suggesting that both groups had similar levels of engagement with the activity.

The experimental group demonstrated a notable increase in the number of runs, executing their code approximately 41% more times on average compared to their control counterparts. Upon examining traces from several students, we propose two hypotheses for the increased number of runs. First, students in the experimental group could construct their code by actively experimenting and trial-and-error. Additionally, since this group's active time was lower and they progressed faster toward a solution, they had more chances to play with the code, understand the scientific concept they were learning, and check how code changes impacted execution. As an example, we observed some students changing the number of agents, a modification relevant to those curious about the dynamics of equilibrium in scenarios with an imbalance between predators (fox) and prey (bunny). Conversely, the other group, as previously indicated by their slower progress to the final solution in Figure 8, was more engaged with the implementation process, which might also describe the reason behind their significantly higher frequency of category changes.

The results from the consecutive runs metric revealed that this behavior was significantly more common among students in the experimental group. A study by Dong et al. linked frequent code execution without intermediary actions to tinkering behaviors, which can sometimes be a sign of frustration (e.g., as a debugging behavior) and sometimes be productive (e.g., running the code and attempting to understand what it is doing) depending on the context [12]. While we lacked a measure to assess frustration or productivity, we believe a potential reason behind the higher number of consecutive runs in the experimental group was that Parson's problems had more code present in the coding environment from the start, so the experimental group could run the code to build an understanding of what each part of the code does. On the other hand, this also implies that the control group had to spend more time finding blocks and constructing the code (notably in the first half of their coding time), so they could not spend time early on running the program to understand the impact of the code blocks and how they are ordered.

Our analysis also demonstrated significant differences in the average number of both category-changing and consecutive category-changing between the two groups. Prior research has referred to category switching as a highly frequent search-related action in BBPEs [57], and often as an indicator of struggle [56]. This is in line with other learning outcomes perceived in the control group and also suggests that Parsons problems can serve as a beneficial tool in reducing the cognitive strain of searching for blocks.

## 5.2 RQ2: Parsons Problems' Impact on Students' Scores

In the context of a science classroom where the primary goal is fostering computational thinking, rather than teaching the particulars of the programming language, having students spend more time re-ordering and running code is more relevant to their learning than having them find where the code is located. Although a longitudinal study is needed to determine the long-term impacts, our results suggest that scaffolding via Parsons problems can help students achieve higher scores. This finding, in conjunction with faster progress toward a correct solution and spending less time switching through code categories, corroborates prior research indicating that Parsons problems effectively reduce cognitive load by lowering the load on student working memory, thereby improving learning and performance [18, 46].

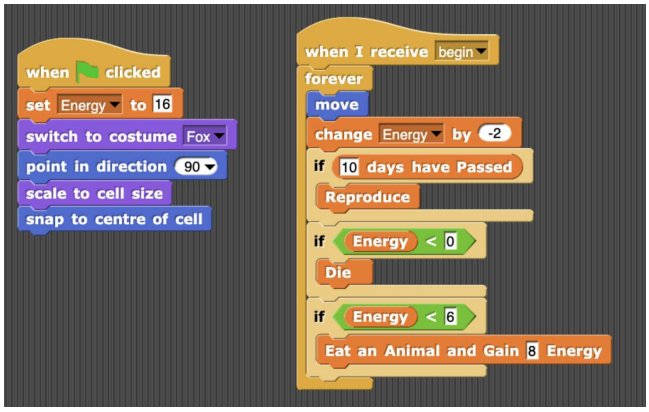
### 5.3 RQ3: Utilizing an Automatic Progress Detector to Assess Parsons Problems Impact on Programming Performance

In studies like ours, where final submissions are not collected, or students may diverge from the main task to explore within the programming environment, automated measures like code distance are highly practical. They enable an understanding of how students' closest approach to a correct solution evolves throughout their programming session. However, relying upon them needs an understanding of how they actually measure students' work. To delve deeper into this, we conducted a detailed analysis of trace logs from two arbitrarily selected students: one from the experimental group, referred to as "Jill," and another from the control group, referred to as "Bob".

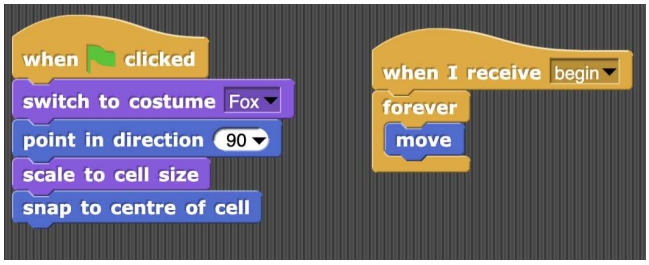
Jill and Bob's coding area after importing the starter code resembled the code shown in Figure 2. Jill received all of the shown blocks, and Bob received the code inside the pink box on the left. Following the import, a notable decrease in their SPD scores was observed as expected, declining from 397 to 24 for Jill and to 28 for Bob. Starting from the first seconds of programming, we observed Jill accurately positioning and snapping the blocks, which led to a reduction in her SPD score. Notably, her score dropped to two within just 66 seconds of beginning the task. As demonstrated in Figure 9(a), her code was fully assembled at this point. The persistence of a 2-unit score, as opposed to zero, may be attributed to the order of blocks, which does not impact the code's functionality and was not considered in our grading rubrics.

Figure 9(b) displays Bob's code at the same time, around 66 seconds into the task, which shows a stark contrast to Jill's progress at the same interval. Although complete code was not expected from Bob at this point, the comparison underscores their completely different experiences, even in the beginning.

Throughout the session, Jill remained engaged with the task, primarily experimenting with blocks, modifying inputs, and executing the code, which could improve not only her programming skills but also her grasp of the scientific concepts underpinning the assignment. This is visualized in Figure 10(a), where roughly six minutes into coding, she had added "Flower" to eat the block and reduced its "if" condition value to 6. On the other side, at a similar point in time, Bob is still focused on placing blocks and editing inputs, as illustrated in 10(b). While Bob's minimum distance gradually decreased, eventually, he reached a distance of 4 after 9.5 minutes, which was a correct implementation upon review. Overall, as we also observed in Figure 8, there is a remarkably sharper reduction in the SPD scores for the experimental group compared to the control group during the initial half of the coding period. While a sharp decrease in SPD scores does not necessarily equate to code correctness, it does indicate that the experimental group is contributing code that is more likely to be correct than students in the control group.




(a)




(b)

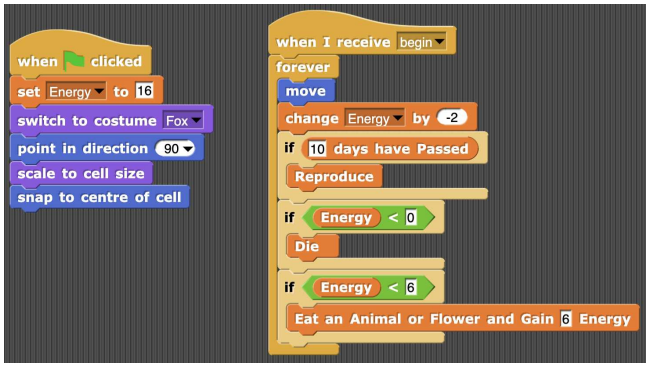
**Figure 9: Jill's Code (a student from the experimental group) and Bob's Code (a student from the control group) Approximately One Minute After Starting to Code**

## 6 LIMITATIONS AND FUTURE WORK

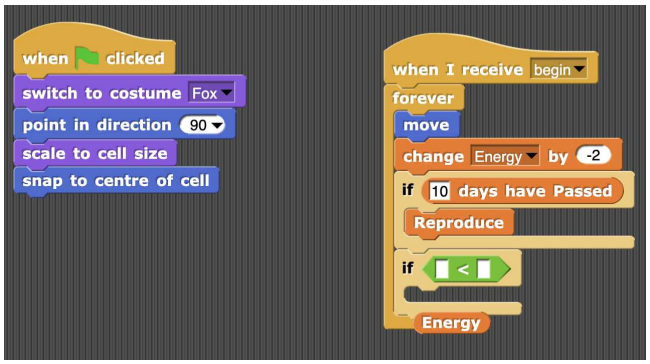
Here, we identify two primary limitations in our study. First, we relied on extracting students' final code states for grading due to the lack of submissions. This approach was not optimal, as many students completed the activity and then modified their code to explore other ideas in the programming environment. We considered extracting code at the point of the lowest SPD score. However, it was not feasible to objectively differentiate between students who restarted their work after completion and those who did so out of confusion. Second, we did not have the specific demographics of our participants. Although we attempted to mitigate this by including the school demographics profile in Table 1, we acknowledge that these statistics may not accurately represent the demographics of our study's population.

As for future work, we recommend that further research replicate our findings with considerations for students' demographics and ensure random design. As classes were assigned to groups rather than individual students, the teacher/classroom environment may have had some influence on students' outcomes. Moreover, while we recognize the inherent challenges in developing a flawless code proximity detector, we see designing a Cellular-specific detector as an intriguing idea worth exploring in future studies. Future work may also focus on the role of programming scaffolding for other non-CS classroom contexts, such as English or Social Studies.

The use of SPD-type analyses of trace log data may be used to help visualize students' progress in systems meant to provide progress monitoring for teachers through dashboards or for students through progress monitors, as in the Adaptive Immediate




(a)




(b)

**Figure 10: Jill's Code (a student from the experimental group) and Bob's Code (a student from the control group) Approximately Six Minutes After Starting to Code**

Feedback (AIF) system designed by Marwan et al. [43]. SPD progress visualizations could allow teachers to see the status of the whole class without the need for precise auto-graders, student submissions of code, or unit tests. Similar metrics using trace log analysis may help teachers more quickly assess programming assignments if precise grades are not needed, and researchers can use them in the assessment of pedagogical interventions.

## 7 CONCLUSION

In this study, we investigated the impact of Parsons problems on students' coding behaviors, their final scores in a programming activity, and their progress toward a correct solution. We sought to understand the impact of this scaffolding strategy by analyzing the coding log data of 199 sixth-grade students and employing an automated progress detection (SPD) model. Our analysis indicated that engaging with Parsons problems improved learning outcomes for these students.

Students working with Parsons problems spent less time on their code and achieved relatively higher scores compared to their peers. Moreover, we noted that Parsons problems effectively reduced the need for students to alternate between different categories in search of appropriate code blocks. The reduction in search time, in conjunction with this group's higher number of code runs, implies that this group had additional opportunities to delve deeper into the assignment and explore how the code related to its execution. This aspect is particularly important in integrated classrooms, where the purpose of programming is to enhance the learning of the main course topic while introducing important computational thinking concepts.

Furthermore, we demonstrated a novel methodology to assess and demonstrate the impacts of our intervention by utilizing the data-driven SPD. By leveraging trace log analysis in this capacity, we can reach a better understanding of students' learning processes and behaviors *throughout* programming, allowing researchers to better pinpoint where interventions are most effective and how they interact with learning schema.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 18022.

## REFERENCES

[1] David Basin, Patrick Schaller, Michael Schläpfer, David Basin, Patrick Schaller, and Michael Schläpfer. 2011. Logging and log analysis. *Applied Information Security: A Hands-on Approach* (2011), 69–80.
[2] Satabdi Basu, Gautam Biswas, Pratim Sengupta, Amanda Dickes, John S Kinnebrew, and Douglas Clark. 2016. Identifying middle school students' challenges in computational thinking-based science learning. *Research and practice in technology enhanced learning* 11, 1 (2016), 1–35.
[3] Brian R Belland. 2017. *Instructional scaffolding in STEM education: Strategies and efficacy evidence.* Springer Nature.
[4] Yoav Benjamini and Yosef Hochberg. 2000. On the adaptive control of the false discovery rate in multiple testing with independent statistics. *Journal of educational and Behavioral Statistics* 25, 1 (2000), 60–83.
[5] Veronica Cateté, Amy Isvik, and Tiffany Barnes. 2020. Infusing computing: A scaffolding and teacher accessibility analysis of computing lessons designed by novices. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research.* 1–11.
[6] Ching-Huei Chen, Wen-Pi Chan, Kun Huang, and Chin-Wen Liao. 2023. Supporting informal science learning with metacognitive scaffolding and augmented reality: Effects on science knowledge, intrinsic motivation, and cognitive load. *Research in Science & Technological Education* 41, 4 (2023), 1480–1495.
[7] Heeryung Choi, Philip H Winne, Christopher Brooks, Warren Li, and Kerby Shedden. 2023. Logs or self-reports? Misalignment between behavioral trace data and surveys when modeling learner achievement goal orientation. In *LAK23: 13th international learning analytics and knowledge conference.* 11–21.
[8] Ruth C Clark, Frank Nguyen, and John Sweller. 2011. *Efficiency in learning: Evidence-based guidelines to manage cognitive load.* John Wiley & Sons.
[9] Michael Cole, Vera John-Steiner, Sylvia Scribner, and Ellen Souberman. 1978. Mind in society. *Mind in society the development of higher psychological processes. Cambridge, MA: Harvard University Press* (1978).
[10] 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.
[11] Daryn A Dever, Megan D Wiedbusch, Sarah M Romero, Kevin Smith, Milouni Patel, Nathan Sonnenfeld, James Lester, and Roger Azevedo. 2023. Identifying the effects of scaffolding on learners' temporal deployment of self-regulated learning operations during game-based learning using multimodal data. *Frontiers in Psychology* 14 (2023).
[12] Yihuan Dong, Samiha Marwan, Veronica Catete, Thomas Price, and Tiffany Barnes. 2019. Defining Tinkering Behavior in Open-ended Block-based Programming Assignments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19).* Association for Computing Machinery, New York, NY, USA, 1204–1210. https://doi.org/10.1145/3287324.3287437
[13] Yihuan Dong, Samiha Marwan, Preya Shabrina, Thomas Price, and Tiffany Barnes. 2021. Using Student Trace Logs to Determine Meaningful Progress and Struggle during Programming Problem Solving. *International Educational Data Mining Society* (2021). https://eric.ed.gov/?id=ED615663
[14] 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.

[15] 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.

[16] Barbara J. Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S. Miller, Briana B. Morrison, Janice L. Pearce, and Susan H. Rodger. 2022. Parsons Problems and Beyond: Systematic Literature Review and Empirical Study Designs. In *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '22)*. Association for Computing Machinery, New York, NY, USA, 191–234. https://doi.org/10.1145/3571785.3574127

[17] Barbara J Ericson, Paul Denny, James Prather, Rodrigo Duran, Arto Hellas, Juho Leinonen, Craig S Miller, Briana B Morrison, Janice L Pearce, and Susan H Rodger. 2022. Parsons problems and beyond: Systematic literature review and empirical study designs. *Proceedings of the 2022 Working Group Reports on Innovation and Technology in Computer Science Education* (2022), 191–234.

[18] 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*. ACM, Espoo Finland, 60–68. https://doi.org/10.1145/3230977.3231000

[19] 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.

[20] Barbara J Ericson, Janice L Pearce, Susan H Rodger, Andrew Csizmadia, Rita Garcia, Francisco J Gutierrez, Konstantinos Liaskos, Aadarsh Padiyath, Michael James Scott, David H Smith, et al. 2023. Conducting multi-institutional studies of Parsons problems. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2*. 571–572.

[21] Peggy A Ertmer and Krista D Simons. 2005. Scaffolding teachers' efforts to implement problem-based learning. *International Journal of Learning* 12, 4 (2005), 319–328.

[22] Katrina Falkner, Rebecca Vivian, and Nickolas J.G. Falkner. 2014. Identifying computer science self-regulated learning strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education (ITiCSE '14)*. Association for Computing Machinery, New York, NY, USA, 291–296. https://doi.org/10.1145/2591708.2591715

[23] Ge Gao, Samiha Marwan, and Thomas W. Price. 2021. Early Performance Prediction using Interpretable Patterns in Programming Process Data. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. ACM, Virtual Event USA, 342–348. https://doi.org/10.1145/3408877.3432439

[24] Jamie Gorson, Nicholas LaGrassa, Cindy Hsinyu Hu, Elise Lee, Ava Marie Robinson, and Eleanor O'Rourke. 2021. An Approach for Detecting Student Perceptions of the Programming Experience from Interaction Log Data. In *Artificial Intelligence in Education (Lecture Notes in Computer Science)*, Ido Roll, Danielle McNamara, Sergey Sosnovsky, Rose Luckin, and Vania Dimitrova (Eds.). Springer International Publishing, Cham, 150–164. https://doi.org/10.1007/978-3-030-78292-4_13

[25] Shuchi Grover, Roy Pea, and Stephen Cooper. 2016. Factors influencing computer science learning in middle school. In *Proceedings of the 47th ACM technical symposium on computing science education*. 552–557.

[26] Diane S Halm. 2015. The impact of engagement on student learning. *International Journal of Education and Social Science* 2, 2 (2015), 22–33.

[27] 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.

[28] Brian Harvey and Jens Mönig. 2010. Bringing "no ceiling" to scratch: Can one language serve kids and computer scientists. *Proc. Constructionism* (2010), 1–10.

[29] Derek Holton and David Clarke. 2006. Scaffolding and metacognition. *International journal of mathematical education in science and technology* 37, 2 (2006), 127–143.

[30] 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 - Volume 1 (ICER '22, Vol. 1)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/3501385.3543977

[31] Xinying Hou, Barbara Jane Ericson, and Xu Wang. 2023. Understanding the Effects of Using Parsons Problems to Scaffold Code Writing for Students with Varying CS Self-Efficacy Levels. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–12.

[32] Jay Jennings and Kasia Muldner. 2021. When does scaffolding provide too much assistance? A code-tracing tutor investigation. *International Journal of Artificial Intelligence in Education* 31 (2021), 784–819.

[33] Robin Jocius, Deepti Joshi, Yihuan Dong, Richard Robinson, Veronica Cateté, Tiffany Barnes, Jennifer Albert, Ashley Andrews, and Nicholas Lytle. 2020. Code, connect, create: The 3c professional development model to support computational thinking infusion. In *Proceedings of the 51st ACM technical symposium on computer science education*. 971–977.

[34] Jordana Kerr, Mary Chou, Reilly Ellis, and Caitlin Kelleher. 2013. Setting the scene: scaffolding stories to benefit middle school students learning to program. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE, 95–98.

[35] Joo Yeun Kim and Kyu Yon Lim. 2019. Promoting learning in online, ill-structured problem solving: The effects of scaffolding type and metacognition level. *Computers & Education* 138 (2019), 116–129.

[36] Nam Ju Kim, Brian R Belland, and Andrew E Walker. 2018. Effectiveness of computer-based scaffolding in the context of problem-based learning for STEM education: Bayesian meta-analysis. *Educational Psychology Review* 30 (2018), 397–429.

[37] Paul A Kirschner. 2002. Cognitive load theory: Implications of cognitive load theory on the design of learning. , 10 pages.

[38] Minji Kong and Lori Pollock. 2020. Semi-Automatically Mining Students' Common Scratch Programming Behaviors. In *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*. ACM, Koli Finland, 1–7. https://doi.org/10.1145/3428029.3428034

[39] Qianhui Liu and Luc Paquette. 2023. Using submission log data to investigate novice programmers' employment of debugging strategies. In *LAK23: 13th International Learning Analytics and Knowledge Conference*. 637–643.

[40] Dastyni Loksa, Benjamin Xie, Harrison Kwik, and Amy J Ko. 2020. Investigating novices' in situ reflections on their programming process. In *Proceedings of the 51st ACM technical symposium on computer science education*. 149–155.

[41] Nicholas Lytle, Veronica Cateté, Danielle Boulden, Yihuan Dong, Jennifer Houchins, Alexandra Milliken, Amy Isvik, Dolly Bounajim, Eric Wiebe, and Tiffany Barnes. 2019. Use, modify, create: Comparing computational thinking lesson progressions for stem classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 395–401.

[42] Nicholas Lytle, Yihuan Dong, Veronica Cateté, Alex Milliken, Amy Isvik, and Tiffany Barnes. 2019. Position: Scaffolded coding activities afforded by block-based environments. In *2019 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 5–7.

[43] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W Price, and Tiffany Barnes. 2020. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM conference on international computing education research*. 194–203.

[44] Chao Mbogo, Edwin Blake, and Hussein Suleman. 2013. A mobile scaffolding application to support novice learners of computer programming. In *Proceedings of the Sixth International Conference on Information and Communications Technologies and Development: Notes - Volume 2 (ICTD '13)*. Association for Computing Machinery, New York, NY, USA, 84–87. https://doi.org/10.1145/2517899.2517941

[45] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. 2010. Learning computer science concepts with scratch. In *Proceedings of the Sixth international workshop on Computing education research*. 69–76.

[46] 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*. ACM, Memphis Tennessee USA, 42–47. https://doi.org/10.1145/2839509.2844617

[47] Jonathan P. Munson and Joshua P. Zitovsky. 2018. Models for Early Identification of Struggling Novice Programmers. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 699–704. https://doi.org/10.1145/3159450.3159476

[48] Nicola Murphy and David Messer. 2000. Differential benefits from scaffolding and children working alone. *Educational Psychology* 20, 1 (2000), 17–31.

[49] Shinichi Oeda and Genki Hashimoto. 2017. Log-data clustering analysis for dropout prediction in beginner programming classes. *Procedia computer science* 112 (2017), 614–621.

[50] 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., AUS, 157–163.

[51] 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.

[52] James Prather, Brett A Becker, Michelle Craig, Paul Denny, Dastyni Loksa, and Lauren Margulieux. 2020. What do we think we think we are doing? Metacognition and self-regulation in programming. In *Proceedings of the 2020 ACM conference on international computing education research*. 2–13.

[53] Thomas W Price, Yihuan Dong, and Dragan Lipovac. 2017. iSnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*. 483–488.

[54] Thomas W. Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *Proceedings of the 10th International Conference on Educational Data Mining, EDM 2017, Wuhan, Hubei, China, June 25-28, 2017*, Xiangen Hu, Tiffany Barnes, Arnon Hershkovitz, and Luc Paquette (Eds.). International Educational Data Mining Society (IEDMS). http://educationaldatamining.org/EDM2017/proc_files/papers/paper_36.pdf

[55] Kiki Prottsman. 2014. Computer science for the elementary classroom. *ACM Inroads* 5, 4 (2014), 60–63.

[56] Fernando J. Rodriguez, Kimberly Michelle Price, Joseph Isaac, Kristy Elizabeth Boyer, and Christina Gardner-McCune. 2017. How block categories affect learner satisfaction with a block-based programming interface. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, 201–205. https://doi.org/10.1109/VLHCC.2017.8103468

[57] Fernando J. Rodríguez and Kristy Elizabeth Boyer. 2015. Discovering Individual and Collaborative Problem-Solving Modes with Hidden Markov Models, Cristina Conati, Neil Heffernan, Antonija Mitrovic, and M. Felisa Verdejo (Eds.), Vol. 9112. Springer International Publishing, Cham, 408–418. https://doi.org/10.1007/978-3-319-19773-9_41 Book Title: Artificial Intelligence in Education Series Title: Lecture Notes in Computer Science.

[58] Melody Siadaty, Dragan Gasevic, and Marek Hatala. 2016. Trace-based micro-analytic measurement of self-regulated learning processes. *Journal of Learning Analytics* 3, 1 (2016), 183–214.

[59] Krista D Simons and James D Klein. 2007. The impact of scaffolding and student achievement levels in a problem-based learning environment. *Instructional science* 35 (2007), 41–72.

[60] Krista D Simons, James D Klein, and Thomas R Brush. 2004. Instructional strategies utilized during the implementation of a hypermedia, problem-based learning environment: A case study. *Journal of Interactive Learning Research* 15, 3 (2004), 213–233.

[61] John Stachel, Daniela Marghitu, Taha Ben Brahim, Roderick Sims, Larry Reynolds, and Vernon Czelusniak. 2013. Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. *Journal of Integrated Design and Process Science* 17, 1 (2013), 37–54.

[62] Benyamin Tabarsi, Heidi Reichert, Rachel Qualls, Thomas Price, and Tiffany Barnes. 2023. Exploring Novices' Struggle and Progress During Programming Through Data-Driven Detectors and Think-Aloud Protocols. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 179–183.

[63] Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (July 1979), 422–433. https://doi.org/10.1145/322139.322143

[64] Christina Tikva and Efthimios Tambouris. 2023. The effect of scaffolding programming games and attitudes towards programming on the development of Computational Thinking. *Education and Information Technologies* 28, 6 (2023), 6845–6867.

[65] Janneke Van de Pol, Monique Volman, Frans Oort, and Jos Beishuizen. 2015. The effects of scaffolding in the classroom: support contingency and student independent working time in relation to student achievement, task effort and appreciation of support. *Instructional Science* 43 (2015), 615–641.

[66] Arto Vihavainen, Juha Helminen, and Petri Ihantola. 2014. How novices tackle their first lines of code in an ide: Analysis of programming session traces. In *Proceedings of the 14th koli calling international conference on computing education research*. 109–116.

[67] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th international conference on advanced learning technologies*. IEEE, 319–323.

[68] Nathaniel Weinman, Armando Fox, and Marti Hearst. 2020. Exploring challenging variations of parsons problems. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 1349–1349.

[69] David Wood, Jerome S Bruner, and Gail Ross. 1976. The role of tutoring in problem solving. *Journal of child psychology and psychiatry* 17, 2 (1976), 89–100.

[70] Yingbin Zhang, Luc Paquette, Juan D Pinto, and Aysa Xuemo Fan. 2023. Utilizing programming traces to explore and model the dimensions of novices' code-writing skill. *Computer Applications in Engineering Education* 31, 4 (2023), 1041–1058.

[71] Lanqin Zheng, Yuanyi Zhen, Jiayu Niu, and Lu Zhong. 2022. An exploratory study on fade-in versus fade-out scaffolding for novice programmers in online collaborative programming settings. *Journal of Computing in Higher Education* 34, 2 (2022), 489–516.

[72] 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.

[73] 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.

[74] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An automated framework for recommending program elements to novices. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, Lincoln, Nebraska, 283–288. https://doi.org/10.1109/ASE.2015.54