Evaluating the Effectiveness of Parsons Problems for Block-based Programming

Rui Zhi, Min Chi, Tiffany Barnes, Thomas W. Price
North Carolina State University
Raleigh, NC
{rzhi,mchi,tmbarnes,twprice}@ncsu.edu

ABSTRACT

Parsons problems are program puzzles, where students piece together code fragments to construct a program. Similar to block-based programming environments, Parsons problems eliminate the need to learn syntax. Parsons problems have been shown to improve learning efficiency when compared to writing code or fixing incorrect code in lab studies, or as part of a larger curriculum. In this study, we directly compared Parsons problems with block-based programming assignments in classroom settings. We hypothesized that Parsons problems would improve students' programming efficiency on the lab assignments where they were used, without impacting performance on the subsequent, related homework or the later programming project. Our results confirmed our hypothesis, showing that on average Parsons problems took students about half as much time to complete compared to equivalent programming problems. At the same time, we found no evidence to suggest that students performed worse on subsequent assignments, as measured by performance and time on task. The results indicate that the effectiveness of Parsons problems is not simply based on helping students avoid syntax errors. We believe this is because Parsons problems dramatically reduce the programming solution space, letting students focus on solving the problem rather than having to solve the combined problem of devising a solution, searching for needed components, and composing them together.

ACM Reference Format:
Rui Zhi, Min Chi, Tiffany Barnes, Thomas W. Price. 2019. Evaluating the Effectiveness of Parsons Problems for Block-based Programming. In *International Computing Education Research Conference (ICER '19), August 12–14, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3291279.3339419

1 INTRODUCTION

Parsons problems [21] are code-completion problems, which require students to rearrange mixed up blocks of code to create the correct solution to a programming problem (see Figure 1). Parsons problems have been shown to be more efficient in helping novices learn to program than writing equivalent code, fixing buggy code [6, 8], or reading an instruction-based tutorial [13]. In these studies, students can solve Parsons problems more quickly than other forms of practice, while performing just as well or better on post test problems. However, these promising results come from short, lab-based controlled studies, with voluntarily recruited populations, and procedures that differ from many classroom contexts. For example, in a 2.5 hours study, Ericson et al. compared worked examples *paired* with Parsons problems to worked examples *paired* with problem solving practice [6, 8] and measured learning gains for the whole instructional material. More work is needed to understand how this effect might generalize across a semester, and whether Parsons problems are still more effective without paired worked examples.

It is also important to understand what features of Parsons problems make them effective. There are many ways that solving a Parsons problem differs from writing equivalent code. In Parsons problems, students drag and drop code, rather than typing it, they construct a solution from a limited set of available code blocks, and these code blocks may contain multiple lines of code. What role do these elements play in Parsons' problems effectiveness? For example, in block-based programming environments, students also construct code with a limited set of drag-and-drop blocks, rather than by typing, which has been shown to improve programming efficiency [22] and learning [31]. It is therefore worth investigating whether the previously observed benefits of Parsons problems also apply in a block-based programming environment, which already offers similar benefits to Parsons problems.

In this study, we compared Parsons problems with problem solving (writing code) during 6 weeks of a non-majors CS0 course. We investigated whether our Parsons problems increased students' programming efficiency. We also explored *how* Parsons problems save students time, by measuring differences in programming behavior between students who solved Parsons problems and those who wrote equivalent code. We used a quasi-experimental design, and compared student work across six semesters. We hypothesized that (1) students who solve block-based Parsons problems will complete the learning activities faster, and (2) perform no worse than students writing block-based code. We confirmed both of our hypotheses, finding that Parsons problems saved students 15-17 minutes on each lab assignment, between 43.6% and 65.2% of the time they would have spent writing code, and found no evidence that they negatively impacted students' later performance. Our analysis of log data suggests that this time was primarily saved by limiting students' interactions to only those blocks that are part of the solution. The primary contributions of this work are: 1) evidence that Parsons problems improve learning efficiency compared to writing block-based code in a classroom context, 2) an investigation into *how* Parsons problems contribute to programming efficiency, and 3) the design of Parsons problems for block-based programming.

ICER '19, August 12–14, 2019, Toronto, ON, Canada

ACM ISBN 978-1-4503-6185-9/19/08…$15.00
https://doi.org/10.1145/3291279.3339419

2 RELATED WORK

Based on Cognitive Load Theory, instructional designers are advised to design activities that minimize extraneous load, extra elements that distract from the important learning material to allow for germane load [29]. Merriënboer and Croock found that incomplete computer programs, in which students are given part of a correct program and asked to complete the rest, improved novice's programming performance and post-test scores, compared with those who write code with examples as a reference [30].

Parsons problems are code-completion problems, in which students re-arrange mixed up code blocks into the correct order. Ericson et al. argued that Parsons problems have a more constrained problem space, and would impose lower cognitive load than programming from scratch [6, 8]. In their original implementation, Parsons and Haden presented students with problem descriptions and a set of drag-and-drop code fragments [21]. Each code fragment comprises single or multiple lines and the code fragments may have incorrect code. Students select the correct code segments and put them in the right order to solve a puzzle. Parson and Haden argued that Parsons problems are more engaging than writing code, and their initial evaluation showed that the majority of students (14/17) thought this kind of problem is useful or very useful for learning Pascal [21]. Fabic et al. found Parsons problems are only effective for novices, and they would be more effective for supporting learning with self-explanation prompts [10]. Some studies have shown that Parsons problems are engaging [7, 9, 11]. Garner's semester-long study with eight students revealed that students were generally engaged with Parsons problems [11]. Ericson et al. integrated Parsons problems into an ebook and found that Parsons problems were engaging, as evidenced by more students attempted Parsons problems than multiple choice questions [7], and were more frequently mentioned by teachers as being valuable than other questions [9].

Parsons problems difficulty can be varied by changing what students are required to do. 2D Parsons problems are designed to teach Python, and require students to both reorder and indent the code correctly [18]. Ihantola et al. designed mobile phone-based 2D Parsons problems called MobileParsons, which allows users to make small edits to pieces, such as selecting the condition of an if statement [17], but its effectiveness has not yet been evaluated. Helminen et al. analyzed students interaction traces and identified several common patterns and difficulties that novices face while solving 2D Parsons problems, showing students have trouble indenting the code and students go back to a previous code state [14, 15]. Parsons problems can also include distractors, incorrect code with syntax or semantic errors, designed to highlight common errors that novices make. For example, a distractor might use an undefined variable called 'Total' but the defined variable is 'total'. Parsons problems are harder with distractors [4, 11, 12]. Parson and Haden's pilot study showed that students perceived that Parsons problems with distractors are helpful [21]. However, Harms et al. found that Parsons problems with distractors decrease learning efficiency compared to Parsons problems without distractors in a block-based programming environment [12]. Therefore, we did not incorporate any distractors in Parsons problems used in this study.

Moreover, Parsons problems have been studied as assessments. Studies have shown that solving Parsons problems correlates with writing code [2, 4]. Denny et al. explored Parsons problems as CS1 exam questions and they found a moderate correlation (ρ = 0.53) between scores on Parsons problems and writing code [4]. Similarly, Cheng and Harrington found a strong correlation (ρ = 0.65) between the students' score on the writing code problem and the Parsons problems [2]. Morrison et al.'s study shows that the Parsons problems can be more sensitive for assessing students' learning gains than writing code [20]. They found that Parsons problems, which allow students to show their understanding of the meaning and sequence of programs, can measure students' knowledge that cannot be measured by writing code alone.

While it is hypothesized that solving Parsons problems is more efficient and effective than traditional problem solving, only a few studies have explored the effectiveness for Parsons problem compared to writing code in teaching programming [6, 8, 13]. Ericson et al. did a study that interleaved worked examples with Parsons problems, writing equivalent code, or fixing buggy code separately, and they found that when interleaving with worked examples, Parsons problems are more efficient in helping novices learn programming than writing equivalent code or fixing buggy code [8]. Compared with writing code or fixing buggy code, Parsons problems save students time by providing the code and limiting students' solution space. Similarly, Harms et al. showed that Parsons problems are more efficient and effective in helping middle school students learn block-based programming independently than a tutorial [13]. The tutorial provides students video and textual instructions, which requires students to read and follow the instructions by finding and putting the exact code blocks into the programming area. Then students can run the code and see the animation generated by their code. In Harms et al.'s study, students using Parsons problems spent 23% less time on training and performed 26% better on transfer tasks, while they had higher cognitive load than students in the tutorial condition [13].

Overall this research and others have shown that Parsons problems can be effective, especially for learning efficiency. However more research is needed to build a better understanding of how and why they are effective. We could support novices both effectively and adaptively with a better understanding of Parsons problems. For example, we can automatically create Parsons problems from examples if they are proven to be more effective than writing code. In this study, we integrate Parsons problems into block-based programming environment to evaluate their effectiveness when compared to writing blocks-based code.

3 METHOD

To investigate the impact of Parsons problems on novice programmers' problem solving performance, we designed and integrated Parsons problems into Snap*!*. We used a quasi-experimental design, comparing student performance in a non-majors CS0 course on 6 programming assignments and a project across 6 semesters. In the Spring 2019 semester, 3 of the 6 assignments (the labs) were presented as Parsons problems, rather than the traditional problem solving used in prior semesters. We compare student solution time and code quality and their problem-solving behaviors while solving the programming problems to investigate the following research questions: How do block-based Parsons problems compare

to writing code on: **RQ1:** students' programming efficiency on the problem? **RQ2:** students' performance on post-test problems and final project? **RQ3:** students' hint usage while working on the post-test problems? **RQ4:** How do Parsons problems impact students' problem solving behavior on the problem and post-test problems? We hypothesized that students who do Parsons problems instead of writing code during programming labs will: **H1:** Accomplish the lab assignment faster than previous semesters' students. **H2:** Perform no worse than previous semesters' students on the following 'post-test' homeworks (measured by grades, time on task, and hint usage). **H3:** Perform no worse than previous semesters' students on the final project.

iSnap. All programming in this study took place in iSnap [24], which extends the Snap*!* programming environment with detailed logging and on-demand, data-driven hints. It logs all system interactions, such as code edits and click events (e.g. clicking a button or selecting an item from a menu), and students' complete code snapshots after each edit. iSnap offers on-demand hints, generated from previous students' solutions [26]. In iSnap, when students request help, if there are available hints near a block, a hint bubble appears near the block. After clicking it, a popup window will be shown including students' current code and a next-step suggested code highlighted in green. The iSnap system has been evaluated in several contexts and previous studies shows that hints did help students complete the assignment objectives [24] and students may have different expectations of human and computer tutors [25].

3.1 Parsons Problems Design

We designed and implemented block-based Parsons Problems in iSnap (Figure 1). For each problem, we separated an expert solution into individual blocks, which we present in the code palette (Figure 1(1)). Students can drag blocks together or drag variables into block inputs, but cannot duplicate or delete blocks. In traditional Parsons Problems, students manipulate entire lines of code, but in our implementation, students manipulate individual blocks, just like in Snap*!*, and it may take multiple blocks to complete a line (e.g., the "turn" statement in Figure 1). We specifically designed our Parsons problems in this way, so that students would not be disadvantaged when doing later assignments as they have practiced with how blocks interact with one another. In Snap*!*, when a student defines a procedure (a "custom block"), they use its parameters by dragging a copy of the parameter variable from the block definition header (see Figure 1). For Parsons problems, we already know how many times these parameter variables are needed in the expert code. Therefore, students can only use the parameter variables the predetermined number of times, and when they run out, the input variable is greyed out, as shown in Figure 1(D). This design guarantees that the solution will not contain extraneous variables and helps students focus on the relationship between blocks, variables, and parameters. As suggested by the element interactivity hypothesis of Cognitive Load Theory [28], reducing the number of learning elements (which often correspond to UI elements) that students interact with should decrease the cognitive load imposed on students.

Our Snap*!* Parsons problems have three modes for inputs: editable, editable with a default value, and fixed. Editable inputs have a white background, as in Snap*!*, and allow students to drop in variables. We added a default value to some editable inputs, shown in faded gray, as in Figure 1(A)), so that students can experiment with the blocks before they are done. Fixed inputs cannot be changed, and are entirely greyed out, as shown in Figure 1(B). We designed this feature with the intent to help students gain the knowledge for using variables, since we have observed in our previous study that novices have trouble using variables in Snap*!* [32].

Students can run the code whenever they want to test it. They can click the 'Reset Sprite & Clear Stage' button to clean the stage area and put the sprite in its default position (Figure 1(4)). Students can also click the undo button to remove changes one step at a time (Figure 1(3)). We also hide and disable elements such as the stage icon or functions that are not related to the problem. This design eliminates extraneous elements in the environment, taking the "weeding" suggestion from Mayer and Moreno to reduce cognitive load in multimedia learning environment [19].

After students run their code a few times, we enable 'Check My Work' help button (Figure 1(2)). Our system did not limit help requests, in order to keep our design consistent with the hint system used for all the class Snap*!* assignments. If the current program is correct, the system will suggest submitting the assignment. Otherwise, the system will highlight any misplaced blocks in the programming area (Figure 2). Our Parsons problems may contain multiple solutions, and our system can highlight misplaced blocks based on placements that no known solutions would accept. For example, 'pen down' should be put before 'pen up' and the 'move steps' blocks, but it does not matter whether it is right after the hat block or the 'set pen size' block. After they finish, students can upload solutions to the server and download a copy for reference.

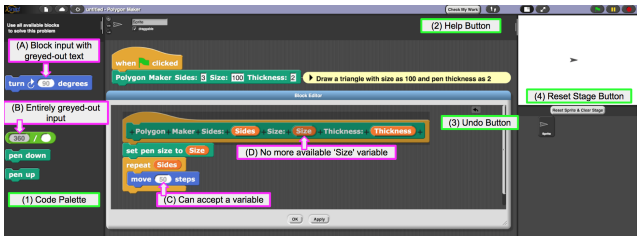

Figure 1: Parsons problem interface in Snap*!*. Available code blocks are shown in the code palette.

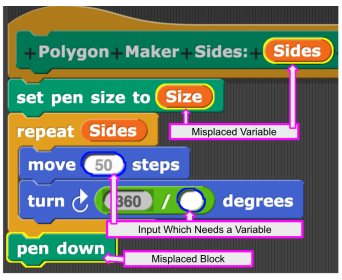

Figure 2: A hint example: yellow highlights misplaced blocks or variables, blue shows input that needs a variable.

3.2 Participants

In this study, the participants were undergraduate students, with minimal prior programming experience, enrolled in a non-majors CS0 course at a U.S. research university over 6 semesters (Fall 2016 through Spring 2019). Students are typically non-STEM majors, but we did not collect student demographic information as part of this study. This course introduces basic programming concepts in Snap*!* including variables, loops, conditionals, procedures, and lists. All semesters of the course were taught by the same instructor, and the weekly programming labs were led by undergraduate TAs in 3-4 sections. Some or all of the TAs changed each semester. There were 46-68 students each semester (M=50; see Table 2). In this course, students learned Snap*!* through 3 lab assignments, 3 homeworks, and a final project (described in the next section). During lab sessions (110 minutes), TAs introduced relevant Snap*!* concepts, administered a quiz, and provided guidance while students worked on the labs. After completing the lab, students can leave early or start the week's homework assignment. In our data, few students chose to start their homework (including in the Spring 2019 semester).

3.3 Study Design

Our study employed a quasi-experimental design. Each semester, students completed 3 programming labs and 3 homeworks, as explained in detail below. Students in the Spring 2019 semester served as the experimental group, and the 3 lab assignments (but not the 3 homeworks) were given as Parsons problems, rather than traditional problem solving. We refer to Spring 2019 as X19 to clearly indicate that this was our experimental semester. Students from the Fall 2016 (F16), Spring 2017 (S17), Fall 2017 (F17), Spring 2018 (S18), and Fall 2018 (F18) semesters served as control groups, as they did not have access to Parsons problems. For all semesters, the 3 homeworks were given as problem solving and served as assessment problems. Additionally, all students in the control and experimental groups could get help from TAs during labs and use iSnap's on-demand hints for homeworks. We did not use a randomized, controlled experiment because it would not be fair or feasible to give different assignments to students in the same class for a given semester. Except for introducing the help features and the Parsons problems right before the first lab in each semester, we were not involved in teaching the course or labs. As mentioned in section 3.1, our Parsons problems require students to compose a correct solution to a given problem from a minimal set of needed blocks and variables. Before the first lab in X19, the first author used a demo project to introduce the Parsons problems and help features to students. Each lab is followed by a homework assignment, used as an immediate post-test. The final project also served as a summative, delayed post-test.

Assignments & Course Project. Students completed 3 labs assignments: *PolygonMaker*, *Pong1*, and *GuessingGame1 (GG1)*; and 3 homeworks: *Squiral*, *Pong2*, and *GuessingGame2 (GG2)*. Students do the lab assignments in person during the lab time, and then do a similar, follow-up homework assignment outside of class. The first lab, PolygonMaker, is to write a program to draw any polygon. Its common solution is 5 lines of code using procedures (functions), loops, and variables. The following homework, Squiral, requires students to write a program to draw a spiraling square-like shape using loops, variables, arithmetic operators, and procedures. Common solutions for Squiral contain 7-10 lines of code. The second lab, Pong1, simulates the classic arcade game Pong where one player can control a paddle to hit a ball back and forth to earn points. The Pong1 lab requires students to program the ball and paddle behaviors, with common solutions using ~18 lines of code using loops, variables, conditionals, and user interactions. The homework Pong2 extends Pong1 to allow two players to compete with each other and its common solution contains ~29 lines of code using similar concepts. The third and last lab assignment, GG1, requires students to program a game that asks players to guess a number generated by the computer until they guess correctly. It informs players if they guessed too high or too low. Common solutions for GG1 have 12-18 lines of code using loops, conditionals, variables, and arithmetic operators. The GG2 homework requires students to extend GG1 by selecting a random number from player-set range and using extra variables and lists to record the total number of guesses made and the value of each guess. Common solutions for GG2 have 25-30 lines of code using loops, conditionals, variables, arithmetic operators, and lists. After all labs and homeworks, students complete a Snap*!* project that requires students to build a user interface prototype for a product using loops, conditionals, procedures, variables, and user interaction.

Because the course curriculum has changed somewhat over 3 years, each assignment can only be compared across a subset of the semesters. Table 1 shows a list of all the assignments and semesters. For a given assignment, we excluded any semester from comparison if the assignment was not offered in that semester, if its instructions did not match those used in X19, if there was another experiment being run during that semester, or if there were unique circumstances (e.g. canceled classes/labs). These cells in the table have dashes to show that semester was excluded. Specifically, Pong1 and Pong2 were not offered until S18, GG2 was modified in F18, PolygonMaker and Squiral were modified for just the F18 semester, there was an experiment during Squiral for S18, and there was a class cancellation during PolygonMaker for S18. In addition, due to a power outage, we do not have log data for Pong2 in F18, so we can only compare grades but not time or hint usage.

4 RESULTS

To answer our research questions, we compared the X19 students' homework performance with previous semesters' students' (the control) based on the code quality and time for finishing the assignment, evaluated their performance on the final project, and also analyzed students' programming traces to identify their problem solving behavior patterns.

4.1 Time on Task

To answer RQ1, we investigated the time that students took to finish the assignments. We did not investigate the time that students spent working on the final project because the project is related to their own majors and does not have specific goals like other assignments. Specifically, we calculated the time that students spent working on the problem instead of the time between opening and submitting the assignment. Since students may just open the assignment and work on other things instead, we exclude

Table 1: Average time (with Standard Deviation) for accomplishing the assignment in different semesters.

	F16	S17	F17	S18	F18	X19	K.-W. Test
Poly. (Lab)	23.6 (11.3)	27.5 (10.7)	30.0 (15.8)	-	-	9.3 (4.1)	$\chi^2(3) = 82.4, p <$ **.001**
Pong1 (Lab)	-	-	-	41.6 (11.0)	25.8 (17.2)	18.8 (6.9)	$\chi^2(2) = 72.8, p <$ **.001**
GG1 (Lab)	31.3 (12.1)	28.4 (10.5)	30.6 (11.8)	24.1 (10.9)	23.3 (8.5)	11.0 (6.2)	$\chi^2(5) = 111.4, p <$ **.001**
Squiral (HW)	23.6 (21.5)	20.4 (15.3)	28.5 (31.8)	-	-	19.0 (14.9)	$\chi^2(3) = 4.4, p = .23$
Pong2 (HW)	-	-	-	47.6 (35.8)	-	51.7 (31.0)	$\chi^2(1) = 1.5, p = .22$
GG2 (HW)	-	-	-	-	62.3 (50.1)	50.8 (24.0)	$\chi^2(1) < 0.1, p = .99$

the idle time, defined as any period of 5 minutes or more without interactions in iSnap. Table 1 shows the average time along with standard deviation that students took for finishing each assignment in each semester. The results show that students spend much less time on Parsons problems than traditional problem solving. The Parsons problems save students nearly 17 minutes on average in PolygonMaker (65.2% of the total problem solving time), 15 minutes (43.6%) on average in Pong1, and 17 minutes (60.6%) on average in GG1. For each assignment, we performed a Kruskal-Wallis test[1] to determine if these differences in active time across semesters were significant. As shown in the final column of Table 1, the differences were significant for each lab assignment, but not for any of the homework assignments. For each lab assignment, we performed a post hoc Dunn's test, using the Benjamini Hochberg procedure to control the false discovery rate at 5% [1, 5], to identify pairwise differences in completion time between semesters. In each case, X19 had significantly lower lab completion time than all other semesters (all $p < 0.001$). These findings support H1, that students with Parsons problems will complete these lab assignments more quickly. The findings also partially support H2, since there were no significant differences across semesters in the time taken to complete homework assignments. For most assignments, there were no significant differences in completion time among the control semesters (without Parsons problems). However, for Pong1, we also found that F18 < S18 ($z = 5.78, p < 0.001$), and for GG1, we found that F18 < F16 ($z = 3.35, p = .002$), F18 < F17 ($z = 2.90, p = .007$), F18 < S18 ($z = 2.15, p = .048$), S18 < F16 ($z = 3.17, p = .003$) and S18 < F17 ($z = 2.75, p = .010$). We would discuss the potential reasons for this result in section 5.

4.2 Effectiveness

To answer our second research question, we compared students' grades for each homework assignment across semesters, as reported in Table 2. Students who did not complete the assignment received a grade of 0, and we found the dropout rate is consistent across semesters, ranged from 0% to 10.9%. For completeness, we also

[1]Nonparametric tests were used because the distribution of the data was non-normal, indicated by the Shapiro-Wilk test.

Table 2: Average class grades (with Standard Deviation) for each assignment in different semesters.

	F16 (68)	S17 (50)	F17 (51)	S18 (46)	F18 (49)	X19 (48)	K.-W. Test Result
Poly. (Lab)	64.6 (16.7)	60.9 (13.7)	63.1 (13.3)	-	-	64.3 (12.6)	$\chi^2(3) = 24.7, p <$ **.001**
Pong1 (Lab)	-	-	-	60.6 (20.2)	67.3 (9.9)	66.0 (11.1)	$\chi^2(2) = 3.1, p = .21$
GG1 (Lab)	65.8 (14.6)	65.7 (14.4)	66.1 (13.9)	61.1 (21.6)	68.1 (10.1)	67.0 (14.1)	$\chi^2(5) = 5.3, p = .38$
Squiral (HW)	81.3 (23.6)	81.0 (31.7)	81.9 (29.8)	-	-	81.5 (24.4)	$\chi^2(3) = 6.9, p = .08$
Pong2 (HW)	-	-	-	74.5 (31.4)	87.5 (22.1)	86.2 (27.5)	$\chi^2(2) = 9.5, p =$ **.009**
GG2 (HW)	-	-	-	-	77.1 (26.5)	87.7 (20.0)	$\chi^2(1) = 7.7, p =$ **.006**
Project	-	-	-	25.1 (7.4)	26.0 (4.9)	27.0 (4.8)	$\chi^2(2) = 3.4, p = .18$

report grades for the lab assignments, but we note that this comparison is less meaningful, since students can get help from TAs and have historically performed uniformly well on these assignments. We performed a Kruskal-Wallis test to determine if these differences in grades across semesters were significant. As shown in Table 2, we found significant differences for 3 assignments: PolygonMaker (in-lab), Pong2 (HW) and GG2 (HW). For these assignments, we performed post hoc Dunn's tests, using the Benjamini Hochberg procedure to control the false discovery rate at 5%, to identify pairwise differences in grades between semesters. For PolygonMaker, we found that F16 had a significantly higher score than all other semesters (F16 vs S17, $z = 4.72, p < 0.001$; F16 vs F17, $z = 3.40, p = .002$; F16 vs X19, $z = 2.20, p = .04$), and that X19 had a higher score than S17 ($z = 2.30, p = .04$). For Pong 2, we found that S18 had a significantly lower score than all other semesters (S18 vs F18, $z = 2.27, p = .04$; S18 vs X19, $z = 2.95, p = .0095$). For GG2, we found the X19 had a significantly higher score than F18 ($p = .006$). We would talk about the possible explanations in section 5.

Our data show no evidence that students with Parsons problems on lab assignments perform any worse than those with traditional problem solving, supporting H2. While this lack of evidence is not sufficient to claim that the two groups performed *equally well* in a statistical sense, the mean X19 grade was never more than 1 percent worse than any other semester and was significantly higher than some semesters for Pong2 and GG2. .

4.3 Log Analysis

In this section we discuss results for RQ3 and RQ4 to investigate the impact of Parson's problems on later programming behaviors. **RQ3:** How do block-based Parsons problems compared to writing code on students' hint usage while working on the post-test problems? **RQ4:** How do Parsons problems impact students' problem solving behavior on the lab problem and on post-test problems?

4.3.1 Hint Usage. We investigated the number of times that students asked for hints across semesters. The average number of hint

requests per student for Squiral are 3.23 (SD = 6.9) in F16, 3.65 (SD = 8.0) in S17, 1.40 (SD = 5.6) in F17, and 5.0 (SD = 9.08) in X19; for Pong2 are 10.28 (SD = 24.7) in S18 and 17.0 (SD = 30.3) in X19, respectively; for GG2 are 21.80 (SD = 42.0) in F18 and 14.62 (SD = 26.2) in X19. We performed a Kruskal-Wallis test for each homework and found no significant differences between X19 and other semesters.

4.3.2 Problem solving behavior. To figure out why Parsons problems save students time, we analyzed students' code traces to investigate their programming patterns. We found several common patterns that differentiate Parsons problems with writing block-based code by manually examining students' code traces. These patterns demonstrate why solving Parsons problems is efficient. We hypothesize that the block-based Parsons problems can save students time by reducing the time spent on, or even avoiding, unproductive patterns. We investigated three potential unproductive behaviors: searching for blocks to use, editing block inputs, and testing irrelevant blocks. These behaviors were selected based on a manual inspection of lab programming traces from control students who took much longer than the median time, looking for patterns that did not seem to result in significant differences in the student code, but took longer than a few seconds. Once these candidate behaviors were determined, we compiled statistics for all students for these behaviors on each homework and control group lab. X19 students could not perform these behaviors within Parsons problems. We hypothesized that although Parsons problems prevent these unproductive behaviors during the lab, they would not disproportionately increase these unproductive times during the homework.

Search blocks. To write code in Snap*!*, students have to search and find the blocks they need from the code palette. They need to change categories and skim the code palette to find desired blocks. Prior work investigating block-based programming examples show that novices have trouble finding blocks presented in examples [16]. We argue that Parsons problems organize all the needed blocks together and hence reduce search time. To investigate this for students in the control group, we calculated the time spent searching for blocks as any time between a category switch and the addition of a block into the script window. We found that students in the control group spent 1.32 ($SD = 0.99, 4.98\%$ of the total problem solving time), 2.66 ($SD = 1.33, 8.0\%$), 2.66 ($SD = 1.47, 9.5\%$) minutes on average for finding blocks in PolygonMaker, Pong1, and GG1 respectively. A Spearman's rank-order correlation was run to assess the relationship between the time spent finding blocks and solving the lab assignments. There was a statistically significant, strong positive correlation between the proportion of time spent finding blocks and solving PolygonMaker ($\rho = 0.75, p < 0.001$), Pong1 ($\rho = 0.59, p < 0.001$), and GG1 ($\rho = 0.69, p < 0.001$). For the homework assignments, we found students in the control group spent 1.47 ($SD = 1.37, 6.1\%$), 2.46 ($SD = 1.95, 5.2\%$), 3.81 ($SD = 4.37, 6.1\%$) minutes on average for finding blocks in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.09 ($SD = 1.05, 5.8\%$), 2.74 ($SD = 2.38, 5.3\%$), 3.61 ($SD = 2.16, 7.1\%$) minutes on average for finding blocks in Squiral, Pong2, and GG2 respectively. We performed a Kruskal-Wallis test to determine if these differences in time spent searching blocks across semesters were significant. We did not find any significant difference between semesters on time spent searching blocks. Students in the experimental group did not spend more time searching for blocks than students in the control group. This indicates that searching for blocks during the lab assignment did not help control group students more quickly find the blocks during the following homeworks. Therefore, block-based Parsons problems can save students time by eliminating the need for changing categories, looking through blocks in the category, and choosing the desired block.

Edit Input. We also investigated the time students spent on editing block inputs. Control group students need to type literals or characters into block inputs while solving a problem from scratch, but block-based Parsons problems either provide the value to a block, or only allow drag and drop from variables, so X19 students do not need to type anything to enter block inputs. To determine edit input time for the control group, we choose 9 seconds (the third quartile value of all edit time) as the maximum value for editing and capped any times that exceeds this limit. We calculated the time students spent on editing block inputs and we found that students in the control group spent 2.88 ($SD = 1.68$, 10.8% of the total problem solving time), 2.05 ($SD = 1.84, 6.2\%$), 2.53 ($SD = 1.19, 9.1\%$) minutes on average to edit block inputs in PolygonMaker, Pong1, and GG1 respectively. We found a statistically significant positive Spearman correlation between the proportion of time spent editing block inputs and solving PolygonMaker ($\rho = 0.55, p < 0.001$), Pong1 ($\rho = 0.88, p < 0.001$), and GG1 ($\rho = 0.53, p < 0.001$). For the homework assignments, we found students in the control group spent 2.70 ($SD = 3.50, 11.2\%$), 2.47 ($SD = 2.36, 5.2\%$), 3.37 ($SD = 2.35, 5.4\%$) minutes on average for editing block inputs in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.92 ($SD = 1.27, 10.1\%$), 2.58 ($SD = 1.72, 5.0\%$), 3.07 ($SD = 1.39, 6.0\%$) minutes on average for editing inputs in Squiral, Pong2, and GG2 respectively. The Kruskal-Wallis test did not show any significant difference between semesters on time spent editing block inputs. This indicates that block-based Parsons problems can save students time by providing the default values for blocks without increasing students' time editing block input for the following homeworks.

Test 'irrelevant' blocks. We defined 'irrelevant' blocks as blocks that were used during problem solving but did not exist in a student's final submission or a common solution. Students may spend time using a block but finally find that it is irrelevant to the solution and hence delete it. For example, a student may use 'set x' and 'change x' block, instead of 'move' block, to change the position of the sprite, but would eventually delete those blocks since they are irrelevant to the solution. To quantify this behavior, we investigated students' programming traces and for each block interaction, we classified the block as irrelevant when it neither exists in the student's submission nor exists in a correct solution. Then we accumulate the time each student spent interacting with irrelevant blocks in each assignment. Note that we excluded from this measure any time that students spent on editing the irrelevant block inputs or on creating a new block since these measures were already included in the previous two behaviors. We found that students in the control group spent 2.31 ($SD = 2.16$, 8.7% of the total problem solving time), 1.36 ($SD = 1.63, 4.1\%$), 1.46 ($SD = 1.58, 5.2\%$) minutes on average for testing irrelevant blocks in PolygonMaker, Pong1, and GG1 respectively. A Spearman's rank-order correlation was run to assess the relationship between the time spent on irrelevant blocks and solving the

lab assignments. There was a statistically significant, very strong positive correlation between the proportion of time spent on irrelevant blocks and solving PolygonMaker ($\rho = 0.87, p < 0.001$), Pong1 ($\rho = 0.96, p < 0.001$), and GG1 ($\rho = 0.95, p < 0.001$). For the homework assignments, we found students in the control group spent 2.38 ($SD = 3.78, 9.1\%$), 2.06 ($SD = 2.22, 4.3\%$), 4.93 ($SD = 9.28, 7.9\%$) minutes on average with irrelevant blocks in Squiral, Pong2, and GG2 respectively. Students in the experimental group spent 1.33 ($SD = 2.09, 7.0\%$), 1.44 ($SD = 1.23, 2.8\%$), 2.20 ($SD = 2.01, 4.3\%$) minutes on average on irrelevant blocks in Squiral, Pong2, and GG2 respectively. The Kruskal-Wallis test revealed significant differences on time spent using irrelevant blocks between semesters in Squiral ($\chi^2(3) = 12.03, p = .007$). However, a post hoc Dunn's test show no higher usage of irrelevant blocks in the X19 semester compared to any other semesters. Additionally, there was no significant difference between semesters on time spent using irrelevant blocks in Pong2 and GG2. This indicates that students did not benefit from using the irrelevant blocks during the preceding lab. This is consistent with Harms et al's study that irrelevant blocks (distractors) took students time without contributing to learning [12].

By adding the time spent on these three behaviors together, we found students spent 6.5 minutes on these unproductive behaviors on average in PolygonMaker (24.5% of the total problem solving time), 6.1 minutes (18.2%) on average in Pong1, and 6.7 minutes (23.8%) on average in GG1. Since there is no significant difference in the total time spent on homework assignments between the control and experimental groups, we found that students in the control group did not benefit from these behaviors. Furthermore, control group students did not spend less time than X19 students on these behaviors while solving homework assignments. This means that the time they spent on these behaviors during the lab did not save them time during the homework– they did not learn where the blocks were, get faster at editing inputs, or spend less time on irrelevant blocks. Hence we can argue that these behaviors are unproductive. The block-based Parsons problems save students time by preventing those unproductive behaviors without causing them to significantly increase later.

Our results in section 4.1 shows that Parsons problems save more time than we calculated here. We estimated the time spent on our three unproductive behaviors conservatively, but since they do not account for all the time differences between the experimental and control groups, control group students must have spent additional time exploring the problem solution space. For example, we observed traces where students repeatedly enter values in 'move' and 'turn' blocks and run the code to see how it draws, rather than trying to design a math- or pattern-based solution to draw a shape. To explore differences in how the groups explored the solution space, we designed a visualization to show how students' programs changed over time via programming path traces. We hypothesize that the limited blocks in Parsons problems reduce the need for students to explore the problem solution space. If this is true, then a visualization of programming traces should show smaller differences between consecutive states for Parsons problems, and more similarity between Parsons problems traces. To investigate this, we sampled 15 snapshots, evenly distributed throughout students' programming traces, from each student's PolygonMaker programming traces in F16, S17, F17, and X19 semesters. Figure 3 visualizes the 15 sequential snapshots of each student for PolygonMaker and Squiral. Each point on the graph represents a program snapshot, and the axes are a 2D projection that preserves tree edit distances between two snapshots[2]. Figure 3 (left) demonstrates that Parsons problems paths (yellow) have much less diversity than those from writing block-based code. Figure 3 (right) shows that this difference did not extend to the next homework assignment, Squiral. This shows that although X19 students did not highly explore the lab solution space, they still had similar programming paths to the control group on the subsequent homework.

Since Parsons problems provide only the necessary blocks for solving a problem, students will not spend time on irrelevant blocks, searching for blocks, or editing block inputs. The limited blocks in Parsons problems constrain students' solution space exploration, which saves students time. However, reducing these behaviors does not seem to negatively impact homework performance, supporting our hypothesis that Parsons problems are more efficient than writing block-based code.

5 DISCUSSION

Our results show that solving Parsons problems can lead to more efficient learning than writing block-based code. Our main hypotheses are supported by the results. We found students who do Parsons Problems instead of problem solving during programming labs: accomplish the lab assignment faster than previous semesters' students (H1), perform no worse than previous semesters' students on the following homework assignments, as measured by grades, time on task, and hint usage (H2) and the final programming project (H3).

Impact of Parsons Problems on Students' Learning. In this study, we found Parsons problems save students 43.6% to 65.2% of total problem solving time, without reducing performance on subsequent problems. This is consistent with previous studies that Parsons problems improve students learning efficiency [6, 8, 13]. Importantly, our results show that the benefits of Parsons problems can generalize to a 6-week classroom study and comparably larger problems (with up to 18 lines of code that took 33 minutes on average). Additionally, our results differ from prior work in that we compared Parsons problems directly with programming from scratch (without worked examples or tutorials), and we find just as strong results.

However, we did find some nuances to Parsons problems' benefits. First, we saw X19 students had higher usage of hints, though not significantly higher, for Pong2 and Squiral than students in other semesters. While X19 students performed just as well, we cannot determine whether this was in part due to a higher reliance on hints. However, due to the quasi-experimental nature of our study, we could not remove hints from the homework without creating an unfair comparison. Additionally, we saw some suggestive evidence that Parsons problems may actually improve performance on subsequent assignments (e.g. Pong2, GG2). This is agreed somewhat with results from Ericson et al. [6], who found that adaptive Parsons problems led to higher learning than students who solve off-task Parsons problems on turtle graphics. Since our Parsons

[2]We derived this layout using non-metric multidimensional scaling [3].

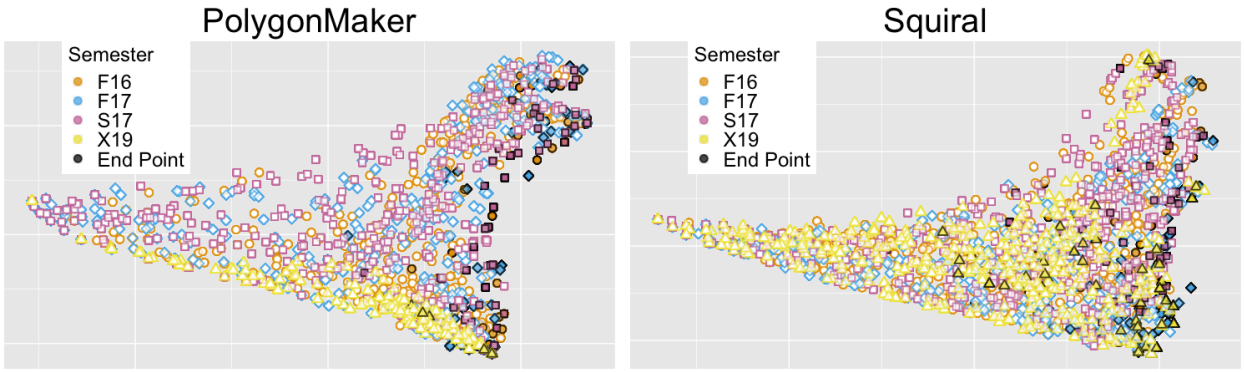

Figure 3: A 2D visualization of the similarity between students' snapshots at different points in time, across semesters, for PolygonMaker and Squiral. Color indicates the snapshot's semester, and black borders indicated a submitted solution.

problems were not adaptive, this suggests that we may see measurably improved performance from Parsons problems if we employ Ericson et al.'s approach.

We did find a few significant differences between control semesters for lab assignments (e.g. time on task and class grades for PolygonMaker), and these differences may be explained by the variance of TAs and student across semesters. The teacher may also become better at teaching this course. However, the magnitude of the reduction in time on task we saw in the X19 semester is much larger than these fluctuations, and it unlikely that our results were due to chance.

How do Parsons Problems Improve Learning Efficiency? To investigate the efficiency of Parsons problems, we analyzed students' programming traces and defined three unproductive behaviors that can be avoided by solving Parsons problems. A key contribution of our study was comparing Parsons problems to writing block-based code directly. Our results show that, despite the affordances of block-based programming that can improve students outcomes [22, 31], such as drag-and-drop blocks and limited block sets, Parsons problems implementation in block-based environment still add some additional benefits. Our results suggest that Parsons problems may save time by preventing students from engaging in various unproductive activities such as searching blocks, editing block inputs, and testing irrelevant blocks. Again, while we cannot conclude that those activities do not contribute to learning, we saw no evidence that engaging in these activities in the lab assignments reduced their incidence in the subsequent homeworks. Interestingly, we found high (sometimes almost perfect) correlations between the proportion of time spent on these behaviors and total time spent on homeworks. However, we do not claim that Parsons problems reduce these unproductive behaviors in later assignments, just that these behaviors are clear indicators of struggling students.

Even with the time saved for unproductive behaviors, we found that we could not explain all the time saved by Parsons problems. However, our results in Figure 3 suggest that one way Parsons problems save time is by reducing the space of possible partial programs that students must explore, constraining them along a more direct path towards a solution. Prior work suggests that novice programmers create a wide variety of solutions to programming problems [27], especially in block-based programming [23]. By constraining the search space, we seem to save time. This also makes sense from the perspective of cognitive load and element interactivity [28]. Since students have fewer blocks to interact with, Parsons problems would impose less cognitive load to students. As Ericson hypothesized, Parsons problems may lead to lower cognitive load, although they did not find enough evidence to support this [6].

6 CONCLUSION

Limitations. This is a quasi-experimental study with important limitations. First, our analysis involved comparisons of 3 variables (time, grades and hint usage) across 6 assignments, which involved repeated statistical tests. Because our analysis was targeted at investigating a priori hypotheses, we did not correct for these multiple tests, except to control the false discovery rate for post hoc tests. Second, we use six semesters data and the teacher may become more skilled in teaching this course and the variance of undergraduate TAs may have an impact on students performance as well. Third, even though the course is designed for non-majors and students are self-selected to register for this course, students' programming ability may vary from semesters. These are inherent limitations to any quasi-experiment across semesters; however, we mitigated this as much as possible by comparing across many semesters.

In conclusion, this study evaluates the effectiveness of Parsons problems for block-based programming and we found that Parsons problems save students nearly half of total problem solving time. This paper contributes to the design of block-based Parsons problems and provides evidence on the effectiveness of Parsons problems that mainly comes from solving the problem instead of from the benefits of the block-based programming language. Additionally, it provides insights on how Parsons problems impact students problem solving behavior. Our results encourage teachers to ask students to solve Parsons problems instead of writing code in novice programming environments. Furthermore, this study also increased our knowledge of how Parsons problems work.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under grant 1623470.

REFERENCES

[1] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society* 57, 1 (1995), 289–300.

[2] Nick Cheng and Brian Harrington. 2017. The Code Mangler: Evaluating Coding Ability Without Writing Any Code. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (2017), 123–128. https://doi.org/10.1145/3017680.3017704

[3] Trevor F Cox and Michael AA Cox. 2000. *Multidimensional scaling*. Chapman and hall/CRC.

[4] 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*. ACM, 113–124.

[5] Olive Jean Dunn. 1964. Multiple comparisons using rank sums. *Technometrics* 6, 3 (1964), 241–252.

[6] 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, 60–68.

[7] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. 2015. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. *Proceedings of the eleventh annual International Conference on International Computing Education Research - ICER '15* (2015), 169–178. https://doi.org/10.1145/2787622.2787731

[8] 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 Conference on Computing Education Research*. ACM, 20–29.

[9] Barbara J. Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison, and Mark Guzdial. 2016. Identifying Design Principles for CS Teacher Ebooks through Design-Based Research. (2016), 191–200. https://doi.org/10.1145/2960310.2960335

[10] Geela Venise Firmalo Fabic, Antonija Mitrovic, and Kourosh Neshatian. 2017. A Comparison of Different Types of Learning Activities in a Mobile Python Tutor. *Proceedings of the 25th International Conference on Computers in Education* (2017), 604–613.

[11] Stuart Garner. 2007. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Issues in Informing Science and Information Technology* 4 (2007), 491–501. https://doi.org/10.28945/966

[12] 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*. ACM, 241–250.

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

[14] Juha Helminen, Petri Ihantola, Ville Karavirta, and Satu Alaoutinen. 2013. How do students solve parsons programming problems? - Execution-based vs. line-based feedback. *Proceedings - 2013 Learning and Teaching in Computing and Engineering, LaTiCE 2013* (2013), 55–61. https://doi.org/10.1109/LaTiCE.2013.26

[15] 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*. ACM, 119–126.

[16] Michelle Ichinco, Kyle J Harms, and Caitlin Kelleher. 2017. Towards Understanding Successful Novice Example User in Blocks-Based Programming. *Journal of Visual Languages and Sentient Systems* (2017), 101–118.

[17] Petri Ihantola, Juha Helminen, and Ville Karavirta. 2013. How to Study Programming on Mobile Touch Devices - Interactive Python Code Exercises. *Koli Calling '13* (2013), 51–58. https://doi.org/10.1145/2526968.2526974

[18] Petri Ihantola and Ville Karavirta. 2011. Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. *Journal of Information Technology Education: Innovations in Practice* 10 (2011), 119–132. https://doi.org/10.28945/1394

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

[20] 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, 42–47.

[21] 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*. Australian Computer Society, Inc., 157–163.

[22] Thomas W Price and Tiffany Barnes. 2015. Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual International Conference on International Computing Education Research*. ACM, 91–99.

[23] Thomas W Price, Yihuan Dong, and Tiffany Barnes. 2016. Generating Data-driven Hints for Open-ended Programming. *EDM* 16 (2016), 191–198.

[24] 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*. ACM, 483–488.

[25] Thomas W Price, Zhongxiu Liu, Veronica Cateté, and Tiffany Barnes. 2017. Factors Influencing Students' Help-Seeking Behavior while Programming with Human and Computer Tutors. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ACM, 127–135.

[26] Thomas W Price, Rui Zhi, and Tiffany Barnes. 2017. Evaluation of a Data-driven Feedback Algorithm for Open-ended Programming. In *EDM*.

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

[28] John Sweller. 2010. Element interactivity and intrinsic, extraneous, and germane cognitive load. *Educational Psychology Review* 22, 2 (2010), 123–138. https://doi.org/10.1007/s10648-010-9128-5 arXiv:arXiv:1002.2562v1

[29] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998).

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

[31] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 3.

[32] Rui Zhi, Thomas W. Price, Nicholas Lytle, Yihuan Dong, and Tiffany Barnes. 2018. Reducing the State Space of Programming Problems through Data-Driven Feature Detection. In *EDM Workshop*.