Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems

Barbara J. Ericson
Georgia Institute of Technology
Atlanta, Georgia
ericson@cc.gatech.edu

James D. Foley
Georgia Institute of Technology
Atlanta, Georgia
jim.foley@cc.gatech.edu

Jochen Rick
Georgia Institute of Technology
Atlanta, Georgia
jochen.rick@gatech.edu

ABSTRACT

Practice is essential for learning. There is evidence that solving Parsons problems (putting mixed up code blocks in order) is a more efficient, but just as effective, form of practice than writing code from scratch. However, not all students successfully solve every Parsons problem. Making the problems adaptive, so that the difficulty changes based on the learner's performance, should keep the learner in Vygotsky's zone of proximal development and maximize learning gains. This paper reports on a study comparing the efficiency and effectiveness of learning from solving adaptive Parsons problems vs non-adaptive Parsons problem vs writing the equivalent code. The adaptive Parsons problems used both intra-problem and inter-problem adaptation. Intra-problem adaptation means that if the learner is struggling to solve the current problem, the problem can dynamically be made easier. Inter-problem adaptation means that the difficulty of the next problem is modified based on the learner's performance on the previous problem. This study provides evidence that solving intra-problem and inter-problem adaptive Parsons problems is a more efficient, but just as effective, form of practice as writing the equivalent code.

CCS CONCEPTS

• **Social and professional topics → Computing education**; **Student assessment**;

KEYWORDS

Parsons problems; adaptive Parsons problems; Parsons puzzles; Parson's problems; zone of proximal development; cognitive load

ACM Reference Format:
Barbara J. Ericson, James D. Foley, and Jochen Rick. 2018. Evaluating the Efficiency and Effectiveness of Adaptive Parsons Problems. In *ICER '18: 2018 International Computing Education Research Conference, August 13–15, 2018, Espoo, Finland.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3230977.3231000

1 INTRODUCTION

Several countries, including the United States, want to increase computing in K-12 [3, 5, 11, 12, 17, 24]. To accomplish this goal, thousands of teachers with no programming experience need to learn programming [18, 31]. However, learning to program can be difficult [7, 10, 39]. Novice programmers spend hours trying to fix errors in their programs, like unmatched parentheses [6]. Most introductory programming courses require novices to learn by writing many small programs [29, 35]. However, writing programs, even short ones, is a complex cognitive task, which can easily overwhelm novices and impede learning [40, 44]. Busy teachers need a more efficient way to learn programming.

One recommended way to reduce cognitive load is to use a completion task rather than a whole task [43]. Parsons problems are a type of code completion task in which the correct code to solve a problem is provided, but is broken into mixed up code blocks, and the user must place the blocks in the correct order [34].

Ericson, Margulieux, and Rick provided evidence that solving non-adaptive Parsons problems is more efficient (with respect to completion time), and just as effective (with respect to learning gains) as fixing the same code with errors or writing the equivalent code [22]. However, that study did not include a control group to verify that the learning gains were due to the instructional practice condition, rather than from answering the same or similar questions with feedback. Also, some students struggle to solve Parsons problems and some never solve them [21], which means that Parsons problems could be improved.

This study compared the efficiency and effectiveness of adaptive (both intra-problem and inter-problem) Parsons problems vs non-adaptive Parsons problems vs writing the equivalent code. It also included a control group that solved off-task adaptive Parsons problems. This study tested the following three hypotheses:

- **H1**: Learners who solve adaptive and non-adaptive Parsons problems will finish the instructional problems significantly faster than the learners who write code.
- **H2**: Learners who solve adaptive Parsons problems will have similar learning gains from pretest to immediate posttest as those who solve non-adaptive Parsons problems and write code.
- **H3**: Learners who solve off-task (not related to the pretest questions) adaptive Parsons problems (the control group) will have lower learning gains than those who solve on-task problems.

This paper contributes to research on adaptive learning and Parsons problem. It is the first study of the efficiency and effectiveness of intra-problem and inter-problem adaptive Parsons problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICER '18, August 13–15, 2018, Espoo, Finland
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5628-2/18/08…$15.00
https://doi.org/10.1145/3230977.3231000

2 RELATED WORK

This research is based on cognitive load theory. The study was informed by prior research on adaptive learning and Parsons problems.

2.1 Cognitive Load Theory

Cognitive Load Theory (CLT) was developed by John Sweller in the late 1980s [40]. It can be used to improve the design of instructional material. Three types of cognitive load are described in the theory: intrinsic cognitive load, germane cognitive load, and extraneous cognitive load. Intrinsic cognitive load is the amount of load due to the difficulty of the problem being solved. Extraneous cognitive load is the load added by the complexity of the instructional materials. Germane cognitive load is the load devoted to the processing, construction and automation of schemas in long-term memory. Schemas are cognitive frameworks for organizing and interpreting information.

Instructional materials should be designed to free up working memory to allow learning to occur by reducing the extraneous load and focusing resources on the germane load to allow for the construction of schemas. If the instructional material overloads working memory then learning is impeded. If the intrinsic load is too high then the problem should be broken into smaller and simpler sub problems. It is important to note that the amount of cognitive load that a learner experiences is based on the learner's prior knowledge. This implies that learning should be enhanced if the task is adapted based on the learner's prior performance.

2.2 Research on Adaptive Learning

Corbalan, Kester, and van Merrienboer found that dynamically adaptive practice, where the practice problems are adapted based on the learners prior performance, improves learning, takes less time, and increases engagement [15]. This is not surprising since adaptive practice is more likely to keep the learner in Vygotsky's zone of proximal development [8], which is what the learner can accomplish with support versus independently.

Soloway, Guzdial, and Hay called for more scaffolding to support learners as they try to accomplish new tasks [36]. To be most effective, scaffolding should fade as the learner develops expertise. In other words, the system should adapt to the learner's performance to reduce the cognitive load of the task.

Intelligent Tutoring Systems (ITS) provide scaffolding on the current problem in the form of explicit hints if the learner is struggling to solve the problem. They also use selection adaptation, which means that after the learner finishes a problem the system selects the next problem based on the learner's performance and a model of what the student has mastered. ITS take a great deal of time to build [16] and are not widely used [2]. Parsons problems are relatively easy to create and Parsons problems are already supported in several free online learning environments [21, 30].

2.3 Research on Parsons Problems

Parsons problems should have lower cognitive load than code writing problems, since they are a type of code completion problem. There are several variants on Parsons problems. Some include distractor blocks which include syntactic or semantic errors and should not be used in a correct solution. In some Parsons problems, a distractor block is shown either above or below the correct code block as shown on the left in Figure 1. These are called paired distractors. In unpaired distractors, shown on the right in Figure 1, the distractor blocks are randomly mixed in with the correct code blocks. Some Parsons problems require the learner to indent the code horizontally. These are called two-dimensional Parsons problems.

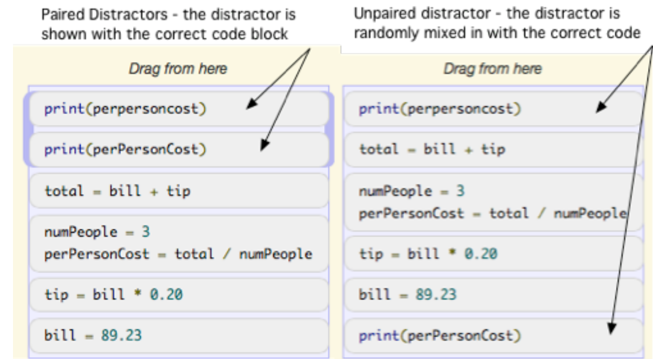


Figure 1: Paired distractor and correct code on the left and an unpaired distractor randomly mixed in with the correct code on the right

Research has shown that Parsons problems with only the correct code (no distractors) are the easiest to solve [25, 26], while conversely, increasing the number of distractors in a Parsons problem increases the difficulty of the problem [19]. Parsons problems with visually paired distractors are easier to solve than those with unpaired distractors [19]. Parsons problems that require the learner to provide indentation are harder than those that do not [19, 28]. Parsons problems with more blocks tend to be harder to solve than those with fewer blocks [21], especially if students are just randomly trying different combinations of blocks [27]. While many students find solving Parsons problems engaging [34], students sometimes struggle to solve the problems, and some students give up without ever solving them [21]. Since learning gains are based on the number of practice problems that students solve and understand [1], scaffolding that allows students to solve more problems should improve learning. However, it is also important that practice problems challenge the learner. Bjork and Bjork found that making practice too easy reduced learning gains [9]. They advocate for *desirable difficulties* during learning to improve long-term retention. Our adaptive Parsons problems were designed to scaffold learners that need help solving the current problem while providing desirable difficulties to learners who found the last problem too easy.

Kumar created a web-based tool for adaptive Parsons problems called Epplets [30]. Epplets uses selection adaptation, similar to what is used in Intelligent Tutoring Systems. This means that the next Parsons problem to solve is selected based on the learner's performance on the previous problem. This is the same approach used in Intelligent Tutoring Systems. Similar problems are presented until the user has demonstrated mastery on a particular concept. He reported that students get faster at solving similar Parsons problems, but did not compare solving selection-based adaptive Parsons

problems with other forms of practice, such as writing code. His approach also requires a valid student model to track mastery.

3 INTRA-PROBLEM AND INTER-PROBLEM ADAPTIVE PARSONS PROBLEMS

This study used two new types of adaptation: intra-problem adaptation and inter-problem adaptation. In *intra-problem adaptation* if the learner is struggling to solve the current Parsons problem, then the problem can be made easier by removing distractors, providing indentation, or combining blocks. These changes are implicit hints which should guide the learner to the correct solution, compared to the explicit hints offered by Intelligent Tutoring Systems.

Each time the user asks for help by clicking the "Help Me" button one action is taken to make the problem easier. Help is only provided if the user has made at least three full attempts, otherwise an alert is shown explaining that help is not yet available. A full attempt contains at least the required number of blocks.

After three full unsuccessful attempts an alert is shown to inform the user that help is available. If the user asks for help, and a distractor block has been used in the solution area on the right side, then the distractor block animates moving back from the solution area to the source area on the left and then grays out to show that it is disabled. Animation is useful for grabbing attending and conveying a change over time [4, 13]. If the distractor block was originally shown paired with its correct code block and the correct code block is still in the source area on the left, then the distractor moves below the correct code and the purple edge decorations shown in Figure 1 are drawn to again show that the blocks are paired. If there are no distractors in the solution area, and the solution requires indentation, then space is slowly added to the code blocks to provide the indentation. Finally, if there are no distractors in the solution and indentation is not needed, the system will animate moving one block below another and then redraw the two blocks as one block. If the user asks for help, and there are only three blocks left in the correct solution, the user is told that they should be able to solve the problem.

In *inter-problem adaptation*, the difficulty of the next problem is modified based on the learner's performance on the previous problem. While this is somewhat similar to selection adaptation, it differs in that it does not require a model of the learner's mastery of concepts and it does not affect which problem the learner solves next. If the learner solved the last Parsons problem in only one attempt, then the next Parsons problem is made more difficult by un-pairing distractors (randomly mixing them in with the correct code) and by using all available distractors. If it took the learner four or five attempts to solve the last Parsons problem, then on the next problem the distractors are shown paired with the correct code blocks. If it took the learner 6-7 attempts to solve the last problem, then 50% percent of the available distractors are removed and the remaining distractors are shown paired with the correct code blocks on the next problem. If it took the learner 8 or more attempts to solve the last problem, then all distractors are removed from the next problem. The goal is to keep the learner in Vygotsky's zone of proximal development to optimize learning.

4 METHODS

This was a between-subjects study to test the efficiency (time to complete the practice problems) and effectiveness (learning gains) from solving intra-problem and inter-problem adaptive Parsons problems versus non-adaptive Parsons problem versus writing the equivalent code.

4.1 Participants

Undergraduate students were recruited from two sections of an introductory computer science course for computing majors at a research-intensive university in the United States. The sections had the same instructor and followed the same curriculum with the same homework and assessments. This course covers introductory programming concepts in Python including variables, selection, iteration, and lists. At the time of the study, the course had covered all of these topics and was covering files and dictionaries. Ericson visited the course during lecture to recruit participants and also sent an announcement to all of the students enrolled in the course. Participants earned 2.5 points of extra credit for completing the first session and another 2.5 points of extra credit for completing the second session one week later. Students who did not participate in the study could alternatively earn up to 5 points of extra credit by writing a paper on a computing innovation, which Ericson graded and that grade was submitted to the course instructors. None of the authors were involved in the teaching of the course.

4.2 Study Design

The first of two sessions took 2.5 hours and included consent, a demographic survey, pretest, instructional material, and an immediate posttest. The second session, lasting an hour one week later, was a delayed posttest to measure retention of the instructional material.

The instructional material in the first session contained four worked-example plus practice pairs. A worked example is a worked out expert solution to a problem [41]. Research has found that interleaving worked-examples with similar practice problems improves learning gains [42]. Students were randomly assigned to one of four practice conditions for the instructional material: 1) solving on-task adaptive Parsons problems with distractors, 2) solving on-task non-adaptive Parsons problems with distractors, 3) writing the equivalent code as the Parsons problems, or 4) a control group that solved off-task adaptive Parsons problems with distractors.

4.3 Study Procedure

Both sessions were held in a closed classroom with all participants attending at the same time. Students were instructed to bring their laptops and were provided with scratch paper and a pen. All of the study materials were online and students were asked to only use those materials, even though they had access to the Internet. Proctors checked that the students were on task.

In the first session, the procedure was 1) provide consent and randomly be placed into one of the four practice conditions, 2) complete the demographic survey, 3) complete the practice problems which familiarized the students with the online environment and problem types, 4) complete the pretest, 5) complete four worked examples plus practice pairs, where the type of practice problem

differed based on the condition, and 6) complete the immediate posttest.

At the second session, a week later, participants completed the delayed posttest, which was isomorphic to the first posttest. Only the variable names and some values were changed, but the structure of the problems was the same, meaning that they required near transfer to solve. Near transfer is being able to solve a new problem in a similar context to one that you have already solved. The delayed posttest tested for retention of the material one week later.

4.4 Study Materials

The study materials include a demographic survey, familiarization (practice) materials, a pretest, instructional material, an immediate posttest, and a delayed posttest.

4.4.1 Demographic Survey. The survey asked the participant's age, gender, race, first spoken language, comfort level with reading English, high school grade point average, college grade point average, current major, expected grade in the course, and prior programming experience. If participants reported prior programming experience, they were also asked what courses, where they took them, and how many years they had been programming.

4.4.2 Familiarization Activities. The participants in this study had not previously used the online study environment, so materials were developed to introduce them to the types of problems they would see in the pretest. This included instruction on how to start and finish a timed section (a section that must be completed in a given amount of time), how to move to the next page, how to answer multiple-choice questions, how to check the solution for a fix code or write code problem, and how to drag blocks and check the solution on a Parsons problem.

The familiarization activities also include two easy practice multiple-choice questions, a practice fix code problem, a practice Parsons problem, and a practice write code problem. The fix code problem included instructions on how to fix the code. The correct solution was displayed above both the Parsons and write code problems.

4.4.3 Pretest. The participants had 15 minutes to complete the first section of five multiple-choice questions and 10 minutes to complete each of the other three sections (fix code, Parsons problem, and write code). If the participant ran out of time, the current answer(s) were automatically recorded. The goal was to control the amount of time the learner had for each section.

The five multiple-choice questions required tracing code with lists, ranges, selection, and iteration. The questions included code to find the minimum value in a list between a range of indices as shown in Figure 2, compare values in two sorted lists, return the count of the number of times a target value appeared in a range of indices in a list, trace the values of variables in a complex for loop, and return the average of values in a range of indices in a list. The question that compared the values in two lists had been used in prior research with 65% of the those students getting it correct [32].

The fix code problem was intended to calculate and return the average of a list of numbers, but double the highest value. However, it had errors that the learner had to find and fix, as shown in Figure 3. The problem included unit tests, to verify the correctness of the code as shown in Figure 4.

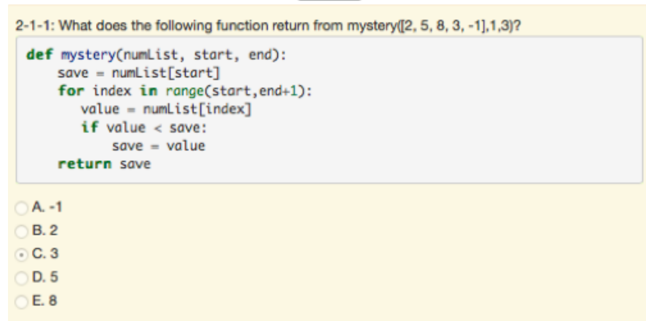


Figure 2: The first multiple-choice question in the pretest and immediate posttest

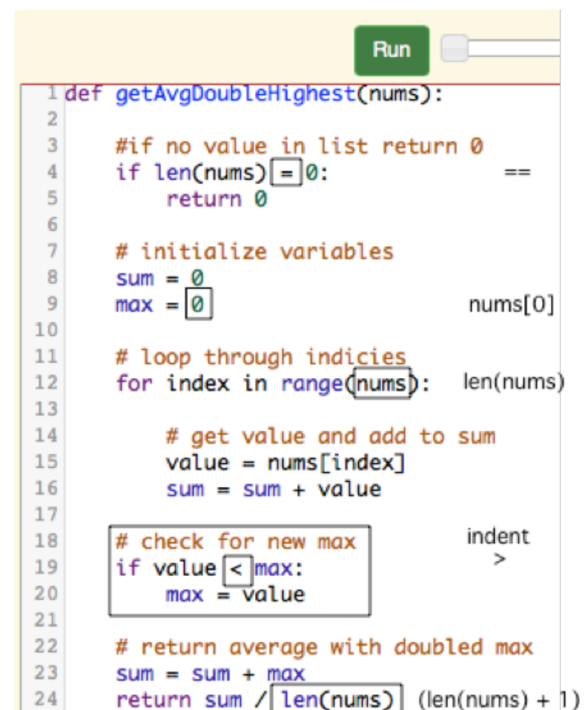


Figure 3: Pretest fix code problem with the errors boxed and the correct code to the right

Result	Actual Value	Expected Value	Notes
Pass	2	2	Test of getAvgDoubleHighest([1,1,3])
Pass	0	0	Test of getAvgDoubleHighest([])
Pass	3.0	3.0	Test of getAvgDoubleHighest([3.0,5.0,2.0,0])
Pass	-3.5	-3.5	Test of getAvgDoubleHighest([-3,-3,-5])

You passed: 100.0% of the tests

Figure 4: The unit test results when all the errors in the fix code problem have been corrected

Drop blocks here

```
def isLevel(elList, start, end):
    max = elList[start]
    min = max
    for index in range(start,end+1):
        value = elList[index]
        if value > max:
            max = value
        if value < min:
            min = value
    return (max - min) <= 10
```

Figure 5: The solution to the pretest Parsons problem

The Parsons problem asked the participant to order the code for a function, *isLevel*, which should return true if the difference between the maximum and minimum value between a given start and end index (inclusive) was 10 or less. This problem included five unpaired distractor code blocks, in which the distractor blocks were randomly mixed in with the correct code blocks. The solution to this problem is shown in Figure 5. Note that the user had to order the blocks vertically as well as indent the blocks horizontally to achieve a correct solution.

The write code problem was a modified version of Soloway's rainfall problem [10], which has been studied by several researchers [20, 23, 37, 38]. The original problem totals the non-negative values in an input loop until a sentinel value is reached and then outputs the average. The solution must avoid a division by zero. The problem was modified to loop through a list of numbers rather than read input until a sentinel value was reached as shown in Figure 6. Simon found that students still perform poorly on this problem and that students are not used to reading input in a loop until a sentinel value is reached [38]. The instructions explained the algorithm in English, provided example input and output, and provided unit tests.

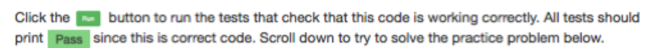
4.4.4 Instructional Material (Worked Example + Practice). There were four worked examples with interleaved practice problems in the instructional material. The type of practice problem depended on the condition: adaptive Parsons problems, non-adaptive Parsons problems, write code problems, or the control group which solved off-task adaptive Parsons problems on turtle graphics.

Each worked example contained an algorithm in English and example input and output. It also included runnable code with comments as shown in Figure 7. When the user ran the example code it displayed the results from running the unit tests.

```
1  # Write the getAverageRainfall function below
2  # It should sum all the non-negative values in
3  # the list rain and return the average which is
4  # the sum divided by the count of non-negative values
5  # if there are no non-negative values it should
6  # return 0
7  def getAverageRainfall(rain):
8      sum = 0
9      count = 0
10     for value in rain:
11         if value >= 0:
12             sum = sum + value
13             count = count + 1
14     if count > 0:
15         return sum / count
16     return 0
17
```

Figure 6: The write code problem with a correct solution

Run Code


Click the Run button to run the tests that check that this code is working correctly. All tests should print Pass since this is correct code. Scroll down to try to solve the practice problem below.

Run

```
1  # define the function
2  def getAverage(numList):
3
4      # prevent a divide by zero
5      if len(numList) == 0:
6          return 0
7
8      # init sum
9      sum = 0
10
11     # loop through indicies
12     for index in range(len(numList)):
13
14         # get value at index
15         value = numList[index]
16
17         # add value to sum
18         sum = sum + value
19
20     # return the average
21     return sum / len(numList)
22
23
```

Figure 7: A worked example with runnable code

Each of the practice problems also contained an algorithm in English, example input and output, and unit tests to verify the user's solution. The user had 10 minutes to complete each problem. The user answer was saved if the user ran out of time.

The first worked example returned a count of the number of times a target value appeared in a list using a loop that looped through all the indices. The associated practice question was to return the count of a target value in a given range of indices (inclusive). The second worked example returned the maximum value from a list and the associated practice problem was to return the minimum value. The third worked example returned the average of the values in a list and protected against a divide by zero error as shown in Figure 7. The associated practice problem returned the average but did not include the lowest value in the list in the average and also guarded against a divide by zero error as shown in Figure 8. The fourth worked example returned the minimum value in a given range of indices (inclusive). The associated practice

Drop blocks here

```
def getAverageDropLowest(numList):
    if len(numList) == 0:
        return 0
    sum = 0
    lowest = numList[0]
    for index in range(len(numList)):
        value = numList[index]
        sum = sum + value
        if value < lowest:
            lowest = value
    return (sum - lowest) /
            (len(numList) - 1))
```

Figure 8: The correct solution to the third instructional Parsons problem

2-1-4: What do a and b equal after the following code executes?

```
a = 10
b = 3
t = 0
for i in range(1,4):
    t = a
    a = i + b
    b = t - i
```

A. a = 5 and b = -2
B. a = 6 and b = 7
C. a = 6 and b = 3
D. a = 12 and b = 1
E. a = 5 and b = 8

Pretest and Posttest

4-1-4: What do x and y equal after the following code executes?

```
x = 7
y = 1
z = 0
for i in range(1,4):
    z = x
    x = i + y
    y = z - i
```

A. x = 8 and y = 0
B. x = 9 and y = -1
C. x = 2 and y = 6
D. x = 5 and y = 3
E. x = 3 and y = 5

Second (delayed) posttest

Figure 9: One of the multiple-choice questions from the pretest, first posttest, and second (delayed) posttest. Note that the second posttest changed the variable names and values.

problem returned the maximum value in a given range of indices (inclusive).

4.4.5 Posttests. The immediate posttest, which was administered at the end of the first session, had the exact same questions as the pretest. The delayed posttest, administered one week later, was isomorphic to the immediate posttest, meaning that the problems to be solved had the same structure, but different surface level features, like variable names as shown in Figure 9.

5 ANALYSIS

A total of 163 students participated in the first session. However, 37 of these students did not answer at least one question during the session or spent less than 30 seconds answering a question without getting the question correct. This paper reports on the data from the remaining 126 students (32 in the adaptive Parsons condition, 34 in the non-adaptive Parsons condition, 27 in the write condition, and 33 in the control group that solved off-task adaptive Parsons problems) from the first session.

Students were not required to come back for the second session one week later, but earned an additional 2.5 points of extra credit for completing this session. A total of 126 students returned for the second session. Of these, 100 students completed all the questions in both the first session and second session and spent at least 30 seconds on each question or got the question correct in under 30 seconds (27 in the adaptive Parsons condition, 30 in the non-adaptive Parsons condition, 19 in the write condition, and 24 in the control group that solved off-task adaptive Parsons problems). These 100 students were used to study the retention of the material one week later.

Table 1: Mean time in seconds and standard deviation for each of the four practice problems by group (condition)

Group	P1 secs (std dev)	P2 secs (std dev)	P3 secs (std dev)	P4 secs (std dev)
1. (*n=32*) A. Parsons	115.65 (50.1)	97.88 (34.3)	191.88 (130.5)	74.63 (23.5)
2. (*n=34*) Parsons	114.29 (56.3)	92.85 (31.0)	190.79 (91.2)	72.94 (26.8)
3. (*n=27*) Write	177.44 (152.0)	118.07 (113.3)	270.48 (152.0)	102 (63.6)
4. (*n=33*) Control	252.24 (100.9)	176.70 (79.6)	178.12 (107.13)	325.06 (160.3)

5.1 Testing for Efficiency

The mean time in seconds to complete each practice problem and the standard deviation is shown in Table 1 for each condition. Note that the adaptive Parsons (group 1) and non-adaptive Parsons (group 2) had similar mean completion times. In an observational study of teachers solving both adaptive and non-adaptive Parsons problems, the adaptive problems sometimes took longer to solve than the non-adaptive because the teacher checked their solution after each change. With inter-problem adaptation, if the learner struggled on the previous problem then the next problem was made easier and if the learner solved the previous problem in one attempt the next problem was made harder, which probably kept the total completion times similar. The Cohen's d for the write code group mean total time to solve the practice problems in seconds compared to the adaptive Parsons group is (*d=0.756*) and for the non-adaptive Parsons group is (*d=0.845*). Cohen describes 0.2 as a small effect size, 0.5 as a medium effect size, and 0.8 as a large effect size [14]. There was a large effect between the non-adaptive Parsons and the write code group and a medium effect between the adaptive Parsons and write code group.

To test if the time differences were significant, outliers were removed (values more than three standard deviations from the mean) to normalize the data so that z-scores could be calculated.

Z-scores allow for different size groups to be compared. A test for skew (a test to indicate whether or not the data falls in a normal distribution) revealed that the values were all in the acceptable range (under 2). Removing outliers left 31 students in the adaptive Parsons group, 33 in the non-adaptive Parsons group, and 22 in the write group. Z-scores were created from the total time in seconds to solve the four practice problems minus the mean and divided by the standard deviation. A Least Squares Difference test (LSD) was used to compare the three on-task groups (Adaptive Parsons, Parsons, and Write). The fourth condition was the control group, which was solving off-task problems, so it was not included in the test for efficiency. There was no significant difference in completion time between the adaptive Parsons group and non-adaptive Parsons group. The time was significantly different between the adaptive Parsons group and the write group (mean difference of -.33 and p=.025) as well as the non-adaptive Parsons group and the write group (mean difference of -.32 and p=.025).

5.2 Testing for Effectiveness

Grading rubrics were created for the pretest and posttest write and fix code problems. Two people graded each problem independently and then met to resolve any differences in scores. The hand graded scores correlated with the unit test results. A factor analysis showed that the hand graded scores and unit test scores appeared to be measuring the same construct.

The Parsons problems were graded automatically. Each correct line in the correct order starting from the beginning of the solution received a half point. If the correct line was also indented correctly it received an additional half point. If the line was incorrect, but was the paired distractor and was indented correctly, it received a half point. Grading continued until a line was found that was neither the correct line nor its paired distractor (i.e. a line out of order). Grading then continued from the end of the solution back towards the first line that had been found to be incorrect.

This grading approach was based on an observation that learners had the most difficulty in the middle of the solution. We also wanted the grading to be similar to the grading of the fix code problems, and the fix code problems had the advantage that the code was already in the correct order.

The mean and standard deviation for each pretest and immediate posttest are shown by condition in Table 2. The pretest and posttest both contained four sections. One section had five multiple-choice (MC) questions which a maximum score of five, one had a fix code problem with a maximum score of 11, one had a Parsons with a maximum score of 10, and one had a write code problem with a maximum score of 10. The Cohen's d for the differences in the gains from pretest to immediate posttest by condition versus the control group were: *(d=0.799)* for the adaptive Parsons group, *(d = 0.311)* for the non-adaptive Parsons, and *(d=0.133)* for the code writing group. This is a medium effect size for the adaptive Parsons group and a small effect size for the non-adaptive Parsons group.

Remember that not all of the students took the delayed (2nd) posttest one week later. The mean score and standard deviation for the pretest, immediate (1st) posttest, and delayed (2nd) posttest for just the students who attended both sessions is shown in Table 3.

Table 2: Mean score and standard deviation for the pretest and immediate posttest (first posttest) by group

	Group 1 A. Parsons *(n=32)*	Group 2 Parsons *(n=34)*	Group 3 Write *(n=27)*	Group 4 Control *(n=33)*
Pre MC	2.7 (1.5)	3 (1.1)	3.8 (1.4)	3.6 (1.4)
Post MC	3.8 (1.1)	3.4 (.17)	4.3 (1.3)	4.2 (1.2)
Pre Fix	8.1 (1.6)	8.9 (2.0)	9.0 (1.6)	8.8 (1.8)
Post Fix	9.2 (1.8)	9.6 (2.0)	9.8 (1.8)	8.8 (2.1)
Pre Order	7.3 (3.3)	8.6 (3.0)	7.7 (3.4)	7.4 (3.6)
Post Order	8.5 (3.0)	9.5 (1.8)	8.0 (3.3)	7.9 (3.5)
Pre Write	8.6 (2.3)	9.3 (1.3)	9.0 (1.7)	9.2 (1.2)
Post Write	9.3 (1.3)	9.4 (1.0)	9.0 (1.9)	9.2 (1.4)

Table 3: Mean score and standard deviation for the pretest, immediate posttest (first posttest), and delayed posttest (2nd posttest) by group

	Group 1 A. Parsons *(n=27)*	Group 2 Parsons *(n=30)*	Group 3 Write *(n=19)*	Group 4 Control *(n=24)*
Pre MC	2.7 (1.5)	2.9 (1.1)	3.8 (1.3)	3.8 (1.5)
1st Post	3.9 (1.0)	3.7 (1.6)	4.2 (1.5)	4.3 (1.3)
2nd Post	3.8 (1.1)	3.7 (1.2)	4.3 (1.1)	3.8 (1.2)
Pre Fix	8.1 (1.6)	8.9 (2.0)	8.9 (1.6)	8.6 (1.9)
1st Post	9.3 (1.7)	9.6 (2.0)	9.4 (2.0)	8.8 (2.2)
2nd Post	9.1 (1.8)	9.7 (1.9)	9.5 (1.8)	9.3 (1.6)
Pre Order	7.7 (3.1)	8.1 (3.3)	6.8 (3.7)	7.2 (3.8)
1st Post	8.2 (3.1)	9.4 (2.0)	7.1 (3.7)	7.4 (3.7)
2nd Post	8.7 (2.4)	9.7 (1.4)	9.2 (2.0)	8.3 (2.9)
Pre Write	8.9 (1.9)	9.4 (1.4)	8.7 (2.0)	9.1 (1.3)
1st Post	9.4 (1.2)	9.5 (1.1)	8.7 (2.2)	9.0 (1.5)
2nd Post	9.3 (1.1)	9.7 (1.0)	9.0 (2.1)	9.3 (1.3)

There was a statistically significant change from pretest to the immediate posttest using a multivariate analysis of variance (MANOVA) with Pillai's trace *(F=2.36 and p=.031)*. A MANOVA was used since there were four conditions in this study. The Bonferroni post-hoc test does not indicate a statistically significant difference by condition from the pretest to the immediate posttest, which means that no condition seemed to have done better or worse than the others.

The differences in the scores from pretest to immediate posttest were compared for all the on-task conditions compared to the control group. Twenty-seven results were chosen at random from each condition in order to compare equal size samples. A Mann-Whitney U test was used, which does not assume that the data follows a normal distribution. The difference between the pretest score and the posttest score was significant *(p=.007882)* for the adaptive condition versus the control group (who solved turtle graphics problems). There was not a significant difference for any of the other on-task

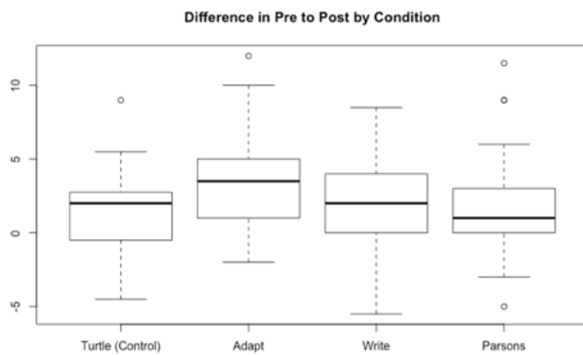


Figure 10: Results from a Mann-Whitney U test comparing the pretest score to the immediate posttest score by condition

conditions compared to the control group. See Figure 10 for a box and whisker plot by condition.

Kurtosis was high on the pretest write and posttest write problems which indicates that the scores fall in a narrow range. This means there was likely a ceiling effect on the write code problem. The mean scores on the pretest write problem ranged from 8.7 to 9.4 out of a maximum of 10 as shown in Table 3.

5.3 Analysis of the Demographic Information

Of the 126 students who completed all questions in the first session, 73 (58%) self-identified as male and 51 (40%) as female and two (2%) students did not answer the question.

An analysis of the demographic information showed a strong positive correlation with the student's actual grade in the course *(p < .001)*. There was a moderate negative correlation between the pretest score and the student's age $r(121) = -.413$, $p < .001$, which means that older students did worse than younger. This course is intended to be a first course for majors, so older students may be retaking the course after failing it in the past, or be weaker students who delayed taking the course. We found a moderate negative correlation for gender with males performing better than females on the delayed posttest $\rho(99) = -.362$, $p < .001$.

There was no interaction between condition and any of the demographic characteristics that affected performance. This means that the groups were comparable.

6 DISCUSSION

The on-task Parsons groups (both the adaptive and non-adaptive) solved the four practice problems in significantly *(p=.025)* less time than the write code group with medium (adaptive Parsons group) *(d=0.756)* to large (non-adaptive Parsons group) (*d=0.845*) effect sizes. This supports hypothesis **H1**.

There was a significant improvement in composite scores from pretest to immediate posttest. However, there was no significant difference between the three on-task conditions. This means that learners solving both adaptive and non-adaptive Parsons problems had equivalent learning gains as those in the write code condition. This supports hypothesis **H2**.

The learners in the control group who solved off-task turtle graphics adaptive Parsons problems had a significantly *(p=.007882)* lower learning gain from pretest to immediate posttest than the on-task adaptive Parsons group and there was a medium effect size *(d=0.799)*, which supports hypothesis **H3**. However, there was no significant difference between the control group and the non-adaptive Parsons group or between the control group and the write code group. This means that hypothesis **H3** was not fully supported.

7 LIMITATIONS

There was not any significant difference for the learning gains from the pretest to the immediate posttest between the control group and the non-adaptive Parsons group or the write code group. This implies that at least some of the learning gains were from answering the same or similar problems with correctness feedback.

This study used both intra-problem and inter-problem adaptation. We do not know the relative effectiveness of each type of adaptation. Further studies should be done to test this.

The results are only from undergraduates from one research intensive university in the United States. The results would be strengthened by replication at other universities around the world and by similar studies with teachers.

8 CONCLUSIONS

This study provides evidence that solving either adaptive Parsons problems or non-adaptive Parsons problems is a more efficient, but just as effective, form of practice than writing the equivalent code. It also found that solving adaptive Parsons problems led to a significant learning gain compared to the control group. However, there was no significant difference in learning gains between the control group and either the non-adaptive Parsons problems group or the write code group. This means that at least some of the learning gains were likely due to repeated exposure to the same problems with correctness feedback. Further studies are needed to verify the learning gains from solving adaptive Parsons problems versus non-adaptive Parsons problems versus writing the equivalent code. If solving adaptive Parsons problems is a more efficient, but just as effective form of low cognitive load practice, they could be used to help prepare thousands of new computing teachers and reduce the time that it takes to learn to program. The Parsons software is freely available as part of the Runestone Interactive platform [33].

ACKNOWLEDGMENTS

This study was supported by the National Science Foundation under grants 1138378 and 1432300. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank all the people who assisted in this research. Lauren Margulieux conducted most of the statistical tests. Matthew Guzdial ran the Mann-Whitney U test. Matt Lord graded the fix and write code problems. We also thank the reviewers for their helpful comments.

REFERENCES

[1] John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. 1989. Skill Acquisition and the LISP Tutor. *Cognitive Science* 13 (1989), 467–505.

[2] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger, and Ray Pelletier. 1995. Cognitive Tutors: Lessons Learned. *Journal of the Learning Sciences* 4, 2 (1995), 167–207. https://doi.org/10.1207/s15327809jls0402_2

[3] Owen Astrachan, Jan Cuny, Chris Stephenson, and Cameron Wilson. 2011. The CS10K project: mobilizing the community to transform high school computing. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 85–86.

[4] Ronald Baecker and Ian Small. 1990. Animation at the interface. *The art of human-computer interface design* (1990), 251–267.

[5] Tim Bell, Peter Andreae, and Lynn Lambert. 2010. Computer science in New Zealand high schools. In *Proceedings of the Twelfth Australasian Conference on Computing Education-Volume 103*. Australian Computer Society, Inc., 15–22.

[6] Klara Benda, Amy Bruckman, and Mark Guzdial. 2012. When Life and Learning Do Not Fit: Challenges of Workload and Communication in Introductory Computer Science Online. *Trans. Comput. Educ.* 12, 4 (2012), 1–38. https://doi.org/10.1145/2382564.2382567

[7] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *SIGCSE Bull.* 39, 2 (2007), 32–36. https://doi.org/10.1145/1272848.1272879

[8] Laura E. Berk and Adam Winsler. 1995. *Scaffolding Children's Learning: Vygotsky and Early Childhood Education*. National Association for the Education of Young Children.

[9] Elizabeth L Bjork and Robert A. Bjork. 2011. Making things hard on yourself, but in a good way: Creating desirable difficulties to enhance learning. *Psychology and the real world: Essays illustrating fundamental contributions to society* (2011), 56–64.

[10] Benedict Du Boulay. 1988. *Some Difficulties of Learning to Program*. Lawrence Erlbaum Associates, 283–299.

[11] Neil C. C. Brown, Sue Sentance, Tom Crick, and Simon Humphreys. 2014. Restart: The Resurgence of Computer Science in UK Schools. *Trans. Comput. Educ.* 14, 2 (2014), 1–22. https://doi.org/10.1145/2602484

[12] Michael E Caspersen and Palle Nowack. 2013. Computational thinking and practice: A generic approach to computing in Danish high schools. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*. Australian Computer Society, Inc., 137–143.

[13] Fanny Chevalier, Nathalie Henry Riche, Catherine Plaisant, Amira Chalbi, and Christophe Hurter. 2016. Animations 25 years later: new roles and opportunities. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*. ACM, 280–287.

[14] Jacob Cohen. 1988. Statistical power analysis for the behavioral sciences. 2nd.

[15] Gemma Corbalan, Liesbeth Kester, and Jeroen JG Van Merriënboer. 2008. Selecting learning tasks: Effects of adaptation and shared control on learning efficiency and task involvement. *Contemporary Educational Psychology* 33, 4 (2008), 733–756.

[16] Albert T Corbett, Kenneth R Koedinger, and John R Anderson. 1997. Intelligent tutoring systems. *Handbook of human-computer interaction* 5 (1997), 849–874.

[17] Tom Crick and Sue Sentance. 2011. Computing at school: stimulating computing education in the UK. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*. ACM, 122–123.

[18] Jan Cuny, Diane A Baxter, Daniel D Garcia, Jeff Gray, and Ralph Morelli. 2014. CS principles professional development: only 9,500 to go!. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 543–544.

[19] 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.

[20] Alireza Ebrahimi. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies* 41 (1994), 457–480.

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

[22] 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.

[23] Kathi Fisler. 2014. The recurring rainfall problem. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 35–42.

[24] Judith Gal-Ezer, Catriel Beeri, David Harel, and Amiram Yehudai. 1995. A high school program in computer science. *Computer* 28, 10 (1995), 73–80.

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

[26] 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.

[27] 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.

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

[29] Paivi Kinnunen and Beth Simon. 2010. Experiencing programming assignments in CS1: the emotional toll. In *Proceedings of the Sixth international workshop on Computing education research*. ACM, 77–86.

[30] Amruth N Kumar. 2018. Epplets: A Tool for Solving Parsons Puzzles. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 527–532.

[31] Karen Lang, Ria Galanos, Joanna Goode, Deborah Seehorn, Fran Trees, Pat Phillips, and Chris Stephenson. 2013. *Bugs in the System: Computer Science Teacher Certification in the U.S.* Report. The Computer Science Teachers Association The Association for Computing Machinery.

[32] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 119–150.

[33] Brad Miller and David Ranum. 2014. Runestone interactive: tools for creating interactive course materials. In *Proceedings of the first ACM conference on Learning@ scale conference*. ACM, 213–214.

[34] 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.

[35] Andrew Petersen, Michelle Craig, and Daniel Zingaro. 2011. Reviewing CS1 exam question content. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. ACM, 631–636.

[36] Barbara Rogoff. 1990. *Apprenticeship in thinking: Cognitive development in sociocultural activity*. Oxford University Press, New York, NY, USA.

[37] Otto Seppälä, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen. 2015. Do we know how difficult the rainfall problem is?. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*. ACM, 87–96.

[38] Simon. 2013. Soloway's Rainfall Problem has become Harder. In *Learning and Teaching in Computing and Engineering*. IEEE Computer Society, Washington DC, USA, 130–135.

[39] Elliot Soloway. 1986. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM* 29, 9 (1986), 850–858. http://dl.acm.org/citation.cfm?doid=6592.6594

[40] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[41] John Sweller and Graham Cooper. 1985. The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra. *Cognition and Instruction* 2, 1 (1985), 59–89.

[42] John Gregory Trafton and Brian J. Reiser. 1993. The contributions of studying examples and solving problems to skill acquisition. In *15th Annual Conference of the Cognitive Science Society*. Lawrence Erlbaum Associates, Inc., 1017–1022. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.9933&rep=rep1&type=pdf

[43] 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.

[44] Jeroen JG Van Merriënboer, Paul A Kirschner, and Liesbeth Kester. 2003. Taking the load off a learner's mind: Instructional design for complex learning. *Educational psychologist* 38, 1 (2003), 5–13.