Solving Parsons Problems Versus Fixing and Writing Code

Barbara J. Ericson
School of Interactive Computing
Georgia Institute of Technology
Atlanta, GA, 30332
USA
ericson@cc.gatech.edu

Lauren E. Margulieux
Learning Technologies Division
Georgia State University
Atlanta, GA, 30303
USA
lmargulieux @gsu.edu

Jochen Rick
School of Interactive Computing
Georgia Institute of Technology
Atlanta, GA, 30332
USA
jochen.rick@gatech.edu

ABSTRACT

Prior research has shown that Parsons problems are an engaging type of code completion problem that can be used to teach syntactic and semantic language constructs. They can also be used in summative assessments to reduce marking time and grading variability compared to code writing problems. In a Parsons problem the correct code is provided, but is broken into mixed-up code blocks that must be assembled in the correct order. Two-dimensional Parsons problems also require the code blocks to be indented correctly. Parsons problems can contain extra code blocks, called distractors, which are not needed in a correct solution. We present a study that compared the efficiency, effectiveness, and cognitive load of learning from solving two-dimensional Parsons problems with distractors, versus fixing code with the same errors as the distractors, versus writing the equivalent code. We found that solving two-dimensional Parsons problems with distractors took significantly less time than fixing code with errors or than writing the equivalent code. Additionally, there was no statistically significant difference in the learning performance, or in student retention of the knowledge one week later.

CCS CONCEPTS

• Social and Professional Topics~Computing Education; Social and professional topics~Student assessment

KEYWORDS

Parsons problems, Parsons programming puzzles, code-competition problems, cognitive load, assessment

1 INTRODUCTION

Learning to program is difficult. Students spend many frustrating hours trying to figure out why their programs don't compile [2]. Drop out and failure rates in many introductory college-level computing classes are high with an average pass rate worldwide of only 67% [3, 54], perhaps in part due to this frustration. Several multi-institution and multi-national studies have found that students perform poorly on tests of their knowledge [28, 30, 47, 51], suggesting that the current instructional strategies do not lead to effective learning.

Beginning programming students have to learn many things. They have to develop a mental model of the computer (the notional machine), the notation (syntax and semantics), the structures (programming schemas and plans), and skill in planning, developing, and debugging programs [5]. Piaget popularized the term *schema*, which is a mental framework for organizing and applying knowledge [52]. Experts have a large number of robust schemas that they can use to recognize and solve similar problems [1, 55]. It can take 10 years of sufficient and sustained practice to turn a novice programmer into an expert programmer [55].

In introductory college-level programming courses students are mostly expected to practice by writing code, which can take a large and unpredictable amount of time. Students have reported spending hours trying to fix a simple syntax error like a comma out of place [2]. Despite its difficulties, writing code is common in an introductory programming course because it is an *authentic task*. An *authentic task* is one that someone in the field of study might encounter in their work [40]. Constructivists encourage the use of authentic tasks to motivate students [13]. However, without appropriate guidance, students can easily be overwhelmed by the complexity of an authentic task, especially when they have little knowledge about the domain [33]. Cognitive load theory (CLT) states that the human mind has limited processing capability and that the cognitive load of complex tasks must be reduced in order for learning to occur [45].

One of the recommended approaches to reducing cognitive load is to use completion tasks rather than whole tasks [31, 32]. An example of a completion task is modifying or extending a program, rather than writing a program from scratch. *Parsons problems* are a type of code completion practice problem in which the learner must place blocks of mixed up program code in the correct order. Some types of Parsons problems, called *two-dimensional (2D) Parsons problems*, also require the code to be indented correctly. Parsons problems can also have *distractor* code blocks that are not needed in the correct solution. The distractor blocks can include syntactic errors like a missing colon as well as semantic errors like the wrong boundary condition on a loop. The distractor blocks can be randomly mixed in with the correct blocks. Alternatively, each distractor can be shown *paired* with the matching correct code block so that the learner only has

Koli Calling 2017, November 16–19, 2017, Koli, Finland

ACM ISBN 978-1-4503-5301-4/17/11...$15.00
https://doi.org/10.1145/3141880.3141895

to choose the distractor or the correct code, as shown in Fig. 1. Parsons problems can be used to teach syntactic and semantic language constructs as well as expose students to common programming plans [37]. Several researchers have hypothesized that solving Parsons problems should result in more effective and efficient learning than having students write the equivalent code [12, 37], but to our knowledge none have empirically tested that hypothesis.

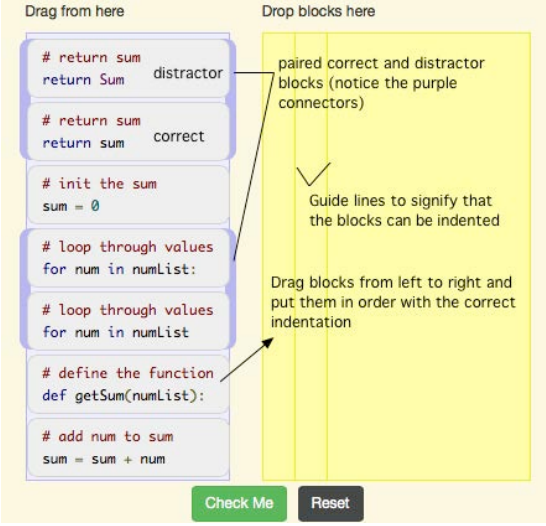

Figure 1: A 2D Parsons problem with paired distractors

Our research questions were:

1. What are the relative effects on performance of 1) solving two-dimensional Parsons problems with paired distractors, 2) fixing the equivalent code with the same distractors, and 3) writing the equivalent code?
2. What is the effect on completion time for each of the three conditions (solving Parsons Problems, fixing, and writing code)?
3. What is their effect on self-reported cognitive load?

2 RELATED WORK

This research is based on several theories from educational psychology including constructivist learning and cognitive load theory. It also incorporates research findings on worked examples, deliberate practice, desirable difficulties, and subgoal labeling. This study was informed by prior research on Parsons problems.

2.1 Constructivist Learning

In Piaget's theory of constructivist learning learners must construct their understanding by making sense of the information and by building a mental representation [52]. Chi found that constructive learning leads to better outcomes than passive or active learning [9]. Solving Parsons problems with distractors should help the learner construct an understanding of common errors and algorithms as she or he selects blocks and places them in the correct order [37].

2.2 Cognitive Load Theory

Cognitive Load Theory (*CLT*) was developed by John Sweller in the late 1980s [44]. For learning to occur new information must be processed in working memory and then added to the knowledge representations (schemas) that exists in long-term memory [6]. However, working memory has a limited capacity, and if that capacity is needed entirely to process new information, it cannot be used to modify or build schemas. Instructional material can be designed to reduce the cognitive load that is devoted to processing new information. It is important to note that the amount of cognitive load a learner experiences is based on three components: the complexity of the material or task, the way the instruction is designed, as well as the strategies used for constructing knowledge. The complexity of the material or task varies with the learner's prior knowledge. Parsons problems, as a type of code completion problem, should have a lower cognitive load than a problem that requires the learner to write the code from scratch, because the problem space is more constrained.

2.3 Worked Examples

One of the most well known effects predicted by cognitive load theory is the *worked example* effect. A *worked example* is a detailed description or demonstration of how to solve a problem, including both the problem statement and all steps of the problem solution. Sweller proposed a "*Borrowing and Reorganizing Principle*" which means that the way humans build long-term knowledge is by imitating others [46]. Studies have shown that worked examples improve learning in algebra, physics, and programming [10, 38, 53, 56]. However, students don't always learn from worked examples [15]. They learn best when the worked examples are interleaved with practice problems that are similar to the worked examples [48]. Another argument in favor of worked examples is that students prefer learning by studying examples rather than learning by reading text [26]. In this study, the instructional material contains four worked examples with interleaved practice problems.

2.4 Subgoal Labeling

One of the reasons experts perform better than novices is that they can recognize structural similarities on tasks or problems, while novices tend to focus on surface level features [6]. For instance, when students are given a worked example about calculating the average rainfall, they typically focus on surface features, like the variable names, rather than the structural process of the solution. This makes it difficult for them to transfer their knowledge to other similar problems (e.g., calculating the average score for a range of indices in a list) and seemingly different, but actually structurally similar, problems (e.g., counting the number of target values in a list). The subgoal learning framework addresses this problem by drawing students' attention to the structural features to improve their problem solving performance and transfer [7, 29].

Subgoal labeling is a method of teaching subgoal learning through worked examples [7, 29]. Subgoal labeled worked examples visually group subgoals of the problem solution (i.e., functional pieces of the problem solution) and give them a meaningful label that describes their function. For example, in Fig. 1, the first block on the left includes the subgoal label "return sum" as a comment. This label describes the function of the line below it. All of the worked examples, fix code problems, and Parsons problems in this study used subgoal labels in the comments to help novices focus on the structure of the solution rather than just the surface level features. In the paired distractor Parsons problems the distractor blocks contained the same subgoal labels as the correct code blocks to further indicate that the blocks were paired and that the student should only choose one of the pair as shown in Fig. 1.

2.5 Deliberate Practice

Practice is essential for learning [6]. It helps the learner focus on, organize, integrate, and retrieve new knowledge from long-term memory. Several studies show the importance of practice in developing expertise [42, 50]. But, it needs to be the right kind of practice. It is possible to spend many hours practicing without any improvement in ability.

To improve performance it needs to be *deliberate practice* which means that it focuses on areas where the learner is weaker and it must include feedback, which can be used to improve results [19]. Parsons problems can be used to focus learning on areas that learners typically struggle with, such as recognizing common syntax errors. Parsons problems can include distractor blocks that contain common syntactic or semantic errors, which should help novices learn to recognize those errors with less frustration than encountering them when they are programming a solution from scratch [37].

2.6 Desirable Difficulties

New information is not just stored or copied into long-term memory; it is related to and integrated into what learners already know [6]. Retrieval of information depends heavily on the context, which can limit our ability to transfer information from one context to another [6]. Retrieving information from long-term memory increases our ability to recall it in the future. *Desirable difficulties* are those that help learners store and recall information in multiple contexts [4]. One key idea of this work is that improving the learner's performance while learning can actually decrease long-term learning, and conversely techniques that reduce the learner's performance while learning can actually lead to long-term retention and better recall. One technique that promotes desirable difficulties is spaced practice over time rather than massed practice [14, 39]. In this study we use paired distractors to increase the difficulty of the Parsons problems and to help the user learn to identify common errors [12].

2.7 Parsons Problems

Researchers have studied several variants of Parsons problems. They have used different names for them such as Parson's programming puzzles [37], Parson's puzzles [24, 25], and Mangled code [8]. Dale Parsons, for whom they are named, has said that Parson's was a mistake since her last name is Parsons. We use Parsons problems, which is consistent with other researchers [12, 23].

2.7.1 Evidence that Parsons Problems are Engaging

One of the reasons we choose to study Parsons problems is that there is evidence that users find them engaging. Dale Parsons and Patricia Haden originally created Parsons problems with syntactic distractors to provide an engaging way to help novices master syntax [37]. Most undergraduate students (82%) in their study (n=17) reported that the Parsons problems were useful or very useful for learning Pascal on a post survey.

Ericson has provided further evidence that learners find Parsons problems engaging. She added two-dimensional Parsons problems to interactive ebooks and found that more students and teachers attempted to solve the Parsons problems than tried to solve the nearby multiple choice questions after a worked example [17, 18]. In an online feedback form, teachers mentioned Parsons problems as being valuable at twice the rate of multiple-choice questions or fill in the blank questions [18].

2.7.2 What Makes Parsons Problems Difficult?

Researchers have studied several variants of Parsons problems and reported on what makes them easier or harder. This information was used to design the Parsons problems for this study.

Garner's small study (n=8) found evidence that Parsons problems with only the correct code and no distractors were easiest to solve and that problems that provided some of the correct code, but also required the solver to write some code were the hardest [20]. This provides some evidence that Parsons problems might have a lower cognitive load than writing code.

Harms, Chen, and Kelleher reported that middle school students (n=92) had less success, took significantly longer to solve problems, and reported a higher cognitive load during the training phase when solving Parsons problems with distractors versus solving Parsons problems without distractors [23].

Ihantola and Karavirta found that two-dimensional Parsons problems, in which the blocks must also be indented correctly, are more difficult than one-dimensional Parsons problems, which do not require indentation [25].

Denny, Luxton-Reilly, and Simon explored paper-based Parsons Problem as a possible replacement for requiring students to write code on exams [12]. They argued that Parsons problems would be quicker to grade and result in more consistent grades between markers. They tested several variants of Parsons problems. During think aloud observations they found evidence that Parsons problems with randomized distractors for nearly every line of code were too difficult for the students. Parsons problems with visually paired distractors were easier. Providing the structure of the code, the number of statements in a block indicated by curly braces and the indention, also made Parsons problems easier to solve.

Denny et al. also had 74 undergraduate students solve a Parsons problem, a write code problem, and a trace code problem on a paper-based exam [12]. The Parsons problem was a paired distractor Parsons problem. The pairing was indicated by extra space above and below each pair of distractor and correct code. The students had to write the code in the correct order and add the curly braces to indicate the block structure. They found a notable correlation (Spearman's ρ of .53), between the score on the write code and Parson problem. The lowest quartile of students did the worst on tracing code, better on writing code, and better still on solving Parsons problems.

Cheng and Harrington's large scale study (n=473) also investigated using a variant of Parsons problems, that they called a *Code Mangler* question, on an exam [8]. They found that the *Code Mangler* problem took less time to score than the equivalent write code problem and that the teaching assistants felt more confident that the grading was easier and consistent. They also found a notable correlation (Spearman's ρ of .6457), between the students score on the write code problem and the Code Manger (Parsons) problem.

Morrison et al. provided evidence that Parsons problems are a more sensitive measure of learning, i.e., that a Parsons problem might detect a learning difference between students that might not appear in a code writing problem [36].

Harms, Rowlett, and Kelleher compared solving Parsons problems with only correct code to following tutorials in Looking Glass [22]. They measured learning, cognitive load, and transfer. They found that the Parsons problem solvers complete the learning task more quickly (23% less time) than tutorial takers and also did 26% better on transfer tasks. The Parsons problem solvers also reported higher mental effort to complete the task than the tutorial followers, which is consistent with the idea that *desirable difficulties* lead to increased learning [4].

This study uses two-dimensional Parsons problems with visually paired distractors in an effort to provide desirable difficulty, but not excessive difficulty, in order to enhance learning.

3 SOFTWARE DEVELOPMENT

We used the Runestone Interactive platform to create and serve our study materials [34]. Our research team added the js-parsons software developed by Ihantola and Karavirta [25] to the Runestone platform in 2012. This software originally supported one and two-dimensional Parsons problems with distractor blocks randomly mixed in with the correct code blocks. We made the following changes: 1) added guidelines to signify that indentation was allowed as shown previously in Fig. 1, 2) allowed distractors to be displayed paired with the correct code blocks with purple edge decorations as shown in Fig. 1, and 3) allowed the specification of the display order for the blocks to guarantee consistency for experiments.

We also modified the Runestone Interactive platform to support timed exams, which have a maximum time limit and can only be taken once. The user must click a button to start the exam and a button to finish the exam and both of these events are logged. If the participant doesn't complete the exam in the specified time, the exam will automatically end and all current answers will be logged. This prevents students from spending too much time on a problem when they had no idea how to solve it.

4 STUDY PURPOSE

While several researchers have hypothesized that solving Parsons problems could result in more efficient learning than writing the equivalent code [12, 37], none to our knowledge have empirically tested this assumption. While some researchers have found a notable correlation between scores on Parsons problems and performance on different write code problems, these studies have not compared groups solving the same problems. In addition, no researchers have compared solving two-dimensional Parsons problems with paired distractors to fixing code with the same errors as the distractors. Since the learner doesn't have to type the code while solving either Parsons problems or fix code problems, they could have similar completion times.

The purpose of the study was to investigate the efficiency, effectiveness, and cognitive load of learning from solving two-dimensional Parsons problems with paired distractors, versus fixing code with the same distractors as errors, versus writing the equivalent code.

Our hypotheses were that 1) Parsons problems would be more efficient (take less time to solve) than fixing code with errors or writing the equivalent code. 2) Parsons problems would lead to at least the same amount of learning and retention as fixing or writing code, and 3) students in the Parsons problem condition would report lower cognitive load than students in the fix code or write code conditions.

5 STUDY DESIGN

This was a between subjects design, with one pretest and two posttests. There were two sessions in the study. The first session was 2.5 hours and included consent, a demographic survey, pretest, instructional material, a cognitive load survey, and a posttest. The second posttest, which lasted one hour and was held one week later, was administered to measure retention of the instructional material.

The instructional material in the first session contained four worked-example and practice pairs. Students were randomly assigned to one of three practice conditions for the instructional material: 1) solving two-dimensional Parsons problems with paired distractors, 2) fixing code with the same errors as the distractors, or 3) writing the equivalent code. The instructional practice condition was the independent variable. The dependent variables were the performance on the pretest and posttests, the time spent on each practice problem, and the cognitive load survey results.

6 STUDY PROCEDURES

This study consisted of two separate sessions one week apart. The sessions were held in a closed classroom with all participants attending at the same time. Students were instructed to bring their

laptops. They 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 and not visiting other web sites. The scratch paper was collected and analyzed to replicate a previous study of code tracing on paper [11].

In the first session the procedure was 1) provide consent and randomly be placed into one of the three practice conditions, 2) complete the demographic survey, 3) complete familiarization activities, 4) complete the pretest, 5) review material on lists and ranges, 6) complete four worked example plus practice pairs where the type of practice problem differed based on the condition, 7) complete a cognitive load survey, and 8) complete the immediate posttest.

At the second session a week later each participant completed the second 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 second posttest also tested for retention of the material one week later.

7 STUDY MATERIALS

We developed, tested, and refined our materials through observations of three undergraduate students from an introductory computing course for computer science majors. Each student was observed as he or she worked through the material for one of the three conditions. After the observational study we added more familiarization material, because some of the students had difficulty using the environment.

We next conducted a pilot study with 24 undergraduate students from an introductory course for computer science majors. In the pilot study, five (21%) of the 24 students submitted at least one solution to the pretest Parsons problem that contained both a correct block and its paired distractor. This indicated that they didn't realize that each distractor was shown paired with the correct code, for at least some of the distractors. At that time the distractors were shown either above or below the correct code, but there was no other visual indication that they were paired.

After the pilot study we added the purple edge decorations shown in Fig. 1 to better indicate that each distractor block was displayed paired with its correct code block. We also added the same subgoal label as a comment to both the correct and distractor code blocks to further indicate that the blocks were paired. The blocks in the source area were always displayed with the purple edge decorations, which helped to show that they were one of a pair. However, the purple edge decorations were not shown on the blocks in the solution area.

7.1 Demographic Survey

The demographic survey asked for 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 they had any prior programming experience, they were also asked what courses and where they took them and how many years they had been programming. In addition, participants were asked to rate their ability to read, fix, and write Python code on a 5-point Likert scale.

7.2 Familiarization Material

The familiarization activities included instruction on how to use the environment, including how to start and finish a timed exam, how to get to the next page, how to answer multiple-choice questions, how to check the solution for the fix code and write code problems, and how to drag blocks and check the solution on a Parsons problem.

This section also included two easy practice multiple choice question, a practice fix code problem with instructions for how to fix the problem, a practice Parsons problem and a write code problem. Both the fix code problem and the Parsons problem included the correct solution displayed above the problem.

7.3 Pretest

There were four timed exams in the pretest. The participants had 15 minutes to complete the first timed exam of multiple-choice questions and 10 minutes to complete each of the other three timed exams (fix code, Parsons problem, and write code).

The five multiple-choice questions included tracing code that included lists, ranges, selection, and iteration. The questions included code to find the minimum value in a list between a range of indices, 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 (as shown in Fig. 2).

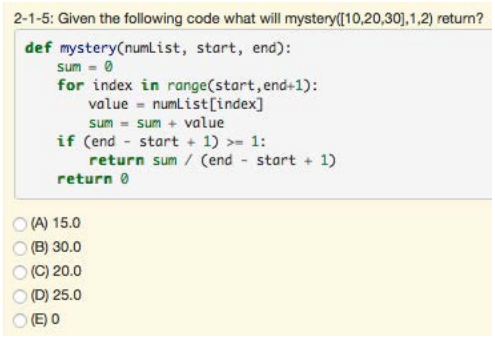

Figure 2: One of the pretest multiple-choice questions

One question provided code that was intended to return the longest run in a list of numbers, but the code contained an error and the student had to select the answer that matched what the code actually returned.

The second timed exam contained one fix code problem. It was a modified version of Soloway's rainfall problem which has been extensively studied [41, 43] as shown in Fig. 3. This problem totals all of the non-negative values in an input loop until a sentinel value is reached and then outputs the average. The

solution should also 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. 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 [41]. The instructions explained the algorithm in English, provided example input and output, and provided hidden unit tests.

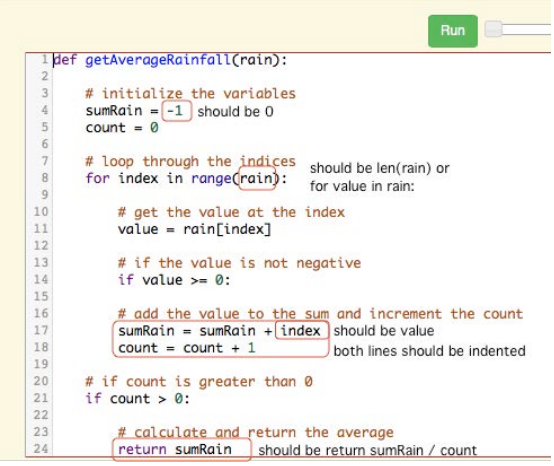

Figure 3: The pretest fix code problem with errors

The third timed exam contained one Parsons problem to create a function to calculate and return the average of the values at a range of indices (inclusive) in a list. The problem had five paired distractors as shown in Fig. 4. The instructions explained the algorithm in English, provided example input and output, and gave feedback on the solution. The feedback was either that the solution was correct, or too short, or one or more code blocks were either out of order or the wrong blocks (and these blocks were highlighted in red), or that the indentation was wrong (and yellow decorations were added to the side of the block with arrows to indicate the direction the block needed to move).

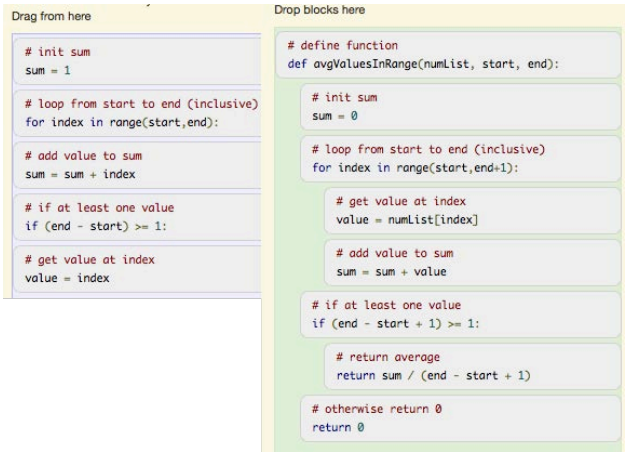

Figure 4: The pretest Parson problem showing unused distractors on the left and the correct solution on the right.

The fourth timed exam contained one write code problem as shown in Fig. 5. This problem asked the participant to write a method to check if a trail was level between a start and end index (inclusive). A trail was considered to be level if the difference between the minimum and maximum values was less than or equal to 10. The problem provided the function header and hidden unit tests. The instructions explained the algorithm in English, provided example input and output, and provided hidden unit tests to test the solution.

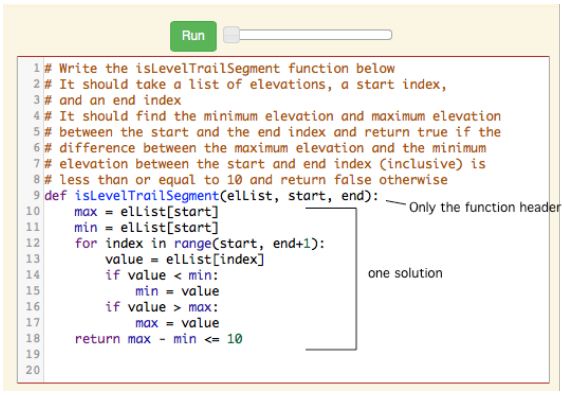

Figure 5: A correct solution to the write code problem

7.4 Review Material

The review material explained what a list was, how to use the range function to create a list, how to get a value from a list, how to get the length of a list, how to loop through all values in a list, and how to loop through a range of indices in a list. It contained example Python code that the participant could run. The students in the experiment had already covered these concepts and had moved on to cover more advanced topics, so we felt it would be helpful to provide this review material.

7.5 Instructional Material

The instruction material contained four worked examples with interleaved practice problems. The worked examples contained an algorithm in English, example input and output, and runnable Python code with hidden unit tests, which all passed. The practice problems varied by condition with one group solving two-dimensional Parsons problems with paired distractors, one solving fix code problems with the same distractors as errors, and the third writing the equivalent code. Each of the practice problems also contained an algorithm in English, example input and output, and a way to test the solution. Each practice problem was in a timed exam and each had a time limit of 10 minutes. The page following the timed exam displayed an English description of a correct solution and the code for that solution.

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. The associated practice problem returned the average, but didn't include the lowest value in the list in the average and also guarded against a divide by zero error. The fourth worked example returned the minimum value in a given range of indices (inclusive). The associated practice problem returned the maximum value in a given range of indices (inclusive).

7.6 Cognitive Load Survey

To measure the cognitive load for each of the practice conditions we used the CS Cognitive Load Component Survey, which has been tested and shown some initial validation in computer science [35]. This survey was adapted from the Cognitive Load Component Survey that has been used to measure cognitive load in statistics and health sciences [27].

7.7 Posttests

The immediate posttest in the first session had the same questions as the pretest. The second posttest, which was administered one week later, was isomorphic to the first posttest, meaning that the problems to be solved had the same structure, but different surface level features, like variable names.

8 PARTICIPANTS

Undergraduate students were recruited from two sections of a first computer science course for computing majors at a research-intensive university in the US. The sections had different instructors, but they 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. One of the authors visited the course during lecture to recruit participants and also sent an announcement to all of the students enrolled in the course. Participants could earn 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 pilot study or large-scale study could alternatively earn up to 5 points of extra credit by writing a paper on a computing innovation, which was graded by one of the authors and that grade was submitted to the course instructors. None of the authors were involved in the teaching of the course.

9 ANALYSIS

A total of 159 students participated in the first session. However, 24 of these students did not answer at least one question during the session or spent less than 30 seconds answering a question. We are reporting on the data from 135 students (45 in the fix condition, 44 in the Parsons condition, and 46 in the write condition) 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 that session. A total of 106 students returned for the second session. Of these, 82 completed all the questions in both the first session and second session and spent at least 30 seconds on each question (27 in the fix condition, 25 in the Parsons condition, and 30 in the write condition).

9.1 Data Analysis

For each instructional practice problem we recorded the start time and end time and then calculated the elapsed time in seconds to compare the efficiency of the three conditions. We created grading rubrics for the write and fix code problems on the pretest and posttests. Two people graded each problem independently and then met to resolve any differences in scores. The hand graded scores on the fix and write problems correlated with the number of unit tests passed ($p < .001$ for all).

We automated the grading for the Parsons problems. Grading started from the beginning of the solution and each line in the correct order received one point and if the line or its paired distractor was indented correctly it received half a point. Grading continued until a line was found that was neither the correct line nor its paired distractor. Grading then continued from the end of the solution in the same fashion toward the first line that had been found to be incorrect. We also reviewed the middle of the solutions manually to give credit if at least two consecutive lines were in the correct order relative to each other. This grading approach was based on our 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.

We checked our data for normal distribution using skewness, whether the peak of the bell curve is in the middle, and kurtosis, whether the bell curve is too narrow or wide [21]. For all pre-test measurements, skewness and kurtosis checks were within the acceptable +/-2 range. For all post-test measurements, skewness was about -2, meaning that there was a slight negative skew (i.e., bell curve looks like it is leaning towards the larger numbers), but these values were still acceptable. Kurtosis, however, was above 3 in all cases, meaning that the scores clustered more closely around the mean than in a normal distribution. Based on these results, we suspect that there was a slight ceiling effect for the posttests in which many participants scored the highest score possible. Most parametric statistical tests, including all of those that we have used, are robust to abnormal kurtosis, meaning that they are still valid with a distribution like ours. Therefore, we used parametric tests to analyze our results instead of their non-parametric equivalents, which tend to be more conservative with lower statistical power [49].

9.2 Efficiency

The Parsons problem condition had the lowest average completion time for each of the four practice problems as shown in Table 1. This difference was significant as measured by an independent measures one-way analysis of variance (ANOVA) $F(2,133) =$

10.835, $p < 0.001$. A Least Significant Difference (LSD) post-hoc test indicated that students in the Parsons problem condition took significantly less time to finish the four practice problems than students in the fix code ($p<0.001$) and write code conditions ($p<0.001$). However, there was no significant difference in completion time between the write code and fix code conditions.

Table 1: Mean Time in Seconds (and Standard Deviation) to Complete each Practice Problem by Condition

	Prac. 1	*Prac. 2*	*Prac. 3*	*Prac. 4*	*Total*
Parsons	84.20 (34.77)	83.64 (35.99)	227.42 (124.66)	77.98 (41.29)	473.24
Fix	114.49 (79.17)	147.67 (128.32)	313.42 (153.40)	103.91 (65.67)	679.49
Write	171.63 (137.61)	113.13 (98.62)	313.65 (153.33)	115.54 (69.28)	713.96

9.3 Learning Performance

The pretest measures (multiple-choice, fix, Parsons, and write) were condensed into a single composite pretest score. To ensure that this was valid and that all of the pretest measures were measuring the same underlying construct, factor analysis was used with varimax rotation. The analysis showed that the four tests loaded onto one factor, which we will call prior knowledge, based on the scree plot and eigenvalues. The factor loadings for each of the individual tests was above .7, the typical cutoff: fix score = .75, write score = .85, multiple choice score = .76, and order score = .79.

9.3.1 Comparing the Practice Conditions. None of the practice conditions performed better than the other conditions on the posttest measurements (multiple choice, fix, Parsons, or write) as shown in Table 2. No interactions between condition and performance on the posttest measures were found either, meaning that participants who practiced on Parsons problems performed as well on the writing posttest as participants who practiced on writing problems and vice versa. In addition, there was no significant difference by condition on performance on the second posttest.

Table 2: Mean Score (and Standard Deviation) by Condition for Pretest and Immediate Posttest

	Pretest (std dev)	*Posttest (std dev)*
Fix (n=44)		
Multiple-Choice	3.48 (1.45)	3.50 (1.50)
Fix	10.36 (2.01)	11.41 (1.23)
Parsons	11.76 (1.01)	11.63 (1.48)
Write	9.50 (3.56)	10.18 (3.49)
Parsons (n=45)		
MC	3.22 (1.17)	3.78 (1.17)
Fix	10.96 (1.69)	11.42 (1.34)
Parsons	11.61 (1.31)	11.77 (1.19)
Write	8.40 (3.68)	9.78 (3.27)
Write (n=46)		
MC	3.41 (1.24)	3.72 (1.19)
Fix	10.96 (1.69)	11.37 (1.25)
Parsons	11.38 (1.93)	11.70 (1.28)
Write	9.48 (3.75)	10.30 (2.62)

9.3.2 Comparing the Pretest to the Posttests. When analyzing repeated measures data, as we have for the pre-test, immediate post-test, and delayed post-test, it is common to violate the assumption of sphericity, as tested with Mauchly's test. Our data violated the sphericity assumption, $p < .001$, so we've used the Huynh-Feldt correction to make our ANOVA results more conservative. We found a significant difference from the pretest to both posttests using an omnibus repeated measures ANOVA for the fix problem F(using Huynh-Feldt correction; 1.9, 161.2) = 7.34, $p = .001$, and the write problem F(using Huynh-Feldt correction; 1.6, 139.9) = 4.56, $p = .018$. Participants also performed better on the fix and write code problems on both post-tests than on the pre-test. However, their performance on the second post-test was worse than the first post-test, though not so bad as to be statistically equivalent to the pre-test. There were no significant differences from the pretests to the posttests on the multiple-choice questions or the Parsons problem. For the multiple-choice questions this may have been due to the lack of feedback on the correctness of the pretest answers. It is possible that the students simply remembered what they had answered before and used the same answer since they were not told if the answers were wrong. The lack of significant difference on the Parsons problem is likely due to a ceiling effect, because each group had a mean above 11 (out of 12 possible points).

9.4 Cognitive Load

We found no significant difference in the self-reported cognitive load measures between the three conditions, $F(2, 132) = 1.21$, $p = .30$. However, the students in the Parsons problem condition solved the same problems as those in the fix code and write code conditions in significantly less time as shown in Table 1.

9.5 Comparing Demographic Data to Performance

To check for possible interaction between the demographic data and the conditions, we created a composite score using the fix and write code problem scores from the two posttests. We found no interaction between condition and demographic characteristics that affected performance.

We found a moderate correlation for age, $r(88) = -.22$, $p = .04$, with younger students performing better than older students. We also found a moderate correlation by major with computer science majors, $\rho(88) = -.24$, $p = .02$, performing better than the non-computer science majors. One unusual finding was that students who had a previous programming course performed *worse* than those without any prior programming experience, $\rho(88) = -.24$, $p = .02$. This could be due to prior experience in another language with different syntax or semantics than Python. We found a moderate correlation on all of the self-reported measures of ability to read, $\rho(88) = .32$, $p = .002$; fix, $\rho(88) = .28$,

$p = .008$; and write, $\rho(88) = .34$, $p = .001$, Python code. We found a strong correlation between expected grade in the course and performance. $\rho(88) = -.50$, $p < .001$. We did not find any correlations for the other demographic characteristics including race, gender, first language, high school grade point average, or college grade point average.

10 DISCUSSION

The students in the Parsons problem condition completed the four instructional practice problems in significantly less time than those in the fix code or write code conditions. This supports our first hypothesis that Parsons problems are a more efficient form of practice than fixing the same code with errors or than writing the equivalent code. There was no significant difference between the completion time for the students in the fix code or write code conditions. Fix code problems, like Parsons problems, have an advantage over write code problems, because the student doesn't need to type the code for the solution. However, Parsons problems with paired distractors are easier for students to debug since they don't have to interpret compiler errors or debug code. They can simply pick between the paired correct and incorrect blocks.

There was a significant improvement from the pretest fix and write code problems to the same problems on the immediate posttest as well as on the posttest one week later, which provides evidence of near transfer and retention. These findings, coupled with the fact that there was no significant performance difference on the posttests by condition, supports our second hypothesis that solving Parsons problems would lead to similar learning performance than fixing code with errors or than writing the equivalent code.

However, it is possible that the performance improvements may not be solely due to the practice condition. Students may have learned from the pretest problems, review material, worked examples, or answers to the four practice problems. This study could have been improved by adding a control group that did an off-task activity rather than solve the four practice problems. This would have strengthened the claim that the performance gains were due to the practice problems, and not the other materials. The retention results on the delayed posttest could have also been partially due to learning in the students' course during the week after the immediate posttest, however the topics covered that week were more advanced. Further experiments should be done to verify that solving Parsons problems results in equivalent performance gains compared to fixing and/or writing code.

We did not find any significant difference on the self-reported cognitive load survey by condition, so our third hypothesis that Parsons problems would have lower self-reported cognitive load was not supported. While the cognitive load survey that we used had been initially validated, it may not be an effective measure for comparing the cognitive load of different types of practice problems. The first study was 2.5 hours long and included many different parts, which were completed one after the other without a break. It is possible that students were responding to the difficulty of the entire study rather than just the instructional section or were fatigued and just wanted to finish. Further studies should be done to test if the self-reported cognitive load of solving Parsons problems is lower than that of solving fix code and write code problems. However, students in the Parsons problem condition solved the same practice problems in significantly less time than those in the fix code or write code conditions, which implies that they do have a lower cognitive load.

11 CONCLUSIONS

This study provides evidence that solving two-dimensional Parsons problems with paired distractors takes significantly less time than fixing the same code with the same errors as the distractors or than writing the equivalent code, while still resulting in statistically significant improvement in scores from pretest to immediate posttest and retention one week later. This demonstrates that solving Parsons problems with paired distractors is a more efficient, but just as effective, form of practice than writing or fixing code. However, further research needs to be done to verify that the performance gains were solely due to the type of practice.

If Parsons problems are a more efficient and effective form of practice than writing code, they could be used to speed the learning of basic syntax, semantics, and algorithms. While this study only included four practice Parsons problems in the instructional material, our research team has added over a hundred Parsons problems to several free interactive ebooks that we and others have developed for introductory programming in Python and Java [16, 18]. Thousands of students and hundreds of institutions are already using these ebooks. Instructors can use or customize these existing ebooks or use the ebook platform (Runestone) to create their own ebooks with Parsons problems. The ebooks log user interaction, which can be used for research in computing education, including work on the effectiveness and efficiency of Parsons problems.

ACKNOWLEDGMENTS

This material is based on work supported by the National Science Foundation under grants 1138378 and 1432300. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Several undergraduate students contributed to this work: Yamini Nambiar graded the fix and write code problems, Shouyra Singh calculated averages, and Katherine Guzdial analyzed the timing data. We also thank the reviewers for their work to help us clarify this paper.

REFERENCES

[1] Robert K. Atkinson, Sharon J Derry, Alexander Renkl and Donald Wortham Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research*, 70, 2 (2000), 181–214.

[2] Klara Benda, Amy Bruckman and Mark Guzdial 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.

[3] Jens Bennedsen and Michael E. Caspersen Failure rates in introductory programming. *SIGCSE Bull.*, 39, 2 (2007), 32-36.

[4] Elizabeth L Bjork and Robert A. Bjork 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.

[5] Benedict Du Boulay *Some Difficulties of Learning to Program*. Lawrence Erlbaum Associates, 1988.
[6] John D. Bransford, Ann L. Brown and Rodney R. Cocking *How People Learn*. NATIONAL ACADEMY PRESS, 2000.
[7] Richard Catrambone The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General*, 127, 4 (1998), 355.
[8] Nick Cheng and Brian Harrington. The Code Mangler: Evaluating Coding Ability Without Writing any Code. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (Seattle, Washington, USA, 2017). ACM.
[9] Michelene T. H. Chi Active-Constructive-Interactive: A Conceptual Framework for Differentiating Learning Activities. *Topics in Cognitive Science* 1(2009), 73-105.
[10] G. Cooper and J. Sweller The effects of schema acquisition and rule automation on mathematical transfer. *Journal Educational* (1987), 347-362.
[11] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson and Mark Guzdial. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA, 2017). ACM.
[12] Paul Denny, Andrew Luxton-Reilly and Beth Simon. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the International Computing Education Research Conference* (Sydney, Australia, 2008). ACM.
[13] John Dewey *Experience & Education*. Macmillan, New York, 1959.
[14] D. Druckman and R. A. Bjork *In the mind's eye: Enhancing human performance*. National Academy Press, Washington DC, 1991.
[15] Elsa Eiriksdottir and Richard Catrambone Procedural instructions, principles, and examples: how to structure instructions for procedural tasks to enhance performance, learning, and transfer. *Human Factors*, 53, 6 (2011), 749-770.
[16] Barbara Ericson *Java Review Book for the AP CS A exam*, https://runestone.academy/runestone/static/JavaReview/index.html
[17] Barbara J. Ericson, Mark J. Guzdial and Briana B. Morrison. Analysis of Interactive Features Designed to Enhance Learning in an Ebook. In *Proceedings of the 2015 ACM Conference on International Computing Education Research* (Omaha, NE, USA, August 09-3, 2015, 2015). ACM.
[18] Barbara Ericson, Kantwon Rogers, Miranda Parker, Briana Morrison and Mark Guzdial. Identifying Design Principles for CS Teacher Ebooks through Design-Based Research. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia, 2016). ACM.
[19] K. Anders Ericsson *The Influence of Experience and Deliberate Practice on the Development of Superior Expert Performance*. Cambridge University Press, 2006.
[20] Stuart Garner 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.
[21] Frederick J Gravetter and Larry B Wallnau *Statistics for the behavioral sciences*. Cengage Learning, 2016.
[22] Kyle J. Harms, Noah Rowlett and Caitlin Kelleher. Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (Atlanta, GA, 2015). IEEE.
[23] Kyle James Harms, Jason Chen and Caitlin L. Kelleher. Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (Melbourne, VIC, Australia, 2016). ACM.
[24] Petri Ihantola and Ville Karavirta. Open source widget for parson's puzzles. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education* (Bilkent, Ankara, Turkey, 2010). ACM.
[25] Petri Ihantola and Ville Karavirta Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. *Journal of Information Technology Education*, 10 (2011), 1-14.
[26] J. A. LeFevre and P. Dixon Do written instructions need examples? *Cognition and Instruction*, 3 (1986), 1-30.
[27] Jimmie Leppink, Fred Paas, Cees PM Van der Vleuten, Tamara Van Gog and Jeroen JG Van MerriÃ«nboer Development of an instrument for measuring different types of cognitive load. *Behavior research methods*, 45, 4 (2013), 1058-1072.
[28] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hammer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon and Lynda Thomas. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Proceedings of the Working group reports from ITiCSE on Innovation and technology in computer science education* (Leeds, United Kingdom, 2004). ACM.
[29] Lauren E Margulieux, Mark Guzdial and Richard Catrambone. Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications. In *Proceedings of the ninth annual international conference on International computing education research* (Auckland, New Zealand 2012). ACM.
[30] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting and Tadeusz Wilusz A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, 33, 4 (2001), 125-180.
[31] Jeroen J. G. Van Merriënboer Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of educational computing research*, 6, 3 (1990), 265-285.
[32] Jeroen J. G. Van Merriënboer and Marcel B. M. De Croock Strategies for computer-based programming instruction: program completion vs. program generation. *Journal of Educational Computing Research*, 8, 3 (1992), 365-394.
[33] van Merriënboer, Kirschner and Kester Taking the Load Off a Learner's Mind: Instructional Design for Complex Learning. *EDUCATIONAL PSYCHOLOGIST*, 38, 1 (2003), 5-13.
[34] Brad Miller and David Ranum. Runestone interactive: tools for creating interactive course materials. In *Proceedings of the first ACM conference on Learning @ scale conference* (Atlanta, Georgia, USA, 2014). ACM.
[35] Briana B. Morrison, Brian Dorn and Mark Guzdial. Measuring cognitive load in introductory CS: adaptation of an instrument. In *Proceedings of the tenth annual conference on International computing education research* (Glasgow, Scotland, United Kingdom, 2014). ACM.
[36] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson and Mark Guzdial. Subgoals Help Students Solve Parsons Problems. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (Memphis, Tennessee, 2016), ACM.
[37] Dale Parsons and Patricia Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education* (Hobart, Australia, 2006). Australian Computer Society, Inc.
[38] Peter L. Pirolli and John R. Anderson The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 39, 2 (1985), 240-272.
[39] D. Rohrer and K Taylor The effects of over-learning and distributed practice on the retention of mathematics knowledge. *Applied Cognitive Psychology*, 20 (2006), 1209-1224.
[40] David Williamson Shaffer and Mitchel Resnick "Thick" Authenticity: New Media and Authentic Learning. *Journal of Interactive Learning Research*, 10, 2 (1999), 195-215.
[41] Simon. Soloway's Rainfall Problem has become Harder. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering* (2013). IEEE Computer Society.
[42] J. A. Slobada, J. W. Davidson, M. J. A. Howe and D. G. Moore The role of practice in the development of performing musicians. *British Journal of Educational Psychology*, 87 (1996), 287-309.
[43] Elliot Soloway Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29, 9 (1986), 850-858.
[44] John Sweller Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12, 2 (1988), 257-285.
[45] John Sweller *Cognitive Load Theory: Recent Theoretical Advances*. Cambridge University Press, 2010.
[46] John Sweller Instructional design consequences of an analogy between evolution by natural selection and human cognitive architectures. *Instructional Science*, 32 (2004), 9-31.
[47] Allison Elliott Tew and Mark Guzdial. Developing a validated assessment of fundamental CS1 concepts. In *Proceedings of the 41st ACM technical symposium on Computer science education* (Milwaukee, Wisconsin, USA, 2010). ACM.
[48] John Gregory Trafton and Brian J. Reiser. The contributions of studying examples and solving problems to skill acquisition. In *Proceedings of the 15th Annual Conference of the Cognitive Science Society* (Hillsdale, NJ, 1993). Lawrence Erlbaum Associates, Inc.
[49] William MK Trochim and James P Donnelly *Research methods knowledge base (3rd ed)*. Atomic Dog, Cincinnati, OH, 2006.
[50] M. Tuffiash, R. W. Roring and K. A. Ericsson Expert performance in Scrabble: Implications for the study of the structure and acquisition of complex skills. *Journal of Experimental Psychology: Applied*, 13 (2007), 124-134.
[51] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva and Tadeusz Wilusz. A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE working group reports conference on Innovation and technology in computer science education-working group reports* (Canterbury, England, United Kingdom, 2013). ACM.
[52] Barry J. Wadsworth *Piaget's Theory of Cognitive and Affective Development - Fourth Edition*. Longman, New York 1989.
[53] Mark Ward and John Sweller Structuring Effective Worked Examples. *Cognition and Instruction*, 7, 1 (1990), 1–39.
[54] Christopher Watson and Frederick W.B. Li. Failure rates in introductory programming revisited. In *Proceedings of the 2014 conference on Innovation and technology in computer science education* (Uppsala, Sweden, 2014). ACM.
[55] Leon E. Winslow Programming Pedagogy - A Psychological Overview. *SIGCSE Bull.*, 28, 3 (1996), 17-22.
[56] X. Zhu and H. A. Simon Learning mathematics from examples and by doing. *Cognition and Instruction*, 4 (1987), 137-166.