Design and Pilot Testing of Subgoal Labeled Worked Examples for Five Core Concepts in CS1

Lauren E. Margulieux
Georgia State University
Department of Learning Sciences
Atlanta, GA 30302-3978
lmargulieux@gsu.edu

Briana B. Morrison
University of Nebraska Omaha
Computer Science Department
Omaha, NE 68182
bbmorrison@unomaha.edu

Adrienne Decker
University at Buffalo
Dept. of Engineering Education
Buffalo, NY 14260-4200
adrienne@buffalo.edu

ABSTRACT

Subgoal learning has improved student problem-solving performance in programming, but it has been tested for only one-to-two hours of instruction at a time. Our work pioneers implementing subgoal learning throughout an entire introductory programming course. In this paper we discuss the protocol that we used to identify subgoals for core programming procedures, present the subgoal labels created for the course, and outline the subgoal-labeled instructional materials that were designed for a Java-based course. To examine the effect of subgoal labeled materials on student performance in the course, we compared quiz and exam grades between students who learned using subgoal labels and those who learned using conventional materials. Initial results indicate that learning with subgoals improves performance on early applications of concepts. Moreover, variance in performance was lower and persistence in the course was higher for students who learned with subgoals compared to those who learned with conventional materials, suggesting that learning with subgoal labels may uniquely benefit students who would normally receive low grades or dropout of the course.

CCS CONCEPTS

• **Social and professional topics** → *Computer science education*

KEYWORDS

CS1; subgoal learning; worked examples; problem solving

ACM Reference format:
Lauren Margulieux, Briana B. Morrison and Adrienne Decker. 2019. Design and Pilot Testing of Subgoal Labeled Worked Examples for Five Core Concepts in CS1. In *Proc. of the 24th Annual Conf. on Innovation & Technology in Computer Science Education (ITiCSE'19). July 15–17, 2019, Aberdeen, Scotland UK., ACM, New York, NY, USA*, https://doi.org/10.1145/3304221.3319756

1 Introduction

The computing education community is constantly exploring methods to improve learning outcomes and student retention in college-level introductory programming courses. Subgoal-labeled worked examples are a promising method to improve problem-solving performance for novice learning, but they have been tested only for one to two hours of instruction at a time [7, 8, 9]. The current project substantially extends this line of work by identifying the subgoals for problem-solving procedures typically taught throughout an introductory Java programming course, developing subgoal-labeled worked examples and paired practice problems to be used while teaching the course, and conducting a pilot test on the effectiveness of the materials to improve problem-solving performance across an entire semester.

The guiding questions for this work were:

1. What are the subgoals of problem-solving procedures typically taught in college-level introduction to programming courses that use an imperative programming language?
2. If students learn procedures using subgoal-oriented worked-examples and paired practice problems, do they perform better than students who learn using non-subgoal-oriented materials on course assessments?

2 Subgoal Learning in Programming Education

Subgoal learning is an instructional design framework used in programming education that improves novice problem-solving performance [3, 6, 7, 8, 9]. Subgoal learning explicitly teaches students the subgoals, or functional pieces, of a problem-solving procedure. For example, to solve a problem with a loop, students must define and initialize variables, so defining and initializing variables is a subgoal of solving a problem with a loop. The specific steps taken to achieve this subgoal varies from problem to problem, but the function remains the same. Novices solve programming problems better when they explicitly learn the subgoals of a procedure because they often do not recognize these functional pieces on their own [4].

Worked examples are commonly used to teach problem-solving procedures for well-structured problems because they demonstrate how to apply an abstract procedure to a concrete problem before the learner can solve problems independently [2, 12, 13]. The drawback of worked examples, however, is that they must include details specific to a problem. For example, to demonstrate how to solve a problem using a `for loop`, the worked example must also include a cover story, such as "write a loop that will calculate the average age of the first 100 people to take a survey." Learners tend to organize information about the procedure using these easy-to-grasp details rather than around the hard-to-conceptualize abstract procedure that they are learning, leading to difficulty transferring knowledge to new problems [2, 11]. Subgoal learning addresses this problem by pointing out shared functional features in worked examples, helping learners to organize information so that it can transfer more easily [4, 7]. Furthermore, by drawing learners' attention to the functional features and away from the superficial details, subgoal learning can help learners manage cognitive load [9].

3 Identifying Subgoals with the TAPS Protocol

Some readers might think that instructors naturally point out the subgoals of problem-solving procedures, but they often do not [4]. Unlike declarative knowledge (i.e., factual knowledge, such as 2+2=4), procedural knowledge (i.e., knowledge about how to do something, such as tying a shoe) becomes more automatized the more it is used [1]. Therefore, experts in a domain have procedural knowledge that has become automatized over years of practice, and they cannot easily recognize or verbalize it. As a result, the process of identifying subgoals is arduous because it requires accessing tacit procedural knowledge from an expert. To access tacit procedural knowledge and identify the subgoals of five core programming procedures, we employed a cognitive task analysis, specifically the Task Analysis by Problem Solving (TAPS) protocol developed by Catrambone [5].

3.1 TAPS Protocol

The TAPS protocol requires a subject-matter expert (SME) and a knowledge-extraction expert (KEE) who is unfamiliar with the problem-solving procedure. The KEE asks the SME to bring problems that exemplify the problem-solving procedure. In the following description, the SME will have female pronouns and the KEE male pronouns to help distinguish between them.

The session starts with the KEE asking the SME to solve the first problem. The SME does not provide a lecture or explanation of the problem-solving process before solving problems. Instead, the SME solves the first problem and explains what she is doing while the KEE takes notes. During the first problem, the KEE typically does not ask many questions while gaining a general sense of the procedure, but the KEE might ask the SME to repeat steps or re-explain steps that he missed or did not understand.

When the KEE is finished taking notes on the first problem, he asks the SME to solve another problem and explain what she is doing. Again, the KEE takes notes on the process, specifically looking for similarities between the problems. During the second problem, the KEE typically asks more questions, especially about 1) analogous components of the two problems, 2) why the SME did a problem-solving step, and 3) how the SME knew which step to take next. SMEs can typically answer questions about analogous steps easily. Beyond the explanation that they provide while solving the problem (i.e., the declarative knowledge that they have about the procedure), they often struggle to explain why they took a step or how they knew which step to take next (i.e., the procedural knowledge that they have automatized). Automatized procedural knowledge is often what instructors struggle to impart to their students because they think that it is common knowledge or because they think it is intuitive.

When the SME starts to struggle to explain steps of the problem-solving procedure, this is the level at which the KEE often identifies subgoals. In TAPS, it is important that the KEE be unfamiliar with the problem-solving procedure because his novice perspective will help distinguish between common knowledge and automatized procedural knowledge, both of which will seem like common knowledge to the SME. The first stage of TAPS ends when the KEE feels like he has a complete set of notes for explaining the problem-solving procedure. The number of problems that the SME solves to reach this point depends on the complexity of the procedure, the skill of the KEE at extracting knowledge, and the skill of the SME at verbalizing tacit knowledge. The first stage typically takes between one and four hours. It is a demanding task for both the KEE and SME, and we recommend taking an extended break every two hours.

During the second stage of TAPS, the KEE attempts to solve problems using his notes for guidance. When the KEE reaches an impasse, he can ask the SME for help and update his notes. The SME should not offer help. Once the KEE can reliably solve new problems using only his notes, the notes are complete.

During the final stage of TAPS, the KEE organizes and edits the complete notes to create a list of subgoals for the procedure and asks the SME for feedback. The subgoals represent only the procedural knowledge required for a problem-solving procedure, not the declarative knowledge, such as what operation `%` represents (modulus). While both types of knowledge are necessary to solve problems, instructors can easily recognize and explain declarative knowledge. Therefore, subgoal learning interventions focus on the procedural knowledge that instructors can struggle to share.

3.2 Identifying Subgoals in Introductory Programming (Java)

We used the TAPS protocol to identify subgoals of problem-solving procedures using expression (assignment) statements, selection statements, loops, object instantiation and method calls, writing classes, and arrays in Java. The SME was one of the authors, Morrison, a computing education researcher and assistant professor in the CS Department at University of Nebraska Omaha. Morrison has 23 years of experience teaching programming and over 15 years of experience specifically teaching introductory courses in Java. The KEE was another one of the authors, Margulieux, a computing education researcher and assistant professor in the Department of Learning Sciences at Georgia State University. Margulieux has 6 years of experience using the TAPS protocol and had never learned programming before serving as KEE on this project.

For each programming concept, the SME and KEE identified subgoals for evaluating code and writing code. Furthermore, after creating the list of subgoals, they received feedback from the other author, Decker, a computing education researcher with 18 years of experience teaching introductory programming. The subgoals are listed in Figure 1. Part A for each subgoal topic lists the evaluate subgoals, and part B lists the write subgoals. Some subgoals are broken down into sub-subgoals.

4 Designing Instruction

After the identification phase, we designed instructional materials to help students learn the subgoals of the problem-solving procedures. The traditional method of teaching subgoals in STEM education is through subgoal-labeled worked examples (SLWEs) [4, 7, 9]. Students who study SLWEs perform better than those who study unlabeled worked examples because the subgoal labels highlight the structure of the problem-solving procedure and prompts students to recognize similarities between solutions [4, 7, 9]. Therefore, we designed SLWEs for each set of subgoals with multiple levels of difficulty.

Figure 1. Subgoals identified through TAPS protocol.

Subgoals for evaluating and writing expression (assignment) statements	
A. Evaluate expression statement	B. Write expression statement
1. Determine whether data type of expression is compatible with data type of variable 2. Update variable for pre based on side effect 3. Solve arithmetic equation 4. Check data type of copied value against data type of variable 5. Update variable for post based on side effect	1. Determine expression that will yield variable 2. Determine data type and name of variable and data type of expression 3. Determine arithmetic equation with operators 4. Determine expression components 5. Operators and operands must be compatible
Subgoals for evaluating and writing selection statements	
A. Evaluate selection statement	B. Write selection statement
1. Diagram which statements go together 2. For if statement, determine whether expression is true or false 3. If true – follow true branch, if false –follow else branch or do nothing if no else branch	1. Define how many mutually exclusive paths are needed 2. Order from most restrictive/selective group to least restrictive 3. Write if statement with Boolean expression 4. Follow with true bracket including action 5. Follow with else bracket 6. Repeat until all groups and actions are accounted for
Subgoals for evaluating and writing loops.	
A. Evaluate loops	B. Write loops
1. Identify loop parts a. Determine start condition b. Determine update condition c. Determine termination condition d. Determine body that is repeated 2. Trace the loop a. For every iteration of loop, write down values	1. Determine purpose of loop a. Pick a loop structure (while, for, do_while) 2. Define and initialize variables 3. Determine termination condition a. Invert termination condition to continuation condition 4. Write loop body a. Update loop control variable to reach termination
Subgoals for calling and writing methods	
A. Call or trace method calls	B. Write methods
1. Classify method as `static` method or `instance` method a. If `static`, use the class name b. If `instance`, must have or create an instance 2. Write (instance / class) dot method name and ( ) 3. Determine whether parameter(s) are appropriate a. Number of parameters passed must match method declaration b. Data types of parameters passed must match method declaration (or be assignable) 4. Determine what the method will return (if anything: data type, void, print, change state of object) and where it will be stored (nowhere, somewhere) 5. Evaluate right hand side of assignment (if there is one). Value is dependent on method's purpose	1. Define method header based on problem 2. Define return statement at the end 3. Define method body/logic a. Determine types of logic (expression, selection, loop, etc.) b. Define internal variables c. Write statements
Subgoals for using objects and writing classes	
A. Use objects (creating instances)	B. Write classes (associated rules sheet)
1. Declare variable of appropriate class datatype. 2. Assign to variable: keyword `new`, followed by class name, followed by (). 3. Determine whether parameter(s) are appropriate (API) a. Number of parameters b. Data types of the parameters	1. Name it 2. Differentiate class-level (`static`) vs. instance/object-level variables 3. Differentiate class-level (`static`) vs. instance/object behaviors/methods 4. Define instance variables (that you want to be interrelated) 5. Define class variables (`static`) as needed 6. Create constructor (behavior) that creates initial state of object 7. Create 1 accessor and 1 mutator behaviors per attribute 8. Write toString method 9. Write equals method 10. Create additional methods as needed

Figure 1. Subgoals identified through TAPS protocol *(continued)*

Subgoals for evaluating and writing arrays	
A. Evaluate arrays	B. Write arrays
1. Set up array from 0 to size-1 2. Evaluate data type of statements against array 3. Trace statements, updating slots as you go a. Remember assignment subgoals	1. Data type plus [ ] 2. Variable name = `{initializer list}`, or `new datatype [size]`

For each topic in Figure 1, we created several SLWEs in increasing level of difficulty. The simplest version may skip some of the subgoals identified for the procedure because they are not necessary for simple problems. It may also include more sub-subgoals than later levels of SLWEs to provide more guidance at the earliest stage of learning. The difficulty level of SLWEs gradually increase, adding subgoals and reducing sub-subgoals as the problems increase in complexity. The parameters for problem complexity in each level were determined by Morrison and Margulieux after identifying the subgoals of each procedure, and Decker provided feedback. Based on the parameters, Morrison and Decker designed the first draft of the SLWE and practice problems (see Figure 2), and Margulieux provided feedback and help during iterative design.

The SLWEs were interleaved with practice problems so that after students studied a worked example, they attempted to solve at least one similar problem. Interleaving worked examples and practice problems improves learning efficiency over studying worked examples as in a block before attempting to solve problems [14]. The SLWE--practice-problem pairs were intended to be used as either a homework assignment or as instructional materials that the instructor discusses in class. If using the materials in class, other instructional techniques can be combined with the materials. For example, instructors might use a flipped classroom approach in which students learn about the problem-solving procedure conceptually before class; then class time is used to practice problem solving with the SLWE--practice-problem pairs. While students are working on practice problems, they can engage in Peer Instruction, which asks students to explain their solution to a peer and resolve differences in answers before the instructor provides the correct answer and has been effective in introductory programming courses [10].

5 Report of Pilot Test

The instructional materials were pilot tested in the introductory programming courses at University of Nebraska Omaha. There are five sections of the course taught by three full-time faculty instructors with similar experience levels and supported by six graduate teaching assistants. All sections of the course are coordinated so that they include the same topics at the same pace and have the same quizzes and exams. The course was designed as a flipped class in which students watched recorded lectures before class and then answered peer instruction questions during class and problem solved in small groups. All sections follow this format, but the online section uses a different medium for class. Two sections of the course were taught by Morrison and used the SLWE during class. Three sections of the course, including the online section, were taught with conventional, non-subgoal worked examples. All other instructional features of the sections were the same among the sections. The pilot test compares student performance (i.e., quantitative grades) on quizzes and exams across sections.

5.1 Study Methods

The total number of students across all five sections was 307 based on enrollment at the beginning of the semester. Students were excluded from analysis if they did not complete at least one exam and one quiz, effectively dropping the course, making $N = 265$. They were split across the conventional course group ($n = 145$) and the SLWE course group ($n = 120$).

Though we do not have space in the current paper to fully discuss learner characteristics, we found no correlations between group and demographic factors or learner characteristics, including reason for taking the course, expected grade, expected difficulty of the course, interest in the course content, anxiety about course performance, age, gender, full-time or part-time status, race, primary language, family socioeconomic status, academic major, high school GPA, college GPA, year in school, or prior experience with programming. There were also no correlations between these factors and quiz or exam scores. Thus, these learner characteristics were not used a covariates or random factors in the analysis.

Figure 2. Example of subgoal-labeled worked examples and practice problem pair for writing expression statements (see Figure 1).

Subgoal-Labeled Worked Example 1 – Simple arithmetic equation	**Paired Practice Problems**

Assume the following given declarations:
`int max = 100;`
`double tax = 0.5, result, bill = 26.12;`
Write the code to store max multiplied by tax in the variable result.
SG1: *Determine expression that will yield variable*
`max * tax`
SG2: *Determine data type and name of variable and data type of expression*
Result to be stored in variable `result`. That variable is a **double.** The expression `max * tax` is an **int** multiplied by a **double** yields a **double.** A **double** can be assigned to a **double.**
SG3: *Determine arithmetic equation with operators*
`result = max * tax;`
	Practice Problem 1:
	Calculate a 15% tip on the bill.
	
	Practice Problem 2:
	Determine the total amount owed including bill, tip, and tax.

5.1.1 Data collection sources. Student performance on the four exams and five quizzes was collected. We also had the initial instructor keep a weekly journal on the teaching experience. Below are the characteristics for the student performance items:

- The majority of quiz questions were either multiple choice or short answer. Exams consisted of multiple choice questions (usually 1/3 to 1/2 of the exam grade) and short answer and long answer questions.
- All exam and quiz questions were based on peer instruction questions presented in class (near transfer) or the homework assignments (far transfer).
- Exams and quizzes were scored identically across sections. All multiple choice and short answer questions were automatically graded, and all student responses were reviewed by one member of the instructional team. Rubrics for open ended questions were developed and a single member of the instructional team graded all responses for a single question.
- For exams, students were allotted 2 hours.
- Quizzes were assigned over weekends, from Friday morning until Monday at midnight and had a 20-minute time allotment.

5.2 Results and Discussion

The quiz and exam scores were each analyzed in a few ways to examine the differences in performance between students who received SLWEs and those who received conventional, non-subgoal instruction. The following values were calculated for each student:

1. **Total** score, which is out of all available points on exams or quizzes. Thus, if a student did not turn in an exam or quiz (e.g., because they dropped out of the course) this score would treat the missing grade as a zero.
2. **Average** score, which is the average score for all exams or quizzes that were submitted by a student. Thus, if a student did not turn in an exam or quiz, this score would not be affected by the missing grade.
3. **Number** of assessments completed, which is the total number of exams or quizzes taken regardless of score.

Conducting analyses with these different values allows us to examine the performance and retention differences between groups. Initially in the analyses, the online section was separate from the other non-subgoal sections in case the medium of the courses affected performance or there was a fundamental difference between students who signed up for the online or in-person courses. In all of the analyses, however, the online and in-person non-subgoal groups performed equivalently. Therefore, they were consolidated for the final analyses.

5.2.1 Quiz performance. For all three values calculated from students' quiz grades, the subgoal group performed better than the non-subgoal group. For the **total** quiz score, including zeros for missing grades, the maximum score was 31. The subgoal group (M = 12.0, SD = 5.6) performed better than the non-subgoal group (M = 9.5, SD = 6.3) with a medium effect size, d = 0.42. The SD of the subgoal group was sufficiently less than that of the non-subgoal group to violate the homogeneity test, p = .03; therefore, the non-parametric and more conservative Mann-Whitney test was used to compare groups, U = 6703, p = .001. For the **average** quiz score, not including zeros for missing grades, the maximum score was 6.2 points. The subgoal group (M = 2.98, SD = 0.9) performed better than the non-subgoal group (M = 2.57, SD = 1.0) with a medium effect size, d = 0.44, t(264) = 12.03, p = .001. For the **number** of quizzes taken out of five, the subgoal group (M = 3.9, SD = 1.2) took more quizzes than the non-subgoal group (M = 3.4, SD = 1.6) with a small-medium effect size, d = 0.32. The SD of the subgoal group was less than that of the non-subgoal group, p < .01; therefore, Mann-Whitney was used, U = 7126, p = .01. This pattern of results means that the subgoal group completed more quizzes and performed better on them, regardless of whether the missing grades are factored in as zeros or not.

More detailed examination of scores on each quiz with repeated measures analysis suggests that the subgoal group consistently performed better than the non-subgoal group on each quiz. Mauchley's test of sphericity was significant, p < .01, as is common in repeated measures analyses, and the Huynh-Feldt correction was used to make the statistical values more conservative. There was no main effect of quiz, F(5, 260) = 2.04, p = .11, suggesting that the subgoal intervention was equally effective across all topics. Given that each quiz was designed to test only the new concepts that had been taught, this finding means that students benefitted from the SLWEs for each new topic, despite gaining knowledge about other programming topics. This analysis could only be conducted with the total quiz score because the average quiz score would be missing data from un-submitted quizzes. Given that the effect size for the difference between groups was equivalent for the analyses with the total and average quiz scores, this analysis is expected to be representative of average quiz score as well.

5.2.2 Exam performance. Students' exam grades had a different pattern than their quiz grades. For the **total** of all exam scores, including zeros for missing grades, the maximum score was 200. The subgoal group (M = 140.3, SD = 42.4) performed better than the non-subgoal group (M = 128.2, SD = 51.6) with a small effect size, d = 0.26, t(264) = 4.20, p = .04. For the **average** exam score, not including zeros for missing grades, the maximum score was 50. In this case, the subgoal group (M = 37.5, SD = 7.6) did not perform statistically better than the non-subgoal group (M = 35.8, SD = 9.1), d = 0.20. The SD of the subgoal group was less than that of the non-subgoal group, p = .02, so Mann-Whitney was used, U = 7975, p = .24. This difference in results can be explained by the different in **number** of exams taken. Out of four exams, the subgoal group (M = 3.7, SD = 0.8) took more than the non-subgoal group (M = 3.5, SD = 1.0) with a small effect size, d = 0.22. The SD of the subgoal group was less than that of the non-subgoal group, p < .01, so Mann-Whitney was used U = 7785, p = .045. The most plausible explanation for this pattern of results, based on the statistics, is that exam performance was equivalent between the subgoal and non-subgoal groups for students who took all exams. The difference in the **total** exam analysis is likely due to the zeros from students who did not take all exams. Because students in the non-subgoal group had disproportionally more zeros than the subgoal group, their mean total score would decrease more than the subgoal group's.

This pattern of results has two likely implications. First, it implies that SLWEs did not improve exam scores, which aligns with the theory behind subgoal learning. Subgoal learning has been shown to be effective because it helps learners to recognize the abstract structure of problem-solving procedures before they have enough knowledge to recognize it for themselves. Therefore, subgoal learning should be most effective at the beginning of learning a new procedure (e.g., when learners take a quiz), and as learners gain more knowledge about a procedure (e.g., by the time they take an exam), the effect should diminish. Second, the pattern of results implies that students in the subgoal group were more likely than those in the non-subgoal group to complete the semester. Especially because the subgoal group had less variance on exam scores than the non-subgoal group, subgoal learning might have helped students who would typically drop out of the course to remain in the course and be more successful. Further analysis of students who dropped out of the course and any common characteristics that they share would be needed to determine whether this is the most likely cause of the differences between groups.

5.2.3 Instructor experience. Morrison taught two sections of the introductory programming course for this study. Each section of students had a different culture. One section contained mostly computing majors, was held earlier in the day, and taught in a large auditorium classroom. The second section contained mostly students taking the course as a requirement for an engineering degree and was held late in the afternoon, allowing working professional students to attend.

The students in both sections expressed that the subgoals and the SLWEs done during class helped them to learn the material, either in anonymous comments in mid-term student surveys or through personal discussions. While working through the SLWEs during class, students were asked to state what the next subgoal to be accomplished would be, or what the code would be to accomplish a specific subgoal. By having the students continually verbalize the subgoal labels and associated code, it was hoped that the students would internalize the subgoals.

The most rewarding use of teaching with subgoal labels occurred at the end of the semester when covering arrays. While explaining the typically difficult topic of references versus primitives with shallow and deep copies, the notion of revisiting subgoals from assignment statements proved especially helpful. When walking through code and bringing back the assignment subgoal labels, the students could quickly determine what needed to be done or whether the code was correct by looking at the data types of the variables involved. Reminding the students to evaluate the data type of the variables involved in the statements allowed them to see if the action was being taken on an array element that was a primitive or reference type. This also proved beneficial on test questions that used nested [ ], such as

```
array[array[1] + array[2]] = 10
```

6 Conclusions

In this project, we used the TAPS protocol to identify the subgoals of problem-solving procedures that use expression/assignment statements, selection statements, loops, object instantiation and method calls, writing classes, and arrays in Java (see Figure 1). We then used those subgoals to design subgoal-labeled worked examples and paired practice problems to be used as the concepts were taught in an introductory programming course. To begin exploring the efficacy of the materials, we conducted a pilot test that compared quiz and exam performance of students who were taught with the subgoal materials and those who were taught with the conventional, non-subgoal materials for the course. Based on a sample of 265 students over the fall 2018 semester, we found that students who learned with the subgoal materials performed better on quizzes throughout the semester. This result suggests that the subgoal materials helped learners to solve problems using the procedure more effectively during the early stages of learning even though no performance difference between groups was found on the exams. We also found that students who learned with the subgoal materials were more likely to submit all of the exams (i.e., not drop out of the course). This finding paired with the finding that subgoal materials did not predict exam performance suggests that the subgoal materials helped more students to stay in the course and achieve equivalent exam performance as their peers. Moreover, average exam performance in the subgoal group had consistently less variance than that in the non-subgoal group, suggesting that the subgoal materials helped to equalize performance across students.

Though these results are promising, the pilot test has significant limitations. The instructor who was teaching with the subgoal materials was also part of the research team. This circumstance was necessary to fix any errors or overlooked details that would disrupt using the materials in class, but it also diminishes the validity of our results. The instructor is a veteran at teaching introductory programming and, thus, has significant prior experience, which helps to increase consistency of instruction and reduce bias. Some level of bias, however, is still likely to have been represented in the data. Now that the materials have been fully applied in a course, we will begin testing them in courses taught by instructors who are independent from the project. The promising results that we found in the pilot test suggest that further testing is worthwhile. If we can find the same pattern of results in more valid experimental settings, then we will have strong evidence that adopting the subgoal materials can improve learning in introductory programming courses. The materials are designed to be used in place of existing worked examples and practice problems, as they are currently used in a course. Thus, we expect that the barriers for adopting the materials will be low but offer substantial benefits, particularly, we hope, for students who are most likely to struggle.

7 ACKNOWLEDGMENTS

Our thanks to the reviewers who helped to improve this paper. We also thank the students who consented to be part of this study.

This work is funded in part by the National Science Foundation under grants 1712231 and 1712025. 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 NSF.

8 REFERENCES

[1] Anderson, J. R. (1996). ACT: A simple theory of complex cognition. *American Psychologist, 51*(4), 355.

[2] Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research, 70*(2), 181-214.

[3] Brown, N. C., & Wilson, G. (2018). Ten quick tips for teaching programming. *PLoS Computational Biology, 14*(4).

[4] Catrambone, R. (1998). The subgoal learning model: Creating better examples so that students can solve novel problems. *Journal of Experimental Psychology: General, 127*(4), 355.

[5] Catrambone, R. (2011). Task analysis by problem solving (TAPS): Uncovering expert knowledge to develop high-quality instructional materials and training. Paper presented at the 2011 Learning and Technology Symposium (Columbus, GA, June).

[6] Joentausta, J., & Hellas, A. (2018, February). Subgoal Labeled Worked Examples in K-3 Education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 616-621). ACM.

[7] Margulieux, L. E., Guzdial, M., & Catrambone, R. (2012). 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* (pp. 71-78). New York, NY: ACM.

[8] Morrison, B. B., Decker, A., & Margulieux, L. E. (2016). Learning loops: A replication study illuminates impact of HS courses. In *Proceedings of the Twelfth Annual International Conference on International Computing Education Research* (pp. 221-230). New York, NY: ACM.

[9] Morrison, B. B., Margulieux, L. E., & Guzdial, M. (2015). Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research* (pp. 21-29). New York, NY: ACM.

[10] Porter, L., Bailey Lee, C., Simon, B., & Zingaro, D. (2011). Peer instruction: Do students really learn from peer discussion in computing?. In *Proceedings of the Seventh International Computing Education Research Conference* (pp. 45-52). ACM.

[11] Renkl, A. (1997). Learning from worked-out examples: A study on individual differences. *Cognitive Science, 21*(1), 1-29.

[12] Schwonke, R., Renkl, A., Krieg, C., Wittwer, J., Aleven, V., & Salden, R. (2009). The worked-example effect: Not an artefact of lousy control conditions. *Computers in Human Behavior, 25*(2), 258-266.

[13] Sweller, J. (2006). The worked example effect and human cognition. *Learning and Instruction*.

[14] Trafton, J. G., & Reiser, B. J. (1993). Studying examples and solving problems: Contributions to skill acquisition. In *Proceedings of the 15th Conference of the Cognitive Science Society* (pp. 1017-1022).