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

Lauren E. Margulieux
Georgia State University

Briana B. Morrison
University of Nebraska at Omaha, bbmorrison@unomaha.edu

Adrienne Decker
University of Buffalo

Follow this and additional works at: https://digitalcommons.unomaha.edu/compscifacpub

Part of the Computer Sciences Commons

Please take our feedback survey at: https://unomaha.az1.qualtrics.com/jfe/form/SV_8cchtFmpDyGfBLE

Recommended Citation

This Article is brought to you for free and open access by the Department of Computer Science at DigitalCommons@UNO. It has been accepted for inclusion in Computer Science Faculty Publications by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
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

Brian 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:

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
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 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.
<table>
<thead>
<tr>
<th>Subgoals for evaluating and writing expression (assignment) statements</th>
<th>Subgoals for evaluating and writing selection statements</th>
<th>Subgoals for evaluating and writing loops</th>
<th>Subgoals for calling and writing methods</th>
<th>Subgoals for using objects and writing classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Determine whether data type of expression is compatible with data type of variable</td>
<td>1. Determine expression that will yield variable</td>
<td>1. Define how many mutually exclusive paths are needed</td>
<td>1. Determine purpose of loop</td>
<td></td>
</tr>
<tr>
<td>2. Update variable for pre based on side effect</td>
<td>2. Determine data type and name of variable and data type of expression</td>
<td>2. Order from most restrictive/selective group to least restrictive</td>
<td>a. Pick a loop structure (while, for, do_while)</td>
<td></td>
</tr>
<tr>
<td>3. Solve arithmetic equation</td>
<td>3. Determine arithmetic equation with operators</td>
<td>3. Write if statement with Boolean expression</td>
<td>2. Define and initialize variables</td>
<td></td>
</tr>
<tr>
<td>4. Check data type of copied value against data type of variable</td>
<td>4. Determine expression components</td>
<td>4. Follow with true bracket including action</td>
<td>3. Determine termination condition</td>
<td></td>
</tr>
<tr>
<td>5. Update variable for post based on side effect</td>
<td>5. Operators and operands must be compatible</td>
<td>5. Follow with else bracket</td>
<td>a. Invert termination condition to continuation condition</td>
<td></td>
</tr>
<tr>
<td>1. Determine expression that will yield variable</td>
<td></td>
<td>6. Repeat until all groups and actions are accounted for</td>
<td>4. Write loop body</td>
<td></td>
</tr>
<tr>
<td>2. Determine data type and name of variable and data type of expression</td>
<td>1. Define method header based on problem</td>
<td>a. Update loop control variable to reach termination</td>
<td>a. Determine types of logic (expression, selection, loop, etc.)</td>
<td></td>
</tr>
<tr>
<td>3. Determine arithmetic equation with operators</td>
<td>2. Define return statement at the end</td>
<td>b. Define internal variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Determine expression components</td>
<td>3. Define method body/logic</td>
<td>c. Write statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Operators and operands must be compatible</td>
<td>1. Name it</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subgoals for calling and writing methods</td>
<td>Subgoals for using objects and writing classes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A. Call or trace method calls</td>
<td>A. Use objects (creating instances)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Classify method as static method or instance method</td>
<td>1. Declare variable of appropriate class datatype.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. If static, use the class name</td>
<td>2. Assign to variable: keyword new, followed by class name, followed by ()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. If instance, must have or create an instance</td>
<td>3. Determine whether parameter(s) are appropriate (API)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Write (instance / class) dot method name and ( )</td>
<td>a. Number of parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Determine whether parameter(s) are appropriate</td>
<td>b. Data types of the parameters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a. Number of parameters passed must match method declaration</td>
<td>3. Differentiate class-level (static) vs. instance/object-level variables</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b. Data types of parameters passed must match method declaration (or be assignable)</td>
<td>4. Differentiate class-level (static) vs. instance/object behaviors/methods</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>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)</td>
<td>4. Define instance variables (that you want to be interrelated)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Evaluate right hand side of assignment (if there is one). Value is dependent on method’s purpose</td>
<td>5. Define class variables (static) as needed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Name it</td>
<td>6. Create constructor (behavior) that creates initial state of object</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Differentiate class-level (static) vs. instance/object-level variables</td>
<td>7. Create 1 accessor and 1 mutator behaviors per attribute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Differentiate class-level (static) vs. instance/object behaviors/methods</td>
<td>8. Write toString method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Define instance variables (that you want to be interrelated)</td>
<td>9. Write equals method</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Define class variables (static) as needed</td>
<td>10. Create additional methods as needed</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The instructional materials were pilot tested in the introductory programming courses which asks students to explain their solution to a peer and resolve working on practice problems, they can engage in Peer Instruction, solving with the SLWE which students learn about the problem example, instructors might use a flipped classroom approach in instruction 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.

### 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 as covariates or random factors in the analysis.

<table>
<thead>
<tr>
<th>Subgoal-Labeled Worked Example 1 – Simple arithmetic equation</th>
<th>Paired Practice Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int max = 100;</code>&lt;br&gt;<code>double tax = 0.5, result, bill = 26.12;</code>&lt;br&gt;Write the code to store max multiplied by tax in the variable result. SG1: <em>Determine expression that will yield variable</em>&lt;br&gt; <code>max * tax</code>&lt;br&gt;<code>SG2: Determine data type and name of variable and data type of expression</code>&lt;br&gt; Result to be stored in variable <code>result</code>. That variable is a <code>double</code>. The expression <code>max * tax</code> is an <code>int</code> multiplied by a <code>double</code>. A <code>double</code> can be assigned to a <code>double</code>. SG3: <em>Determine arithmetic equation with operators</em>&lt;br&gt;<code>result = max * tax;</code></td>
<td>Practice Problem 1:&lt;br&gt;Calculate a 15% tip on the bill. Practice Problem 2:&lt;br&gt;Determine the total amount owed including bill, tip, and tax.</td>
</tr>
</tbody>
</table>

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

5.2.2 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, \(d = 0.26, \eta^2(264) = 4.20, p = .04\). 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 disproportionately 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

\[ \text{array[array[1] + array[2]] = 10} \]

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


