When is Reading More Effective than Tutoring? An Analysis Through the Lens of Students' Self-Efficacy among Novices in Computer Science

Priti Oli
University of Memphis
Memphis, TN, USA
poli@memphis.edu

Rabin Banjade
University of Memphis
Memphis, TN, USA
rbanjade1@memphis.edu

Arun Balajiee Lekshmi Narayanan
University of Pittsburgh
Pittsburgh, PA, USA
arl122@pitt.edu

Peter Brusilovsky
University of Pittsburgh
Pittsburgh, PA, USA
peterb@pitt.edu

Vasile Rus
University of Memphis
Memphis, TN, USA
vrus@memphis.edu

ABSTRACT

Self-efficacy, or the belief in one's ability to accomplish a task or achieve a goal, can significantly influence the effectiveness of various instructional methods to induce learning gains. The importance of self-efficacy is particularly pronounced in complex subjects like Computer Science, where students with high self-efficacy are more likely to feel confident in their ability to learn and succeed. Conversely, those with low self-efficacy may become discouraged and consider abandoning the field. The work presented here examines the relationship between self-efficacy and students learning computer programming concepts. For this purpose, we conducted a randomized control trial experiment with university-level students who were randomly assigned into two groups: a control group where participants read Java programs accompanied by explanatory texts (a passive strategy) and an experimental group where participants self-explain while interacting through dialogue with an intelligent tutoring system (an interactive strategy). We report here the findings of this experiment with a focus on self-efficacy, its relation to students' learning gains (to evaluate the effectiveness, we measure pre/post-test), and other important factors such as prior knowledge or experimental condition/instructional strategies as well as interaction effects.

Keywords

tutoring, self-explanation, self-efficacy, reading, novices

1. INTRODUCTION

Self-efficacy plays a key role in determining the effectiveness of instructional methods for inducing learning gains, especially in complex subjects like Computer Science. Students with high self-efficacy are more likely to be confident in their ability to learn and succeed, which can lead to better academic performance and a successful professional career. On the other hand, low self-efficacy may lead to discouragement and even the decision to abandon the field.

Indeed, Computer Science is a complex subject matter. According to Morrison et al., [18], using textual languages to identify variables and maintain track of their values while also comprehending and managing an external agent (the computer) entails a degree of complexity not found in science, math, or engineering. The difficulty of Computer Science topics is well documented. Several researchers [17, 12, 7] have reported that students often perceive computer programming courses as being among the most challenging due to the difficulty of learning how to program compared to other subjects. Another testament to this is the high attrition rates in introductory CS courses (e.g., CS1 and CS2) [5], which is quite often due to frustrations students develop towards programming concepts resulting in a negative perception of Computer Science as a discipline [25]. Our work presented here and our larger goal of developing adaptive instructional technologies to help students in CS1 and CS2 courses is meant to address those challenges, help students learn, and improve retention. One adaptive aspect of the underlying computer-based intervention is to tailor instruction to students based on their self-efficacy level and other factors such as prior knowledge. For instance, low-efficacy students may need more support in the form of hints while self-explaining code as well as more encouragement in the form of positive feedback and engaging and motivational elements such as an open social learner model displaying the performance of aspirational peers in the form of virtual peers who impersonate a role model. It is beyond the scope of this paper to address all those elements of our more extensive project. Instead, we focus on self-efficacy and its role in mediating code comprehension and learning.

Self-efficacy was defined by Bandura as "people's judgments of their capability to organize and execute courses of action required to attain designated types of performance" [3].

Proceedings of the 7th Educational Data Mining in Computer Science Education (CSEDM) Workshop, March 2023

The relationship between self-efficacy and performance is straightforward. Students with high self-efficacy are more confident in their ability to learn and succeed, which positively affects their motivation and engagement and, ultimately, their overall performance. Such students may need fewer scaffolding of their code comprehension and learning. The rate of progress and overall performance is mediated by other factors such as prior knowledge and instructional strategy. For instance, students with higher prior knowledge will more likely perform well early on, e.g., early successful steps on whatever instructional tasks they are working on, and make steady progress towards their learning goal, which will have a positive effect on their satisfaction, motivation, and self-efficacy thus leading to a self-fulfilling positive feedback loop. On the other hand, students with low self-efficacy and low prior knowledge may benefit more from personalized and supportive approaches, like scaffolded self-explanations through one-on-one tutoring. The role of the human or computer-based tutor would be to monitor students' comprehension and learning processes and intervene with supporting feedback and hints to help students achieve early successes in various learning tasks and thus provide satisfaction and increase their motivation and self-efficacy. Such support is much needed until they gain a critical mass of knowledge and a level of self-efficacy that will enable them to continue successfully toward their learning and long-term career goals. Other factors are important as well, such as task-level adaptation of instruction. For instance, selecting too difficult instructional tasks, even for students with higher prior knowledge and high self-efficacy, may negatively affect self-efficacy. In such cases, more support from the human or computer-based tutor could overcome the mismatch between the task's difficulty and the student's knowledge and self-efficacy.

Our work presented here is part of our larger goal of understanding the relationship between factors such as self-efficacy and prior knowledge, students' majors, instructional strategies such as scaffolded self-explanations, and outcomes such as comprehension, learning, and retention. Specifically, this paper presents a study that examines the relationship between self-efficacy and student learning for students in intro-to-programming classes. Even more specifically, we focus on scaffolding students' code comprehension, which can positively impact their learning and overall performance and retention. There are two solid reasons for concentrating on code comprehension to improve students' learning and self-efficacy. *First*, source code comprehension is critical for both learners and professionals. Students learning computer programming spend a significant portion of their time reading or reviewing someone else's code (e.g., source code examples from a textbook or provided by the instructor). Furthermore, it has been estimated that software professionals spend at least half of their time analyzing software artifacts to comprehend computer source code. Reading code is the most time-consuming activity during software maintenance, consuming 70% of the total lifecycle cost of a software product [6]. O'Brien notes that source code comprehension is required when a programmer maintains, reuses, migrates, reengineers, or enhances software systems [20]. Therefore, offering support to improve learners' source code comprehension skills will have lasting positive effects on their academic success and future professional careers.

Second, learning to program is perceived as difficult because of too much and too soon emphasizes writing code rather than reading and understanding code. Indeed, this could lead to a lack of foundational conceptual understanding, making it more challenging for students to effectively learn and apply programming concepts [15]. Therefore, reading and tracing first before writing code is an approach that is gaining more attention and is being explored by many researchers and educators. To prepare students for writing code, activities like code tracing and code reading should be used because they allow beginners to develop a conceptual understanding of basic programming skills with a lower cognitive load than writing code itself. This will positively impact their learning and overall retention rates in Computer Science courses.

Another challenge in Computer Science education, and in particular, for intro-to-programming courses, is the need for scale as more and more Computer Science and non Computer Science students enroll in such courses. One way to address the scalability issue is to use learning technologies that can serve many students. The most advanced learning technologies are intelligent tutoring systems that can provide tailored, one-on-one instruction to every student. It is well documented that one-on-one instruction, i.e., tutoring, is among the most effective instructional methods. When coupled with scalable, computer-based technologies, it results in scalable, effective adaptive instructional systems, which is our focus.

To this end, the study presented here investigates the effectiveness of scaffolded self-explanation through tutoring as a strategy for code comprehension and learning of programming concepts and compares it to a self-guided reading strategy while accounting for other factors such as students' prior knowledge and self-efficacy.

More specifically, we experiment with a novel teaching and learning environment for Computer Science education in the form of an Intelligent Tutoring System (ITS) called *DeepCodeTutor* designed to help learners develop code comprehension skills and master programming concepts via scaffolded self-explanations. The proposed DeepCodeTutor has been built on strong theoretical foundations such as self-explanation theories [8] and the Socratic method of instruction in which a sequence of guiding questions scaffold students' code comprehension and learning processes.

The rest of the paper is organized as in the following. Next, related, most relevant work is briefly presented. The main instructional tasks and the targeted computer science concepts, such as loops and arrays, are presented. The description of the experimental system follows, and then the experimental design is presented. Results, Conclusions, and Future Work are the last sections of the paper.

2. RELATED WORK

We briefly highlight prior work on scaffolded self-explanation as a learning strategy as well as comparative studies on the effectiveness of this approach compared to reading worked-out examples. We also discuss prior studies on the role of self-efficacy in learning.

Self-explanation is explaining learning material to oneself through speaking or writing [16]. Self-explanation is an effective strategy to assist students in learning, understanding, and comprehending the target material [8]. Studies with undergraduate [21] and high school students [1] have found that those who applied self-explanation techniques while studying worked-out examples performed better on programming tasks in Visual Basic compared to those who did not use these strategies. In two other studies, [29, 19], a strong correlation between the ability to trace programs and explain code in plain English and the ability to write code on paper in exam settings has been reported. Additionally, using self-explanation strategies has shown improvement in comprehension of programming concepts in introductory computer science courses [24]. In a comparative study, [24] found that the Socratic method of guided self-explanation is more effective than free self-explanation in teaching novices code comprehension.

Although prior research suggests that increased student interaction, i.e., a more interactive learning/teaching strategy as in one-on-one tutoring, leads to improved learning outcomes [28], it is possible that other factors, such as motivation or verbal fluency, could be influencing both the student's interaction, e.g., during tutoring, and their learning gains. Our work will help better understand the role of interactive learning and its relation and interaction with those factors such as motivation, self-efficacy, and prior knowledge and their impact on various student outcomes.

Cronbach and Snow [10] found that highly competent students typically learn effectively using a range of instructional methods, while students with lower levels of competence may benefit more from more structured or scaffolded instruction. VanLehn and colleagues [28] compared the effectiveness of different forms of instruction, which included human tutoring (spoken and computer-mediated), natural language-based computer tutoring, and text-based control conditions, and found tutorial dialogue to be more beneficial for novices studying content written for intermediates. However, when novices studied material written for novices or intermediates studied material written for intermediaries, the tutorial dialogue was not consistently more effective than text-based control conditions, which is confirmed by our study presented here. This suggests that the effectiveness of tutorial dialogue may depend on the level of preparation of the student and the match between the student's preparation and the content of the instruction [11].

According to Bandura, [3], an individual's efficacy perceptions and beliefs about a course of action affect an individual's decision to pursue or continue a task, level of effort put forth on the task, tenacity in the face of task challenges, and overall task performance. Bandura further suggests that individuals with high domain self-efficacy are more likely to decide to take on complex work, put up more effort to complete it, and continue when the activity becomes more difficult. According to a review published in 2016 [4], self-efficacy is the key factor in predicting the academic success of university students. Lewis et al. [13] conducted a study in which they interviewed students to understand the factors that influence student's decision to major in computer science. The study found that students' self-assessment of their programming ability is important for their persistence in the field and that this assessment is based on their perceived prior experience, the speed at completing programming tasks, and grades in computer science courses.

This paper builds on this prior work and explores the role of self-efficacy in students' learning of programming concepts and its interaction with instructional strategies such as scaffolding self-explanations.

3. CONTENT: INTRO-TO-PROGRAMMING TASKS

For the main tasks in our experiment, we used a selection of Java code examples from the DeepCode codeset [22]. The DeepCode codeset is a collection of Java code examples that have been annotated with explanations and can be used for various purposes, including code comprehension and learning tasks for students in introductory programming courses. The DeepCode codeset was chosen for its strong theoretical foundations, which are based on various theories, including code comprehension and self-explanation theories and the ICAP framework [27, 9]. It was also explicitly designed to be used to develop Intelligent Tutoring Systems that scaffold students' code comprehension processes. These two factors make it an ideal choice for the code comprehension tasks in our study. The code examples in the DeepCode codeset include most topics in introductory programming courses, such as operators, loops, arrays, methods, and classes.

It should be noted that the explanations of the Java code examples are divided into two major types: logical step and logical step details. These correspond to the domain and program models, respectively, of major code comprehension theories. The logical step details comments also link the domain model with the program model, thus, corresponding to the integrated model in code comprehension theories. The code examples were designed to minimize the domain knowledge needed for full understanding and center around relatable, world knowledge tasks or contexts, such as determining if a given year is a leap year. There is another type of explanation called statement-level explanations, which focus on the new concept introduced in each example. The code examples are sequenced such that each introduces only one new concept and only relies on previously mastered concepts.

Additionally, the DeepCode codeset also includes scaffolding questions logical and statement-level expert explanations, which human or computer-based instructors can use as hints to scaffold students' comprehension and learning based on the socio-constructivist theory of learning. These hints in the form of questions are progressive, starting with vague hints and becoming more informative, eventually providing fill-in-the-blank type hints. This socio-constructivist approach allows the student to construct their knowledge by themselves as much as possible, providing help in the right dosage through hints only the student is floundering.

4. SYSTEM DESCRIPTION

For the experiment, we used an online experimental system with two main tracks. On one track, the experimental system shows students Java code examples with expert expla-

nations. For the other track, the online experimental system activates a conversational ITS called DeepCodeTutor. DeepCodeTutor presents students with one Java code example at a time, along with its overall goal, i.e., what the code examples are supposed to achieve (what 'business' problem is being solved), and asks them to self-explain the code. The student's initial self-explanation is then automatically evaluated using automated semantic similarity methods, which compares the student's explanation to expert-provided explanations, e.g., the expert explanations in the DeepCode code examples. The semantic similarity is calculated at the sentence and paragraph level by comparing a variety of features, including an alignment score based on the optimal alignment of the sentences using chunks and a branch-and-bound solution to the quadratic assignment problem, word embeddings, Unigram overlap with synonym check, bigram overlap and BLEU score [23]. If the similarity score is 0.5 or higher, the student's explanation is correct, and they proceed to the next task. If the similarity score is between 0.4 and 0.5, the student's explanation is partially correct, and scaffolding is provided for the incorrect parts. The threshold value was selected based on previous studies [14].

The goal of DeepCodeTutor is to help students comprehend and explain the logical step and logical step details of a given code. If a student provides a correct and complete explanation, they will receive positive feedback and a summary explanation of what the code does. If the student is missing important aspects or has misconceptions, DeepCodeTutor will use scaffolding questions to guide their comprehension and learning. The number of hints provided may vary depending on the student's needs, understanding, and articulation. The system will assert the explanation if the student cannot give a correct and complete explanation even after scaffolding.

The user interface of TutorApp consists of the following components. The goal description for the Java code example is displayed in the top left corner of the app and is highlighted in red for immediate attention and easy visibility for students (Fig. 1, A). The interactive code editor (Fig. 1, B) displays the target code example the student should read in order to understand, comprehend, and articulate. The code example is divided into logical blocks/chunks separated by empty lines. When a question is asked about a specific block/line of code, as shown in the figure, the target block is highlighted in yellow. On the right side of the interface (Fig. 1, C), is a display box that shows the entire dialogue history displaying the student's response in blue on the right, while the tutor's response is in green on the left. The student input box is at the bottom right corner of the interface(Fig. 1, D). It is beyond the scope of this paper to present all the details of DeepCodeTutor's design.

5. EXPERIMENT DESIGN

A randomized controlled trial experiment was conducted to explore the effectiveness of the scaffolded self-explanation strategies delivered through an online conversational tutoring system called TutorApp. A control experimental condition involved reading code examples from the DeepCode codeset, annotated with explanations by experts. Students were randomly assigned to one of the two experimental conditions. The two groups were given the same Java code examples. That is, the two groups were exposed to equivalent content. Participants in both groups were tested with respect to their mastery of the targeted concepts. The pre- and post-tests consisted of predicting the output of five Java code examples which were aligned in content with the four main comprehension tasks of the experiment. Students' performance on those tests was used to compute the main outcome variable: normalized learning gains. The experimental group was asked to self-explain the code and received scaffolding as needed, while the control group read the annotated code examples. For the experimental group, the first main task after the pre-test served as a modeling task, i.e., it illustrated what explanations of code should look like.

5.1 Protocol

The overall experiment protocol was as follows. First, students were debriefed about the experiment, given a chance to ask questions, and then asked to sign a consent form if they agreed to proceed. Then, they completed a background questionnaire about their primary language of communication, programming experience, and current major. This was followed by a self-efficacy questionnaire and a pretest, which assessed students' prior knowledge of the targeted programming concepts that would be covered in the main task. The main task refers to the five main code-reading tasks. The tasks targeted the following programming concepts: variables and operator precedence, nested if-else statements, loops, arrays, and creating objects and using their methods. After completing the main task, the students took a posttest targeting the same concepts that were covered in the pretest and main task. Finally, the students completed an evaluation survey to provide their perceptions of TutorApp. The system logged all student inputs and tracked the time associated with each action.

5.2 Participants

We recruited 90 students from an introductory Java Programming class in an undergraduate Computer Science program at a large public university in the United States. The study was conducted online. The participants were compensated with gift cards and extra credit for their participation.

The study was conducted by providing clear instructions for accessing and navigating through the system. Students were asked to share their screens to ensure that they followed the instructions correctly. A graduate student was also available to answer questions during the experiment. The participants were randomly assigned to the control and experimental group. Out of all the participants, 14 identified themselves as non-native English speakers. The participants were enrolled in different majors, including Computer Science, Data Science, Computational Biology, Physics, Statistics, Engineering, Computational Social Science, Economics, and Statistics. Out of the 90 participants, 89 completed the task, with 47 randomly assigned to the control group and 42 to the experimental group.

5.3 Instrumentation of Self-Efficacy

In our study, we assess students' self-efficacy using a self-reporting survey. This survey focuses on students' ability to learn and perform well in computer science courses, specifically their problem-solving confidence, debugging confidence, confidence in mastery, and confidence in receiving a

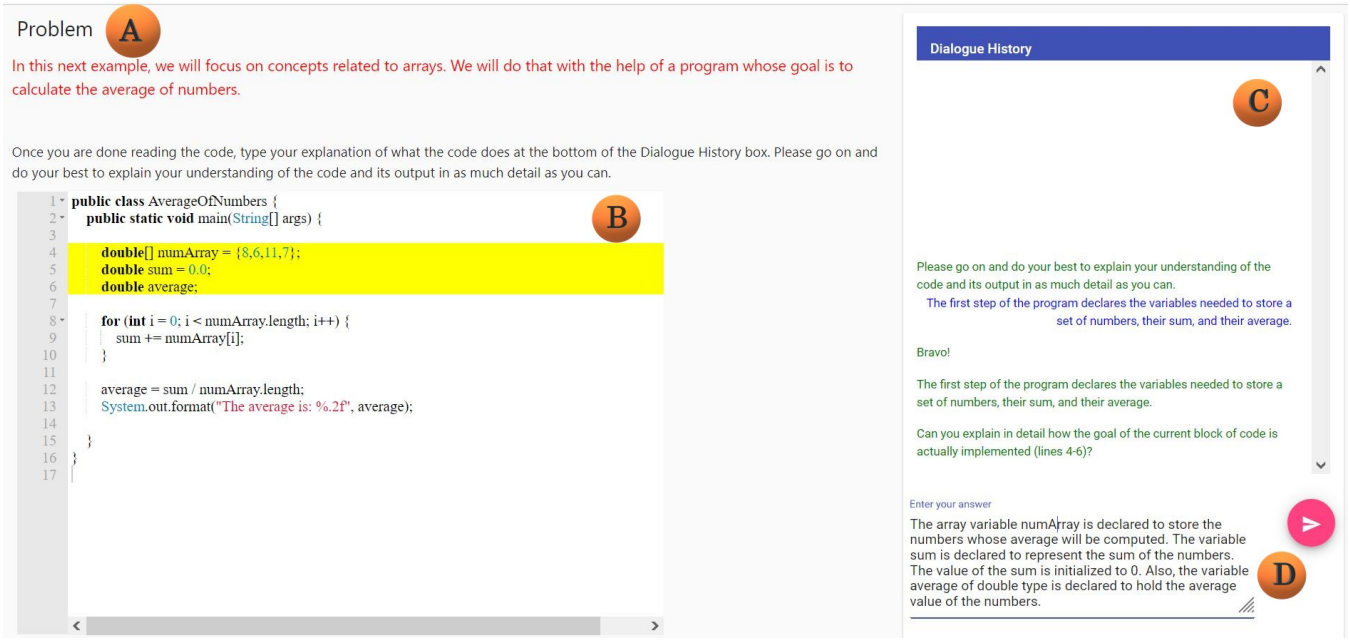

Figure 1: A screenshot of the TutorApp Interface: It includes (A) The goal description for each task, (B) an interactive Java code editor that shows the current Java code example, and (C) a dialogue history of the interaction between the tutor and learners, and (D) an input box for the learner to type their responses.

good grade (i.e., success in the course) and are adapted from Askar and Davenport's computer programming self-efficacy scale [2]. The student's response was recorded on a 5-point Likert scale: (1) Strongly disagree; (2) Disagree; (3) Neither agree nor disagree; (4) Agree; (5) Strongly agree for different questions. All questions were formulated with positive wordings. The questions that were used in the self-efficacy survey can be found in Table 2.

The Cronbach's Alpha for the ten items among all participants was found to be 0.85, which indicates that response values for each participant across all the items are consistent. To calculate the average self-efficacy score for each participant, the average of the responses to all questions was taken.

5.4 Evaluation

Normalized learning gain (LG) [26] was used as the main outcome metric to assess the effectiveness of two different learning strategies (among 5 pretest/posttest questions), which was calculated as follows:

$$LG = \begin{cases} \frac{posttest - pretest}{5 - pretest} & \text{if posttest} > \text{pretest} \\ \frac{posttest - pretest}{pretest} & \text{if posttest} < \text{pretest} \\ discard & \text{if posttest} = \text{pretest} = 5 \text{ or } 0 \\ 0 & \text{if posttest} = \text{pretest} \end{cases}$$

6. RESULTS

We start the *Results* section with an analysis of participants' self-efficacy. Table 1 provides an overview of students' responses to the self-efficacy questionnaire. Overall, the average self-efficacy score is relatively high, indicating our sample is biased toward high self-efficacy students. The average for the experimental condition is lower than for the control condition but not significantly lower.

Group	N	Average	S.D	Median
Overall	89	3.96	0.48	4
Experiment	42	3.92	0.48	4
Control	47	4.01	0.47	4.1

Table 1: **An overview of Average, Standard Deviation (S.D) and Median of Self-Efficacy for different Groups**

The average scores across all participants for all the self-efficacy questions are shown in Table 2. The control group scores are higher for all questions except the last one.

Participants demonstrated the highest levels of self-efficacy in their ability to understand Java conditional expressions and trace well-defined iterative statements in Java. Conversely, the participants reported the lowest levels of self-efficacy with respect to their mastery of Java programming concepts and their ability to achieve an excellent grade in a Java programming class. For the targeted concepts in our main experimental tasks (variables and operator precedence, nested if-else statements, loops, arrays, and creating objects and using their methods), self-efficacy scores are quite high (around 4.0 or above).

Self-efficacy versus Prior Knowledge

Students' level of expertise, which is approximated in our case by their prior knowledge as measured by the pre-test, should have a significant impact on their self-efficacy. Table 3 shows the average self-efficacy score for each level of pre-test score.

Item	Overall Avg(S.D)	Control Avg (S.D)	Exp. Avg (S.D)
I believe I will receive an excellent grade in Java programming class.	3.7 (0.88)	3.74 (0.93)	3.66 (0.83)
I have mastered the concepts taught in the Java programming class.	3.49 (0.8)	3.57 (0.81)	3.4 (0.78)
I can read Java programs and make changes to them according to some specified requirements.	4 (0.7)	4.10 (0.75)	3.88 (0.62)
I can write Java programs if given a specified set of requirements.	4.07 (0.7)	4.14 (0.68)	4 (0.72)
I can mentally trace through the execution of a long, complex Java program given to me.	3.47 (0.79)	3.55 (0.67)	3.38 (0.89)
I can understand Java's conditional expressions (e.g. if...else..).	4.56 (0.53)	4.57 (0.53)	4.54 (0.54)
I can mentally trace well-defined iterative statements in Java (e.g. for loop and while loop).	4.23 (0.68)	4.29 (0.61)	4.16 (0.75)
I can understand the concepts of objects and classes in Java, given a well-defined declaration of a Java class.	3.94 (0.67)	3.95 (0.68)	3.92 (0.66)
I understand how the array data structure works in Java and how to use it when coding.	4.21 (0.79)	4.29 (0.61)	4.11 (0.95)
I can debug(correct the errors) a long and complex program that I had written and make it work.	3.92 (0.76)	3.91 (0.79)	3.92 (0.73)

Table 2: **Average and standard deviation (in parentheses) of self-efficacy per item.**

Pretest Score	N	Avg	S.D
5	36	4.13	0.4
4	23	4.02	0.31
3	16	3.95	0.59
2	3	3.46	0.12
1	7	3.45	0.49
0	4	3.37	3.29

Table 3: **Average and Standard Deviation(S.D) of self-efficacy grouped according to pretest score.**

Overall, the pre-test measured a moderate positive Pearson's correlation of 0.47 between self-efficacy and prior knowledge. We also asked students to self-report their years of programming experience, which can be viewed as a proxy for their prior knowledge. The correlation between self-efficacy and programming experience was found to be 0.45. Those results suggest that students are relatively good at self-reporting their prior knowledge.

CS versus non-CS majors: Self-efficacy and Prior Knowledge Differences

Since our student pool included CS and non-CS majors, we analyzed those two groups' self-efficacy and prior knowledge. Table 4 shows summary statistics for the self-efficacy of those two groups. The difference in self-efficacy is significant. Table 5 shows the average pre-test score for CS and non-CS majors. We observe a significant difference in prior knowledge based on the t-test value.

Major	N	Mean	S.D	t-val	Sig.
CS	56	4.05	0.44	2.33	0.01
Non-CS	33	3.80	0.52		

Table 4: **Self-Efficacy Comparison on CS Major Vs Non-CS Major**

Major	N	Mean	S.D	t-val	Sig.
CS	56	4.01	1.3	2.28	0.01
Non-CS	33	3.27	1.58		

Table 5: **Pretest Comparison on CS Major Vs Non CS Major**

Self-Efficacy and Learning Gains

Students' learning is measured by the normalized learning gains metric mentioned earlier. Furthermore, we characterize their performance by their post-test score. In this section, we look at the impact of self-efficacy on learning gains and post-test scores. A Pearson correlation of 0.19 was found between self-efficacy and learning gains. While this is a modest correlation, we should consider the bias toward high self-efficacy students in our sample. We plan to further investigate with correlation with larger, less biased samples. The correlation between self-efficacy and post-test was 0.56, which shows a considerable positive correlation, and, importantly, it improved relative to the correlation between self-efficacy and pre-test.

The role of the intervention/strategies (reading versus scaffolding self-explanation) and their interaction with self-efficacy, and their cumulated impact on learning will be explored next.

What is the impact of the intervention on learning gains?

For the learning gain (normalized learning gain) analysis, data from 21 participants in the control group and 11 participants in the experimental group were excluded because they had perfect pretest and posttest scores. Similarly, One participant's data was omitted from the analysis of learning gain because they scored 0 on both the pretest and the posttest. After eliminating these participants, the remaining participants in the study had an average pretest score of 3 (N=26, S.D = 1.23) in the control group and around 2.9 (N=30, S.D=1.41) in the experimental group. A t-test indicated that the two groups had similar levels of prior knowledge and that the necessary assumptions for the test

(such as a continuous scale for the dependent variable and a normal distribution) were met. The results of the t-test can be found in Table 6. To better understand the effect size of DeepCodeTutor in the null result, we calculated Cohen's d for the learning gain, which was found to be a small effect size of 0.19 in favor of scaffolded self-explanation using DeepCodeTutor.

Similarly, Table 7 shows the average self-efficacy for each group and indicates no significant difference between the average self-efficacy of the two groups.

Group	N	Mean	S.D	t-val	Sig.
Experimental Group	30	3.0	1.41	-0.8	0.21
Control Group	26	3.19	1.23		

Table 6: **Independent sample t-test for pretest between experimental and control group.**

Group	N	Mean	S.D	t-val	Sig.
Experimental Group	30	3.87	0.49	-0.39	0.35
Control Group	26	3.93	0.51		

Table 7: **Independent sample t-test for difference in average self-efficacy between experimental and control conditions.**

Group	N	Mean	S.D	t-val	Sig.
Low S.E	26	0.23	0.46	-0.51	0.3
High S.E	27	0.30	0.48		

Table 8: **Learning Gain Comparison on Low Self-Efficacy (S.E) Vs High Self-Efficacy (S.E) Split on Median**

Our results indicated that there was no significant difference in learning gain between the control group and the experimental group. However, both groups of students demonstrated learning gains when using the content from the DeepCode codeset, whether through reading the commented code examples or through scaffolded self-explanation, as shown in Table 9. The results of the t-test shown in the table indicate no significant difference in learning between the two groups. An ANCOVA analysis with learning gains as an independent variable, the condition as the grouping variable, and self-efficacy as a covariate found no statistically significant difference in learning gains between the two states when adjusted for self-efficacy ($p = 0.68$).

One possible reason for the non-significant difference in learning gains may be the relatively low difficulty of the main tasks compared to students' mastery levels. Indeed, most students' pre-test scores were quite high (4 or 5 out of a perfect score of 5), which means our sample was biased towards the high-knowledge, high-self-efficacy students. This may be the case that we recruited the students at the end of the semester after an entire semester of an opportunity to master the intro-to-programming concepts. That is, the majority of the students in our sample are not novices anymore but rather intermediaries. Furthermore, students' self-reported perception task difficulty indicates that they didn't find them difficult. As shown in Figure 2, which shows the perception of the difficulty of questions in the pre/post-test and main task, both groups of students reported that the questions were at or below their level. Previous research by VanLehn [28] suggests that when intermediate learners are given intermediate-difficulty tasks and novice learners are given tasks at their difficulty level, their learning in an interactive condition is similar to that of reading. This could have contributed to the lack of difference in learning gain between the two groups in our study.

There may have been several factors that impacted students' learning, such as their self-efficacy. We later examine self-efficacy's role in learning and post-test performance after reporting learning gains for CS and non-CS majors.

Group	N	Mean	S.D	t-val	Sig.
Experimental Group	30	0.26	0.40	0.34	0.33
Control Group	26	0.22	0.54		

Table 9: **Independent sample t-test for Learning Gain**

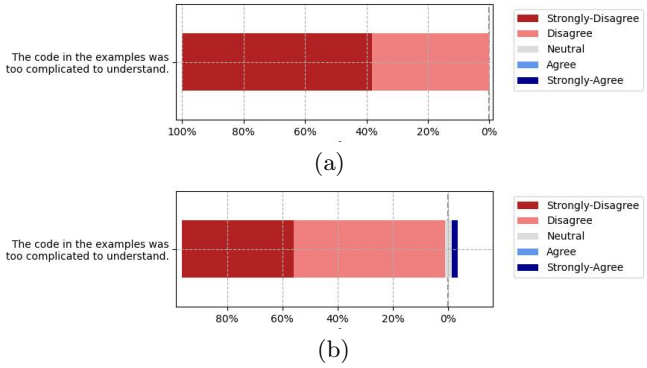

Figure 2: A diverging plot illustrating the responses of students to the difficulty of questions in a pre/post-test and main task, with the left side representing negative responses and the right side representing positive responses for a)Control group b) Experiment group

CS vs Non-CS Majors: Learning Gains

Major	N	Mean	S.D	t-val	Sig.
CS	31	0.30	0.51	1.06	0.14
Non-CS	25	0.17	0.41		

Table 10: **Learning Gain Comparison on CS Major Vs Non CS Major**

Table 10 indicates no statistically significant difference between CS and non-CS majors regarding learning gains. The mean learning gains for CS majors are almost twice as that for non-CS majors. The difference is not significant when accounting for the experimental condition, as shown in Tables 11. For non-CS majors, the learning gains are larger in the experimental condition, whereas in the control condition, the average learning gain for non-CS majors is quite low. While inconclusive, those findings may suggest that non-CS majors benefit much more from the interactive strategy. Further studies with a larger and less biased sample is needed for a more concluding result.

Group	Major	N	Mean	S.D	t-val	Sig.
Experimental	CS	14	0.33	0.48	0.76	0.22
	Non-CS	16	0.21	0.33		
Control	CS	17	0.28	0.55	0.8	0.21
	Non-CS	9	0.10	0.53		

Table 11: **Learning Gain Comparison on CS Major Vs Non CS Major Control**

How does a student's self-efficacy influence their learning when using different learning strategies, explicitly reading versus scaffolded self-explanation?

We divided students in each condition into two subgroups based on their self-efficacy scores: the low-self-efficacy and high-self-efficacy subgroups. The threshold for determining which subgroup a student belonged to was the median self-efficacy score, which was 3.9 for both groups (this median value was obtained after discarding data for normalized learning gain analysis). This split resulted in significant differences in pretest scores between the subgroups (p-value = 0.0009 in the control group and p-value = 0.007 in the experimental group), supporting our decision to divide the groups based on self-efficacy.

Group	Self-Efficacy	N	Mean	S.D	t-val	Sig.
Experimental	Low	14	0.30	0.54	0.46	0.46
	High	14	0.31	0.55		
Control	Low	11	0.16	0.5	0.9	0.37
	High	11	0.36	0.53		

Table 12: **Learning gain of high and low self-efficacy students in experimental versus control groups.**

Table 12 shows no significant difference in learning between students with low and high self-efficacy in the experimental group, suggesting that scaffolded self-explanation is uniformly effective for all students. That is, the scaffolded self-explanation leads to similar learning gains for the low-efficacy students as for the high-self-efficacy students. Interestingly, high-efficacy students have similar mean learning gains in the two conditions. Importantly, table 12 also shows that students with high self-efficacy had twice the learning gain of those with low self-efficacy when just reading explanations of code examples, although this difference was not statistically significant. This may imply that the interactive strategy helps the low-efficacy students twice as much as the more passive strategy of reading experts' code explanations. The difference in learning, while large, is not significant and may be the result of the small sample size. This is something we will need to explore with a bigger sample size in the future.

7. CONCLUSION AND FUTURE WORK

In this study, we examined the relationship between self-efficacy, instructional strategies, learning and other factors, such as prior knowledge of the domain of computer programming while students engage in code comprehension tasks. The results of the experiment show a strong relationship between self-efficacy and prior knowledge which means students with low prior knowledge typically have low self-efficacy which in turn, for instructional purposes, it means they will need more support in the form of hints and feedback as well as motivational support and instructional tasks that are closer to their level of mastery. Furthermore, the interactive instructional strategy of scaffolded self-explanations helped equally students with low and high self-efficacy, whereas the more passive strategy of just reading seemed to benefit more the high-knowledge students although the results were not significant probably due to the small sample size.

Our results suggest that self-efficacy does not significantly impact students' learning. For future work, we plan to investigate further the impact of various aspects of self-efficacy on learning gain with a larger sample of students. Furthermore, since our participant group was biased towards higher self-efficacy, we plan to either change the timing of our experiment, e.g., in the middle of the semester before students had too many opportunities to master the target concepts, or a better selection of code examples, i.e., selecting tasks that are more challenging given students' level of mastery at the end of the semester (intermediate level content for novices and advanced content for intermediaries).

In addition to learning gain, we also aim to examine the relationship between various aspects of the self-explanation-based tutoring strategy and self-efficacy, such as differences in the self-explanation of code examples between students with high self-efficacy and those with low self-efficacy. While our study was based on tasks that were completed in a single session, a more comprehensive study in a classroom setting over a semester could provide further insights into the relationship between self-efficacy and learning gain. We plan to continue our research in this direction.

ACKNOWLEDGMENTS

This work has been supported by the following grants awarded to Dr. Vasile Rus: the Learner Data Institute (NSF award 1934745); CSEdPad: Investigating and Scaffolding Students' Mental Models during Computer Programming Tasks to Improve Learning, Engagement, and Retention (NSF award 1822816), and Department of Education, Institute for Education Sciences (IES award R305A220385). The opinions, findings, and results are solely the authors' and do not reflect those of NSF or IES.

8. REFERENCES

[1] R. Alhassan. The effect of employing self-explanation strategy with worked examples on acquiring computer programing skills. *Journal of Education and Practice*, 8(6):186–196, 2017.

[2] P. Askar and D. Davenport. An investigation of factors related to self-efficacy for java programming among engineering students. *Online Submission*, 8(1), 2009.

[3] A. Bandura. Social foundations of thought and action. *Englewood Cliffs, NJ*, 1986(23-28), 1986.

[4] K. Bartimote-Aufflick, A. Bridgeman, R. Walker, M. Sharma, and L. Smith. The study, evaluation, and improvement of university student self-efficacy. *Studies in Higher Education*, 41(11):1918–1942, 2016.

[5] J. Bennedsen and M. E. Caspersen. Failure rates in

introductory programming: 12 years later. *ACM inroads*, 10(2):30–36, 2019.

[6] B. Boehm and V. R. Basili. Top 10 list [software development]. *Computer*, 34(1):135–137, 2001.

[7] M. E. Caspersen and M. Kolling. Stream: A first programming process. *ACM Transactions on Computing Education (TOCE)*, 9(1):1–29, 2009.

[8] M. T. Chi, N. De Leeuw, M.-H. Chiu, and C. LaVancher. Eliciting self-explanations improves understanding. *Cognitive science*, 18(3):439–477, 1994.

[9] M. T. Chi and R. Wylie. The icap framework: Linking cognitive engagement to active learning outcomes. *Educational psychologist*, 49(4):219–243, 2014.

[10] L. J. Cronbach and R. E. Snow. *Aptitudes and instructional methods: A handbook for research on interactions*. Irvington, 1977.

[11] X. Fang. Application of the participatory method to the computer fundamentals course. In *Affective Computing and Intelligent Interaction*, pages 185–189. Springer, 2012.

[12] Ö. Korkmaz and H. Altun. Adapting computer programming self-efficacy scale and engineering students' self-efficacy perceptions. *Participatory Educational Research*, 1(1):20–31, 2014.

[13] C. M. Lewis, K. Yasuhara, and R. E. Anderson. Deciding to major in computer science: a grounded theory of students' self-assessment of ability. In *Proceedings of the seventh international workshop on Computing education research*, pages 3–10, 2011.

[14] M. C. Lintean and V. Rus. Measuring semantic similarity in short texts through greedy pairing and word semantics. In *Flairs conference*, pages 244–249, 2012.

[15] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180. 2001.

[16] D. S. McNamara and J. P. Magliano. Self-explanation and metacognition: The dynamics of reading. In *Handbook of metacognition in education*, pages 60–81. Routledge, 2009.

[17] S. Mollaoglu-Korkmaz, L. Swarup, and D. Riley. Delivering sustainable, high-performance buildings: Influence of project delivery methods on integration and project outcomes. *Journal of management in engineering*, 29(1):71–78, 2013.

[18] B. B. Morrison, L. E. Margulieux, and M. Guzdial. Subgoals, context, and worked examples in learning computing problem solving. In *Proceedings of the eleventh annual international conference on international computing education research*, pages 21–29, 2015.

[19] L. Murphy, S. Fitzgerald, R. Lister, and R. McCauley. Ability to'explain in plain english'linked to proficiency in computer-based programming. In *Proceedings of the ninth annual international conference on International computing education research*, pages 111–118, 2012.

[20] M. P. O'brien. Software comprehension–a review & research direction. *Department of Computer Science & Information Systems University of Limerick, Ireland, Technical Report*, 2003.

[21] E. S. Rezel. The effect of training subjects in self-explanation strategies on problem solving success in computer programming. 2003.

[22] V. Rus, P. Brusilovsky, L. J. Tamang, K. Akhuseyinoglu, and S. Fleming. Deepcode: An annotated set of instructional code examples to foster deep code comprehension and learning. In *International Conference on Intelligent Tutoring Systems*, pages 36–50. Springer, 2022.

[23] V. Rus, S. D'Mello, X. Hu, and A. Graesser. Recent advances in conversational intelligent tutoring systems. *AI magazine*, 34(3):42–54, 2013.

[24] L. J. Tamang, Z. Alshaikh, N. A. Khayi, P. Oli, and V. Rus. A comparative study of free self-explanations and socratic tutoring explanations for source code comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, pages 219–225, 2021.

[25] P.-H. Tan, C.-Y. Ting, and S.-W. Ling. Learning difficulties in programming courses: undergraduates' perspective and perception. In *2009 International Conference on Computer Technology and Development*, volume 1, pages 42–46. IEEE, 2009.

[26] R. K. Thornton, D. Kuhl, K. Cummings, and J. Marx. Comparing the force and motion conceptual evaluation and the force concept inventory. *Physical review special topics-Physics education research*, 5(1):010105, 2009.

[27] K. VanLehn. The behavior of tutoring systems. *International journal of artificial intelligence in education*, 16(3):227–265, 2006.

[28] K. VanLehn, A. C. Graesser, G. T. Jackson, P. Jordan, A. Olney, and C. P. Rosé. When are tutorial dialogues more effective than reading? *Cognitive science*, 31(1):3–62, 2007.

[29] A. Venables, G. Tan, and R. Lister. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*, pages 117–128, 2009.