An Explicit Strategy to Scaffold Novice Program Tracing

Benjamin Xie
University of Washington
The Information School, DUB Group
bxie@uw.edu

Greg L. Nelson
University of Washington
The Allen School, DUB Group
glnelson@uw.edu

Amy J. Ko
University of Washington
The Information School, DUB Group
ajko@uw.edu

ABSTRACT

We propose and evaluate a lightweight strategy for tracing code that can be efficiently taught to novice programmers, building off of recent findings on "sketching" when tracing. This strategy helps novices apply the syntactic and semantic knowledge they are learning by encouraging line-by-line tracing and providing an external representation of memory for them to update. To evaluate the effect of teaching this strategy, we conducted a block-randomized experiment with 24 novices enrolled in a university-level CS1 course. We spent only 5-10 minutes introducing the strategy to the experimental condition. We then asked both conditions to think-aloud as they predicted the output of short programs. Students using this strategy scored on average 15% higher than students in the control group for the tracing problems used the study (p<0.05). Qualitative analysis of think-aloud and interview data showed that tracing systematically (line-by-line and "sketching" intermediate values) led to better performance and that the strategy scaffolded and encouraged systematic tracing. Students who learned the strategy also scored on average 7% higher on the course midterm. These findings suggest that in <1 hour and without computer-based tools, we can improve CS1 students' tracing abilities by explicitly teaching a strategy.

KEYWORDS

Program tracing, instructional intervention, sketching, think-aloud.

ACM Reference format:
Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An Explicit Strat-egy to Scaffold Novice Program Tracing. In *Proceedings of The 49th ACM Technical Symposium on Computing Science Education, Baltimore, MD, USA, February 21–24, 2018 (SIGCSE '18),* 6 pages.
https://doi.org/10.1145/3159450.3159527

1 INTRODUCTION

Program tracing, the process of emulating how a computer executes a program [6], is a necessary precursor to the ability to write [12, 13, 22], debug [17], and maintain code [28]. However, there is substantial evidence that novice programmers, such as those in introductory computer science (*CS1*) courses, struggle with tracing.

Prior work suggests this is from fragile understanding of program semantics and weak strategies for tracing [11, 14, 17]. For example, novices struggle with tracing variable values as they update because they often try to remember these values [29]; this extraneous load on working memory [1, 26] increases as programs become longer and more complex. Students also struggle to produce external representations of program state, perhaps because of an incomplete understanding of the function of code elements [27] or an inability to put the "pieces" of a program together [25].

Computer-based tools to support tracing are not easily integrated into CS1 curricula, which often require students to trace code on paper. Tools enable students to see a visualization of the program state [8, 15], but they often lack active engagement, having students passively step through code execution [23]. Virtual Program Simulation could improve engagement [24], but this work has yet to be evaluated and focuses on computer-based tools. Low-level tracing without the aid of tools is necessary for expert code comprehension [28], so novices must learn to trace without tools.

As an alternative to tools, researchers have investigated *sketching*, writing visualizations of program state or other computing processes, as a pedagogical tool [3]. Previous work has investigated using sketching to assess object-oriented programming knowledge [10] and integrating sketching into pedagogy [9], with other work proposing standardization of sketched memory diagrams [4]. Cunningham et al. framed sketching as a technique for managing cognitive load and found that complete tracing sketches correlated with improved performance [3]. These findings suggest a correlation between sketching an external representation of memory and improved tracing performance, but fall short of evaluating whether teaching sketching is the cause of this improvement.

We explore these causal effects of sketching. We theorize that sketching is a form of scaffolding [21] that lessens the load on students' working memory and makes changing program state more visible. We predict that explicitly teaching a lightweight sketching strategy will offer two benefits: 1) by encouraging students to trace line-by-line "like a computer does" [15], it will reduce their temptation to create their own strategies or shortcuts which may lead to errors [18]; 2) by having students make updates to an external representation of variable values as the code executes, it will reduce the extraneous working memory load of remembering values, lowering working memory retrieval errors [1]. We hypothesize that explicitly teaching a tracing strategy will take a short amount of time and improve students' performance for tracing code on paper.

To test this hypothesis, we ask the following questions: **1) Does explicitly teaching a tracing strategy improve novice programmers' performance? 2) How do novices trace code and how does an explicit strategy change their behavior?**

2 THE STRATEGY: LINE-BY-LINE + SKETCH

The goal of our strategy was to help novices *embody the computer*. Our strategy achieved this by providing 1) step-by-step instructions

SIGCSE '18, February 21–24, 2018, Baltimore, MD, USA

ACM ISBN ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159527

on how to apply the syntactic and semantic knowledge novices have been learning to tracing questions and 2) a visual representation to explicitly track variables which we refer to as *memory tables*. The strategy instructions consisted of three steps:

(1) Read question: Understand what you are being asked to do. At the end of the problem instructions, write a check mark.
(2) Find where the program begins executing. At the start of that line, draw an arrow.
(3) Execute each line according to the rules of Java.
    (a) From the syntax, determine the rule for each part of the line.
    (b) Follow the rules.
    (c) Update memory table(s).
    (d) Find the code for the next part.
    (e) Repeat until the program terminates.

When tracing through the code, a participant creates a *memory table* with each method call, as shown in Figure 1. This memory table keeps track of parameters passed into and variables instantiated in a given method, similar to sketches from previous work [3, 4]. These were the instructions for using a memory table :

(1) Create a new Memory Table every time a method is called.
(2) Write the method name in the box at the top of table.
(3) When a variable is created, add it as a row in the table (variable name in the "name" column; value in "value" column.
(4) When a variable is updated, find the variable by name, cross out the previous value and write in the new one.
(5) After the method finishes running, write the value for the return and cross out the entire table.

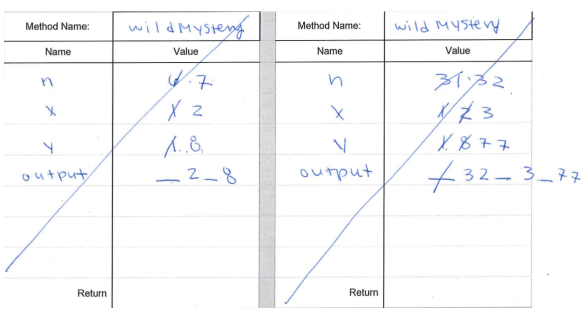

Figure 1: Two memory tables for 2 calls to the same method (Prob. 3). The participant wrote the method name at the top, variable names in the *Name* column, and variable values on the *Value* column. When variables updated, they crossed out the previous value and wrote in the new value. After each method finished executing, they crossed out the table.

In designing the strategy, our intentions were to enable instructors to teach the strategy without major changes to their pedagogy and to enable novices to easily understand and apply the strategy to any tracing problem. From our pilot testing, we found the instructions were easy to recall and discouraged participants from deviating from line-by-line tracing. We designed the memory tables so participants could easily sketch their own tables. We emphasized the separation of variable names and values to ensure variable names were differentiated from Strings and to ensure variable names from different scopes were not passed in as parameters. We emphasized using a new table for each method call to enforce scope.

3 EXPERIMENT: TRACING + THINK-ALOUD

We designed an experiment in which students enrolled in the same CS1 course worked through problems that required them to predict the output of 6 Java programs while verbalizing their thoughts. The control group used their own strategies, while the strategy group was encouraged to use the strategy from Section 2.

Our target population was novices who were just beginning to learn programming. This ensured some exposure to the syntax and semantics of a programming language, but little experience with tracing code. We advertised the study as preparation for an upcoming midterm, with participants receiving tutoring after the problem solving session. We recruited students who had taken 0 or 1 CS courses prior to the course they were enrolled in.

Ultimately, 24 students participated in the study. Eleven identified as males and 13 as females. Nineteen were enrolled in their first CS course, 1 was retaking the course, 1 had previously taken a CS1 course at local community college, 1 had taken AP Computer Science, and the 2 others specified they had previously completed an unspecified CS course. Most participants were in their first year of college (16), with 7 others in years 2-4, and 1 having a Master's degree and taking the course to change careers. Most were *not* CS majors. Only 3 were majoring in CS, with others majoring in another engineering discipline (8), Informatics (4), a non-engineering major (3), were undecided (5), or were not pursuing a major (1).

3.1 Study design: Think-aloud while tracing

The study sessions occurred 3-6 days prior to the course midterm and participants worked through 6 tracing problems that covered potential midterm concepts. Participants met individually at self-selected times with a researcher and completed a pre-survey (demographics, prior knowledge, self-efficacy [19]) prior to arriving. We attempted to block randomize participants by self-reported number of previous CS courses completed and hours spent programming or learning to program. Because of cancellations, there were 11 in the control group and 13 in the strategy group.

After introducing the study and asking a few questions (# of practice midterm problems attempted, describe tracing strategy), we spent 5-10 minutes introducing think-aloud (protocol from [5]). We then asked participants to work through 6 tracing problems in a fixed order while verbalizing their thought process. After each question, they were asked to recall what they remembered thinking. Following the completion of all problems (≈40 min on average, although they could work for as long as they wanted), we asked them to describe their strategy, how they learned it, what strategies their CS1 course taught, and had them complete a post-survey (mindset, study feedback). We then tutored them for ≈ 30 minutes, reviewing study problems, the midterm format and the cheat sheet.

For the strategy group, we spent ≈5 minutes after practicing think-aloud to teach the tracing strategy. We provided the instructions on a sheet of paper and 4 example memory tables on another sheet of paper and walked through the instructions with the participant. We provided help applying the strategy to the 1st problem, with the help typically involving us reiterating some of the written instructions. Our intention was to help the participant learn the strategy without providing hints which may unfairly support their knowledge of syntax and semantics.

#1 (created)

```
public class StrategyPractice {
    public static void main(String[]
        args) {
        int x = 1;
        int y = 2;

        int z = x;
        x = y;
        y = z;

        printSum(x, y);
        printSum(z, y);

        int val = getSum(x, y+z);
        System.out.println(val);
        System.out.println("Bye!");
    }

    public static void printSum(
        int a, int b) {
        System.out.println(a + b);
    }

    public static int getSum(int a, int b) {
        return a + b;
    }
}
```

#2 (midterm)

```
public class Farmer {
    public static void main(String[] args) {
        String farm = "here";
        String old = "macdonald";
        String macdonald = "there";
        String everywhere = "farm";
        String here = "everywhere";
        String there = "old";
        String quack = "duck";

        mystery(old, macdonald, farm);
        mystery("quack", here, "there");
        mystery(quack, "here", "farm");
        mystery(old, everywhere, there);
    }

    public static void mystery(String
        macdonald, String farm, String old) {
        String end = ".";
        if (macdonald.length() >
        farm.length()) {
            end = "!";
        }
        System.out.println(old + " " +
         macdonald + " had a " + farm + end);
    }
}
```

#3 (midterm)

```
public static void wildMystery(int n) {
    int x = 1;
    int y = 1;
    String output = "";
    while (y < n) {
        x++;
        if (x % 2 == 0) {
            n++;
        } else {
            output = n + " ";
        }
        y = 10 * y - x;
    }
    output = output + x + " " + y;
    System.out.println(output);
}
```

```
wildMystery(0);  ___
wildMystery(6);  ___
wildMystery(31); ___
```

#4 (SCS1)

```
public class Conditionals {
    public static void
    main(String[] args){
        int x = 3;
        int y = 9;
        int z = 5;
        manipulate(x, y, z);
        manipulate(y, z, z);
    }

    public static void manipulate
    (int x, int y, int z) {
        if (y % x == 0) {
            if (y >= z) {
                x = x * x;
            } else {
                z = z * z;
            }
        } else {
            if (z >= x) {
                if (y >= z) {
                    y = y * y;
                }
            } else {
                z = z * z;
            }
        }
        if (y % x == 0) {
            x = x + 3;
        } else {
            y = y * 2;
        }
        System.out.println(
            "x = " + x);
        System.out.println(
            "y = " + y);
        System.out.println(
            "z = " + z);
    }
```

#5 (SCS1)

```
public class Greeting {
    public static void main(String[] args) {
        String banner = "Good Night";
        System.out.println("banner");

        greet("Alice", "night");
        System.out.println(banner);

        greet("Bob", "night");
        System.out.println(banner);
    }

    public static String greet(String name, String
    time) {
        String greeting = "Good " + time + " " +
        name;
        if (name.equals("Alice")) {
            return greeting;
        }
        greeting = greeting + "!";
        System.out.println(greeting);
        return greeting;
```

#6 (created)

```
public class OddMystery {
    public static void main(String[] args) {
        int x = 2;
        int y = 3;

        System.out.println(x + y + "!");

        compute(y, x);

        double val = compute(x, y + 1);

        System.out.println(val);
    }

    public static double compute(int
    x, int y) {
        int z = y;
        y = x;
        x = z;

        System.out.println("x" + y + z);

        return Math.pow(x, y);
    }

}
```

Figure 2: The 6 problems for the study, each asking for the program's output. Bold denotes changes to adapted items.

3.2 Problems: Fixed-code tracing questions

The study had 6 *fixed-code* problems which covered early CS1 concepts and required participants to trace provided code and determine the output. In contrast, the midterm assessed tracing (≈50% of midterm), code construction (≈40%), and invariants (≈10%).

Figure 2 shows the 6 problems. Problem 1 helped participants practice using memory tables with variable updates, repeated method calls with parameters passed in, and calls to different methods. Problems 2 and 3 were from practice midterms and Problems 4 and 5 from the SCS1 [16]. Problem 6 assessed midterm concepts not covered in other problems. We used notes from the think-aloud sessions to develop a scoring rubric for the problems completed during the study. The rubric attempted to differentiate scores based on what participants found difficult about each problem. We did not look at participant responses until after the rubric was created.

After the midterm, we sent an online survey to participants to solicit their midterm grades and experiences (perceived problem difficulty, how they prepared for the midterm, how the study may have helped). We also asked the participants in the strategy group to describe the strategy we taught them (to ensure they remembered it) and whether they used the strategy. Of the 24 participants, 17 responded (8 control, 9 strategy, with 1 not sharing their grade).

4 RESULTS

Our data included background information, participants' sketches and responses, researcher notes, audio recordings of participants thinking aloud, and self-reported midterm grades.

4.1 The explicit strategy improved correctness

We answer our first research question by comparing the performance of the two groups on the study problems and midterm. We found a strong linear correlation ($r = 0.91$, Pearson) between the scores on the study problems and the midterm grades, suggesting both measure similar concepts. Because the distributions of scores for each problem in the study deviated from normality ($p < 0.05$, Shapiro-Wilk [20]), we conducted a one-tail non-parametric t-test (Mann-Whitney U test [20]) to determine significance without making assumptions on the distribution shape. Participants' total scores and midterm scores did *not* deviate from normality, so we used a one-tail parametric t-test (Welch t-test [20]) to compare total scores and midterm scores between conditions. We calculated a *Common Language* (CL) effect size from Cohen's d [7, 20] for the parametric test and by dividing the test statistic U by the product of the sample sizes for the non-parametric test [7].

Although we designed the 6 problems to be harder than midterm problems, a large portion of students got many problems correct. Because of this skew, we use the median as the average and the interquartile range (IQR, 3rd quartile/75% - 1st quartile/25%) to measure dispersion for scores on individual problems. For all problems, the median for the strategy group was greater than or equal to the control group's and the IQR was lesser, as shown in Figure 3. The strategy group tended to perform better and with less variability.

Strategy group participants performed significantly better on Problem 4 (p=0.021), which required them to keep track of multiple variables through nested if/else statements. Figure 3 shows that 11 of 13 participants in the strategy group scored perfectly on this problem and above the median score of the control. The U statistic is 101.5, which we interpret as a CL effect size to say that there is a 71% chance that a randomly selected participant from the strategy group scores better than one from the control group on Problem 4.

For the total score across all problems, we found that the strategy group performed on average 15% better and with 46% less variability. We could model the distributions of total problem scores as

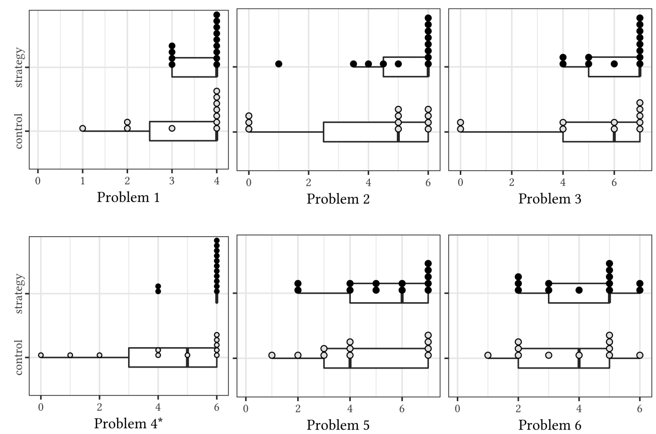

Figure 3: Scores for the 6 study problems by condition. Each dot denotes a participant score. Each boxplot shows the interquartile range. * denotes $p < 0.05$.

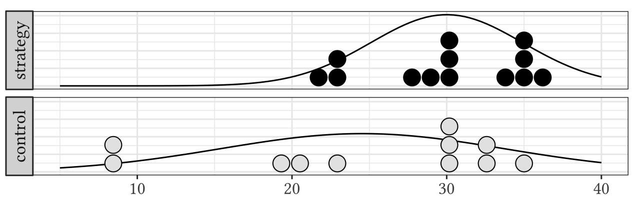

Figure 4: Total scores for study problems by condition (max 36). A dot is a participant score and a curve is a fitted normal curve. ($\bar{x}_s$=30.00, σ_s=4.91). ($\bar{x}_c$=24.55, σ_c=9.16). p=0.049.

approximately normal, so we used mean and the standard deviation to measure dispersion. Figure 4 shows the distributions, where the strategy group performed significantly better ($p = 0.049$). Ten of the 13 students who learned the strategy performed better than the mean of the control group. We interpret the Cohen's d of 0.7243 as a CL effect size and say that there is a 70% chance that a randomly selected participant from the strategy group scores better than one from the control group on the study problems.

Seven of the 8 strategy group participants who provided midterm grades said they used the strategy on the midterm. These 8 scored on average 7% better than the control group and had 42% less variability. Figure 5 shows the distributions of midterm scores and fitted normal curves. The class average of 80.4 was within 0.5 standard deviations of the control group mean of 84.5 (Z=-0.36, σ_c=9.16) but was over 1.5 standard deviations below the strategy group mean of 91.5 (Z=-1.66, σ_s=4.91). We expected both groups to perform above the class average because of the tutoring. We found a trend towards significance (p=0.083) that the strategy group performed better than the control group on the midterm. The tracing questions on the midterm were easier, so the difference in performance came largely from the code writing questions which were more challenging.

Considering these performance scores together, our data show that explicitly teaching a strategy can improve tracing performance, even on a midterm which assesses more than just tracing ability.

4.2 An explicit strategy reduced errors

In this section, we qualitatively investigate the reasons for the performance increases in the previous section. To perform this analysis,

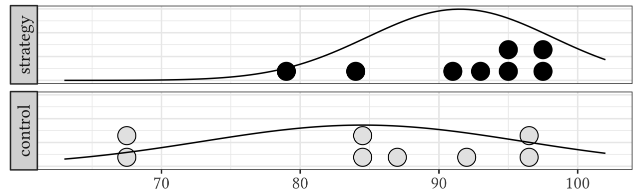

Figure 5: Midterm scores by conditions (max 100). A dot is a participant's score and a curve is a fitted normal curve. ($\bar{x}_s$=91.5, σ_s=6.68). ($\bar{x}_c$=84.5, σ_c=11.5). p=0.083.

we triangulated across think-aloud recordings, sketches, solutions, and researcher notes to characterize the tracing strategy that each participant used. We compare and contrast the strategies that we observed through this process by describing high and low performers in each condition, describing similarities within each subgroup and go into detail on one member of each group (called C_{max}, C_{min}, S_{max}, S_{min} for control/strategy participant with max/min score).

4.2.1 Control group's low performers. The two lowest performing members of the control group demonstrated incomplete knowledge of semantics and often deviated from line-by-line tracing. They deviated because they skipped past an unfamiliar concept (e.g. `x++`) or they made incorrect assumptions about the control flow (e.g. choosing to skip a method call in Problem 4 because they felt the method would not output anything). They often focused on trying to determine if a problem was similar to one they had previously practiced. Both participants quit partway through several problems, either because of a lack of semantic knowledge overwhelming them or inadequate strategy leaving them unsure what to do next.

C_{min}: One of the lowest scoring members of the control group scored 9/36. He had attempted 5 practice midterm questions prior to the study. His strategy involved trying to comprehend the code's overall purpose and translating the code to English. This strategy and an incomplete understanding of Java semantics overwhelmed him, resulting in him giving up and not answering half of the problems. He tried to look at the code holistically before tracing, but either found the code "really weird" for unfamiliar problems or recognized the problem as similar to one he had previously seen, but failed to recall how to solve it ("I swear I've done a problem like this. I just forgot how to do it."). He had trouble applying concepts he learned in class. He often focused on translating code into natural language but was usually unable to do this for the entire problem. In Problems 4 and 6, he considered writing down intermediate values ("Maybe if I write down the variables?"), but did not do it. He also inappropriately drew upon knowledge from other domains, such as seeing a number followed by an exclamation point in a string ("5!", Prob. 6) and thinking it referred to a factorial. Longer code segments (Prob. 4) were more overwhelming, with the participant "thinking that this is crazy, I don't know where to start."

4.2.2 Control group's high performers. The six highest performers in the control group scored 30-35 of 36. They were able to trace code line-by-line and also update a sketch of intermediate values, although they did not consistently create sketches. Some looked at code holistically before tracing line-by-line, often looking for print/output statements and return statements, but did not do this for longer code segments (Prob. 4) or for later, more complex problems (Prob. 5, 6). Others immediately began tracing line-by-line.

High performing participants in the control group tended to intermittently sketch a tabular external representation to keep track of intermediate values. They often began sketching as a response to specific "cues," such as when the code instantiated multiple variables or when a variable began updating ("I needed to make a table to keep track of the variables because there are a lot of moving parts"). Their tracing of intermediate values also appeared to vary by data type, often sketching numeric variables in tables, but ignoring String variables. This may be because Strings take longer to write, something noted by many members of the strategy group.

C_{max}: The highest scoring participant from the control group scored 35/36 and used both inline (next to code) and tabular (separate from code) annotations to track intermediate values. She completed 4 practice midterm questions prior to the study and reported the highest programming self-efficacy [19] of all participants. She self-described her tracing strategy as "writing out the values that were given and keeping track of the series of manipulations as they went through." Like most of the control group, she recognized Problem 2 and applied a problem-specific strategy learned in class. For other problems, she immediately began tracing the code line-by-line. She wrote intermediate values inline when they were numeric and did not update. She created a separate table when values were Strings or when they updated. The only error she made was when she used the + operator incorrectly (Prob. 6).

4.2.3 Strategy group's low performers. The 3 lowest strategy group participants did not use the strategy appropriately, scoring 22-23.5 of 36. Despite being taught the strategy, they still deviated from line-by-line tracing, sometimes unintentionally (not checking an if/else conditional because they believe it is always true, Prob. 3), and sometimes intentionally (see S_{min}). One participant did not create memory tables for the last 3 problems she did. These participants found sketching the memory tables time-consuming and often unnecessary. In sharp contrast to the control group, these low performers never became overwhelmed or gave up.

S_{min}: The lowest scoring participant in the strategy group scored 22/36, still within 3 points of the mean score of the control group. He reported having done 0 practice problems prior to the study. Prior to being taught the strategy, he tried to interpret what the code was doing and translate it to pseudocode, while also trying to remember variable values. When applying the strategy, he still deviated from line-by-line tracing intentionally, such as in Problem 5 when he correctly traced through the first call to a method and realized it resulted in no printed output or stored return, then skipped the second call to the method which had different parameters passed in and would have printed an output. He justified his decision by thinking that "since [the method] doesn't do anything with the return value, I just ignore it. I don't need to write another memory table. It doesn't affect the output in any way. At least, I don't think it does." He also unintentionally deviated from line-by-line tracing, such as in Problem 4 when he correctly interpreted an if condition as false, but skipped past the else statement. S_{min} found the strategy familiar and reliable, "having a very structured set of rules to follow that would generally work for any code".

4.2.4 Strategy group's high performers. The four highest performing participants scored 34-36 of 36, using the strategy as intended to complement their Java semantics knowledge. Participants had strong understanding of Java, even on more advanced concepts which often confused most other participants (e.g. scope, return). They used the strategy as intended, tracing line-by-line and creating and updating memory tables as they traced. One participant felt the strategy "forces you to think through what you're doing...forces you not to skip around." Another noted that he "didn't have to keep thinking about values because they were on paper," suggesting that the strategy does offload values from working memory. Some participants actively kept track of where they were in the code either by pointing at the line they were executing or by crossing out previously executed lines. The most common error among this group was returning an integer instead of a double in Problem 6.

S_{max}: The highest scoring participant from the strategy group was the only one to score perfectly on all 6 problems. S_{max} self-reported having attention deficit hyperactivity disorder (ADHD) and completing around 100 practice problems prior to the study, which is much greater than the average participant who had completed less than 10 practice problems. She was also the only one who was not an undergraduate, already having a Master's degree in an unrelated field. She immediately began tracing the code line-by-line while also filling in the memory tables completely. She also wrote check marks after lines in the main method were executed, perhaps misinterpreting the 2nd step of the strategy. While other participants in both conditions deviated from line-by-line tracing, she never deviated from line-by-line tracing. This enabled her to avoid strategic errors other participants made when skipping line by line tracing. S_{max} found the strategy useful "for keeping track of what a value is," especially when variables were updating or when there were multiple methods (multiple scopes). She recognized that the strategy required time and paper to write down the memory tables, but she felt "like you have to write this stuff down" to solve the problem. With this method, S_{max} felt "you can focus on the calculations without wondering what [a variable] equals."

4.3 Limitations

Because we conducted this study days before course midterms, we were mindful of participants' time constraints. We relied on proxy measures of prior knowledge (est. hours programming, # of CS courses). Our performance measurements are not validated, although we found a strong correlation between the study and midterm scores. Students self-selected to participate in the study, so selection bias may exist. Some students dropped out of the study, resulting in our block-randomization on prior experience slightly favoring the strategy group. Participants took the midterm only 3-6 days after the study, so longer term retention is unclear. Think-aloud and recall may have influenced their thinking, although we followed best practices on protocol analysis [5]. We observed some carryover effect between problems even though we did not provide feedback. Our statistical analysis assumes applying the strategy has the same effect for all participants. A mixed model may more appropriately represent the effect and generalize better [20].

5 DISCUSSION

Our results show that explicitly teaching a tracing strategy that emphasizes line-by-line tracing and an external representation for tracking state may improve tracing performance. We also found that novices who performed well tended to be more able to apply knowledge of semantics, trace line-by-line systematically, and write down intermediate values rather than try to remember them. This was true regardless of whether a student used our strategy, but when a student did, they tended to perform even better. These findings support prior work that identified poor problem solving as a cause of novice programmers' poor tracing [14] and found correlations between sketching and tracing performance [3, 10].

There are many ways to interpret our results. First, confounds such as differences in prior knowledge, self-efficacy, motivation, fatigue, and amount studied could have influenced participants' performance. We did not find significant differences between groups relating to fatigue before the study (# hrs slept) or the amount of practice prior to the study or midterm (# of practice problems attempted). There was a significant difference in average programming self-efficacy [19] favoring the control group (p<0.01, 2-sided Welch t-test). Our sample size was also small and limited to students from a single course, so course instruction (instructor demonstrated tracing with tables, students used jGRASP debugger [2]) could have influenced how participants traced. Replication is necessary to increase confidence in the effects of teaching a strategy.

One interpretation is that the strategy helped lower performers make progress and not give up. Those in the control group (e.g. C_{min}) tended to become overwhelmed and gave up on problems in part because their strategies (e.g. translating code to English, looking at code holistically) were inappropriate for their current abilities. In contrast, those in the strategy group (e.g. S_{min}) used the strategy to make incremental progress, never becoming overwhelmed.

Another interpretation is that strategy did indeed help, but that it did not compensate for a lack of syntactic or semantic knowledge. This was reinforced by our qualitative results, which showed that lower performing participants in both groups tended to make mistakes reflecting incomplete semantic knowledge.

The implications for teaching are simple: help students practice an explicit strategy. That said, our strategy was not necessarily optimal. Our participants had to do additional work to keep track of their place in the code (pointing, crossing lines out). We found the memory tables to be effective for primitive data types and Strings, but this may not hold true for data structures (e.g. arrays) or objects. Integrating this strategy with more advanced representations [4] and determining long-term benefits (e.g. transfer to code writing ability) may be promising future work.

6 ACKNOWLEDGEMENTS

We have archived the study materials at https://github.com/bxie/archive/tree/master/sigcse2018. This material is based upon work supported by the National Science Foundation under Grants No. 1314399, 1735123, 1703304 and the NSF GRFP under Grant No. 12566082. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Alan D. Baddeley and Graham Hitch. 1974. Working Memory. *Psychology of Learning and Motivation* 8 (Jan 1974), 47–89.

[2] J. H. Cross, D. Hendrix, and D. A. Umphress. 2004. JGRASP: an integrated development environment with visualizations for teaching java in CS1, CS2, and beyond. In *34th Annual Frontiers in Education, 2004. FIE 2004.* 1466–1467.

[3] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using Tracing and Sketching to Solve Programming Problems: Replicating and Extending an Analysis of What Students Draw. In *2017 ACM Int'l Computing Education Research Conf. (ICER '17).* ACM, 164–172.

[4] Toby Dragon and Paul E. Dickson. 2016. Memory Diagrams: A Consistant Approach Across Concepts and Languages. In *47th ACM Technical Symp. on Computing Science Education (SIGCSE '16).* ACM, 546–551.

[5] Karl Anders Ericsson and Herbert Alexander Simon. 1993. *Protocol Analysis: Verbal Reports as Data Revised Edition.* The MIT Press.

[6] Sue Fitzgerald, Beth Simon, and Lynda Thomas. 2005. Strategies That Students Use to Trace Code: An Analysis Based in Grounded Theory. In *First Int'l Workshop on Computing Education Research (ICER 2005).* ACM, 69–80.

[7] Robert J. Grissom and John J. Kim. 2012. *Effect Sizes for Research: Univariate and Multivariate Applications, Second Edition.* Routledge.

[8] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Technical Symp. on Computer Science Education (SIGCSE '13).* ACM, 579–584.

[9] Matthew Hertz and Maria Jump. 2013. Trace-based Teaching in Early Programming Courses. In *Technical Symp. on Computer Science Education (SIGCSE '13).* ACM, 561–566.

[10] Mark A. Holliday and David Luginbuhl. 2004. CS1 Assessment Using Memory Diagrams. In *Technical Symp. on Computer Science Education (SIGCSE '04).* ACM, 200–204.

[11] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, and et al. 2004. A Multi-national Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04).* ACM, 119–150.

[12] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introductory Programming. In *14th Annual ACM SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITiCSE '09).* ACM, 161–165.

[13] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships Between Reading, Tracing and Writing Skills in Introductory Programming. In *4th Int'l Computing Education Research Workshop (ICER '08).* ACM, 101–112.

[14] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-national, Multi-institutional Study of Assessment of Programming Skills of First-year CS Students. *SIGCSE Bull.* 33, 4 (Dec 2001), 125–180.

[15] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *2017 ACM Int'l Computing Education Research Conf. (ICER '17).* ACM, 2–11.

[16] Miranda C. Parker, Mark Guzdial, and Shelly Engleman. 2016. Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment. In *2016 ACM Int'l Computing Education Research Conf. (ICER '16).* ACM, 93–101.

[17] David Perkins and Fay Martin. 1985. *Fragile Knowledge and Neglected Strategies in Novice Programmers.*

[18] D. N. Perkins, Chris Hancock, Renee Hobbs, Fay Martin, and Rebecca Simmons. 1986. Conditions of Learning in Novice Programmers. *J. of Educational Computing Research* 2, 1 (Feb 1986), 37–55.

[19] Vennila Ramalingam and Susan Wiedenbeck. 1998. Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Group Analyses of Novice Programmer Self-Efficacy. *J. of Educational Computing Research* 19, 4 (Dec 1998), 367–381.

[20] J. Robertson and M. Kaptein (Eds.). 2016. *Modern Statistical Methods for HCI.* Springer.

[21] B. Rosenshine and C. Meister. 1992. The use of scaffolds for teaching higher-level cognitive strategies. *Educational Leadership* 49, 7 (Apr 1992), 26.

[22] Elliot Soloway. 1986. Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM* 29, 9 (Sep 1986), 850–858.

[23] Juha Sorva, Ville Karavirta, and Lauri Malmi. 2013. A Review of Generic Program Visualization Systems for Introductory Programming Education. *Trans. Comput. Educ.* 13, 4 (Nov 2013), 15:1–15:64.

[24] Juha Sorva, Jan Lönnberg, and Lauri Malmi. 2013. Students' ways of experiencing visual program simulation. *Computer Science Education* 23, 3 (Sep 2013), 207–238.

[25] James C. Spohrer and Elliot Soloway. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Commun. ACM* 29, 7 (Jul 1986), 624–632.

[26] John Sweller. 1994. Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction* 4, 4 (Jan 1994), 295–312.

[27] Lynda Thomas, Mark Ratcliffe, and Benjy Thomasson. 2004. Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results. In *Technical Symp. on Computer Science Education (SIGCSE '04).* ACM, 250–254.

[28] Rebecca Tiarks. 2011. What maintenance programmers really do: An observational study. In *Workshop on Software Reengineering.* 36–37.

[29] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in Novice Programmers' Poor Tracing Skills. In *12th Annual SIGCSE Conf. on Innovation and Technology in Computer Science Education (ITiCSE '07).* ACM, 236–240.