Using Tracing and Sketching to Solve Programming Problems

Replicating and Extending an Analysis of What Students Draw

Kathryn Cunningham
School of Interactive Computing
Georgia Institute of Technology
85 5th St NW
Atlanta, Georgia 30332
kcunningham@gatech.edu

Sarah Blanchard
School of Psychology
Georgia Institute of Technology
654 Cherry Street
Atlanta, Georgia 30332
sblanchard6@gatech.edu

Barbara Ericson, Mark Guzdial
School of Interactive Computing
Georgia Institute of Technology
85 5th St NW
Atlanta, Georgia 30332
{ericson,guzdial}@cc.gatech.edu

ABSTRACT

Sketching out a code trace is a cognitive assistance for programmers, student and professional. Previous research (Lister *et al.* 2004) showed that students who sketch a trace on paper had greater success on code 'reading' problems involving loops, arrays, and conditionals. We replicated this finding, and developed further categories of student sketching strategies. Our results support previous findings that students who don't sketch on code reading problems have a lower success rate than students who do sketch. We found that students who sketch incomplete traces also have a low success rate, similar to students who don't sketch at all. We categorized sketching strategies on new problem types (code writing, code ordering, and code fixing) and find that different types of sketching are used on these problems, not always with increased success. We ground our results in a theory of sketching as a method for distributing cognition and as a demonstration of the process of the notional machine.

CCS CONCEPTS

•**Social and professional topics → Computing education;** *CS1;*

KEYWORDS

CS1, novice programmers, tracing, sketching, notional machine, distributed cognition

1 INTRODUCTION

Introductory computing courses aim to teach students computing fundamentals through instruction on basic programming skills, syntax, and semantics [1]. Studies suggest these courses often fail to teach students how to successfully write code to solve basic problems [17, 27]. Many students reach the end of CS1 without being able to read and understand short pieces of code [14, 24].

Why do students struggle? Research suggests that the intrinsic cognitive load [23] of solving typical introductory programming problems may be higher than that of introductory problems in other fields [18]. Also, students exhibit a wide range of misunderstandings about a crucial concept: the *notional machine* [22]. The notional machine is a theoretical construct representing the process through which a computer executes code of a particular language or paradigm [22]. An accurate understanding of the notional machine is central to a student's ability to trace and write code, since they must run code through their mental model of the notional machine in order to predict the outcome of that code's execution.

The understanding of the notional machine and the management of cognitive load are hidden mental processes. However, when students sketch out a problem-solving technique such as a code trace on paper, we can observe some signs of their cognition. Prior work in computing education research has used student drawings to identify misconceptions about variable assignment [15]. In educational psychology student sketches are also used to gauge understanding of concepts [3].

In this paper, we use the term ***sketching*** to describe a programmer's written visualizations of program state or any other computing process. Sketching may be most frequently associated with *code reading*, when students are likely to write down a trace of variable states. However, we purposefully keep the definition of sketching broad to cover many situations in which a student or professional programmer may take pencil to paper. Such tasks may include planning an algorithm, designing an object-oriented hierarchy, or annotating existing code.

Sketching may help students manage cognitive load while also demonstrating the student's understanding of the notional machine to themselves, peers, and instructors. Sketching requires that the student take an active role as they describe a computational process with their pen. Sketching the process of code execution, as during a trace, forces a student to ask themselves "what happens next?" as they draw each step of the notional machine.

This study investigates the following questions about sketching by CS1 students:

- *Are some sketching techniques more associated with correct problem-solving than others?*
- *Does sketching differ for different problem types, like reading, writing, fixing, and ordering code?*
- *What may influence students' sketching choices?*

ICER '17, August 18–20, 2017, Tacoma, WA, USA
© 2017 ACM. 978-1-4503-4968-0/17/08…$15.00
DOI: http://dx.doi.org/10.1145/3105726.3106190

2 BACKGROUND

2.1 Sketching in computing education research

The Leeds Working Group (LWG) at ITiCSE 2004 produced an influential multi-institutional, multi-national study analyzing the code reading skills of hundreds of students [14]. While outcomes were not as stark as those of the McCracken Working Group analysis at ITiCSE 2001 which evaluated code writing skills [17], the LWG found that CS1 students were challenged by many code reading problems involving loops, conditionals, and arrays. Their assessment involved two question types: code prediction problems where students determined the values of variables after code execution, and code completion problems where students chose pieces of code needed to complete a code snippet with certain functionality.

For a subset of participants, LWG researchers characterized and analyzed all notes and marks drawn on the test sheets. Certain sketches seemed more helpful than others. Tracing sketches that tracked multiple values of the same variable were most correlated with correct responses. When students left their paper blank, their success rate was lower than when any sketching type was employed. These findings suggest that sketching may support students' ability to solve code prediction and code completion problems.

Some members of the LWG took this analysis further [16]. Sketching amounts varied widely from institution to institution, from as as low as 28% of problems sketched to as high as 92%. Students consistently sketched more often on code prediction problems than on code completion problems.

Whalley *et al.* [29] analyzed student sketching on questions from the LWG analysis and on additional code reading question types. Like the LWG, they found that sketching is associated with more successful problem-solving. Different sketch types were more common on different problem types. The authors also noted that a minority of participants used the tracing sketch type demonstrated in their classroom. Lister et al. (2010) [13] also replicated the LWG's results about the success of tracing sketches and expressed a need for more fine-grained categories of sketched traces.

Researchers have reported that CS1 students' ability to sketch variable states correlates with their scores on other assessments. Such diagrams include "memory diagrams" [9], where differing shapes distinguish objects (circles), primitives and references (squares), and classes (diamonds). Similar results were observed with "object diagrams" [25], although Thomas *et al.* found that students were not more likely to use diagrams after they were introduced in class, and that using a diagram did not correlate with increased success for students with prior programming experience. Hertz and Jump used "trace-based-teaching" [8], a curricular method involving intricate sketched program traces, including details about variables' location in the stack or in the heap and variables' enclosing method call. Each class started with 20-30 minutes of tracing activities using these diagrams. They found that CS2 grades increased and dropout rates decreased after introduction of this technique.

2.2 Tracing and the notional machine

The idea of the notional machine was first proposed in 1986 [5], yet recent work [22] has made the notional machine a key framework for viewing how students understand computing. The notional machine is an abstraction of how the computer processes code, based in a particular language or paradigm. When a student writes code with an expected outcome or predicts the result of running a piece of code, the student runs that code through their mental model of the notional machine [22]. The understanding of the notional machine forms a hidden but crucial intermediate step in programming work.

When teachers understand the notional machine, it becomes clear what mental processes a student must learn. Effective classroom techniques assist students in defining and refining their mental models, and successful assessments allow teachers to gauge their students' depth of understanding of the notional machine.

Tracing code, which is stepping through the process of the notional machine, makes a student's mental model of the notional machine evident [22]. A teacher reviewing a sketch of a student's trace might be able to identify misconceptions and focus on improving the student's understanding. By creating a sketch of a code trace, students may be able to clarify and refine their mental model of the notional machine.

2.3 Sketching as distributed cognition

Intelligence is commonly considered a characteristic of an individual, and cognition is commonly considered a process of a single mind. However, from the perspective of *distributed cognition*, cognitive processes occur in a socio-technical system composed of humans, artifacts, and their interactions [10]. The sketch and the sketcher can be viewed as part of a distributed system that is performing cognition about programming. Together, the markings on the paper, the process of modifying the marks, and the cognition of the sketcher work to generate an answer.

Cognition can be off-loaded from a single mind with memory aides like diagrams and text, technological tools like measurement devices and data displays, and other individuals like teachers and peers [20]. Many introductory-level fields make effective use of drawing and sketching to help students work "smarter". Examples include structured long division calculations in elementary mathematics [12], the pushing electron formalism in organic chemistry [6], and force body diagrams in introductory physics [4].

A key insight from work in distributed cognition is that the design of visual aids can modify the amount and type of cognition a human must perform when interacting with the aid [10]. With the limited working memory that a student must use to run their mental model of the notional machine [26], effective off-loading is crucial. Some sketching methods may successfully offload more cognition from the human to the paper. Another sense of a sketch's effectiveness is greater correctness. Sketches may not always be highly successful at both. Single value tracing is a phenomenon noted by Vainio and Sajaniemi [28], where students only keep track of one "memory slot" that holds the most recently modified value. This trace has low cognitive load, but with its inaccurate representation of the notional machine, it produces incorrect results for all but the simplest problems.

3 EXPERIMENTAL DESIGN

CS1 students from a large research university in North America participated in a computer-mediated experiment, using their own laptops in a supervised setting. The experiment took place during the 10th week of a 16 week semester. The analysis presented here is from the pre-test portion of this experiment.

The test consisted of eight questions written in Python. It tested CS1 knowledge on lists, loops, and conditionals. As shown in Table

Table 1: Question types and grading strategies.

#	Type	Graded by
1	Reading - Prediction	Multiple choice
2	Reading - What code does	Multiple choice
3	Reading - Prediction	Multiple choice
4	Reading - Prediction	Multiple choice
5	Reading - Prediction	Multiple choice
6	Fixing	Rubric (12 point scale)
7	Ordering - 2D Parsons	Rubric (12 point scale)
8	Writing	Rubric (12 point scale)

What do a and b equal after the following code executes?

```
a = 10
b = 3
t = 0
for i in range(1,4):
    t = a
    a = i + b
    b = t - i
```

Figure 1: Wording of Problem 4

1, the test consisted of five multiple choice questions about code reading, one question about fixing code to meet a specification, one question involving ordering code in a two-dimensional Parsons problem with paired distractors [11, 19], and one question involving writing code to meet a specification. The code reading questions were of two types: four involved predicting the result of executing a code snippet (see Figure 1 for an example), while one involved determining what code does. The problems involving fixing, ordering, and writing code all contained sample input and output in the problem descriptions. The fix code and write code problems provided feedback to test-takers in the form of unit tests. Students were allowed a maximum of 45 minutes to complete the eight-question test: 15 minutes for the reading questions, 10 minutes for the fixing question, 10 minutes for the ordering question, and 10 minutes for the writing question.

During the test, participants were instructed to use provided pens and blank scratch paper (labeled with their unique identification number) if they wished to draw. Participants were instructed to return their scratch paper to the experiment administrators after completion of the test.

In this study, we examine scratch sheet and performance data. 159 participants attended the experiment. 24 participants were eliminated from the data set because they did not answer at least one of the questions or spent less than 30 seconds on trying to answer one of the fix code, order code, or write code problems.

4 REPLICATION OF THE LEEDS WORKING GROUP ANALYSIS

Our data provides the opportunity to replicate the Leeds Working Group (LWG) analysis on the use of sketching by CS1 students on code reading problems about loops, lists, and conditionals. We present a replication of the key figure about sketching from the LWG study in Table 2. For this analysis, we used a random subset of the data: scratch sheets with even-numbered IDs (N=65).

4.0.1 *Differences between studies.*
Minor procedural differences between the LWG study and the current study should be noted. In our computer-mediated experiment, students read questions on their laptop screen, sketched on a blank sheet of paper, and selected an answer by clicking a button on the screen. In the LWG study, participants sketched on paper that contained printed test questions, and chose their answer by circling a printed answer choice. While the LWG's questions were written in Java, this study used questions written in Python. Our participants were CS1 students a little more than halfway through their course, while Leeds Working Group students had recently completed or were near completion of CS1.

4.0.2 *Sketch types.*
The LWG identified twelve types of sketching in their analysis. The following eight sketch types are used in this replication:

- *Blank Page (B)*: Nothing written at all.
- *Computation (C)*: An expression containing an operation such as addition or division. The operands may be two literal values or a value and a variable.
- *Keeping Tally (K)*: Tally marks keeping track of the number of occurrences of something.
- *Number (N)*: A standalone value, or a variable and an associated single value.
- *Position (P)*: Indices are written on top of an array, assisting with lookup by index.
- *Trace (T)*: Multiple values of one or more variables are tracked by listing values near variable names. Previous values may be crossed out.
- *Synchronized Trace (S)*: Values of multiple variables are tracked, with the value of all variables re-written any time any variables changes.
- *Odd Trace (O)*: Multiple values of one or more variables are tracked, but in a way that is not T or S.

The computer-mediated nature of our experiment meant that some of the LWG sketch categorizations are not applicable because they require writing on or near pre-existing problem text. The categories *Alternate Answer (A)*: a different answer choice circled, *Underlined (U)*: part of the original problem text underlined, and *Ruled Out (X)*: one or more of the answer choices crossed out were not considered in this analysis. We also removed the *Extraneous Marks (E)* category, and instead did not record sketching that did not clearly fall into one of the other categories

We noticed some additional sketch types in our data that we created new categories for:

- *Describe (D)*: English words describing expected functionality of code.
- *Loop Variables (L)*: A listing of the loop variables returned by the `range` function[1] are written out. This provides a reminder of the value for each loop iteration.
- *Rewrite (R)*: Portions of code from the question are rewritten. Question information is then more easily accessible for tracing or other sketching.

[1]Used in Python `for` loops to generate index variable values

Table 2: Percentage of correct answers on code reading problems when students (N=65) use a particular LWG sketch type

	Current Study		**LWG Study**	
Sketch Category	% Correct	n	% Correct	n
Trace (T)	82.1	95	75	215
Computation (C)	78.7	61	60	30
Loop Variables (L)	77.3	44	-	-
Position (P)	66.7	9	64	75
Number (N)	65.3	118	70	189
Rewrite (R)	60.7	56	-	-
Blank (B)	60.7	135	50	256
Odd Trace (O)	55.6	9	78	23
Synchronized Trace (S)	-	0	77	73
Keep Tally (K)	-	0	100	6
Alternate answer (A)	-	-	69	26
X-ruled Out (X)	-	-	60	60
Underlined (U)	-	-	52	44

4.0.3 Coding the data. We created a coding system for each of the eleven categories and the first two authors coded the data. Inter-rater reliability on these sketching categories an 81% match across 20% of the data.

In the LWG study, sketching was performed on or near question text, making determination of which sketch belonged to which question straightforward. In our data, all problems were sketched on a single scratch sheet, requiring additional analysis to pair sketches with questions.

Noting the distinct variable names used in the different problems, we developed a coding scheme to determine which sketches belonged to which problems. The first two authors implemented this coding scheme to match sketches with questions, while leaving out sketches whose originating question was unclear. Inter-rater reliability on how each line of sketching was categorized was 95% across 20% of the data.

4.1 Not all sketching is created equal

We replicate the first major result of the LWG: different sketching types are associated with different rates of success. Students' average score across all code reading problems was 66.8%. However, the use of certain sketches like Trace and Computation was associated with higher than average success, while the use of Rewrite or Odd Trace was associated with lower than average success. Some methods of offloading cognition appear to be more helpful than others.

4.2 The sketch type "Trace" is the most successful

The LWG study identified three different types of tracing sketches, and found that they were the most highly associated with success out of all sketch types. In our data, one of those trace categories was the most successful category of all sketching categories (82.1% success rate), while the other two tracing categories were rarely used. These results support the idea that tracing is a strong strategy for solving code reading problems.

4.3 No sketching at all is associated with lower success

While students who left problems blank did not score as poorly in our data as in the LWG study (61% vs 50%), they still scored poorly compared to other sketch categories. In the LWG study, leaving a problem blank was associated with the lowest average score of any sketch type, closely followed by Underlined. In our analysis it has the second lowest average score overall, tied with Rewrite. It should be noted that the category with the lowest score, Odd trace, has few data points. Choosing not to sketch is not a successful strategy for solving code reading problems.

4.4 Differences from LWG results

4.4.1 Odd Traces are not successful. In the LWG, the Odd Trace category was one of the sketch categories with the highest rate of success (78%). However, in our analysis, the technique had the lowest success rate (56%). In both studies, Odd Traces were rare, occurring in only 3% of all problems. Synchronized Traces were not found at all in our data.

It is possible that we interpreted the Odd Trace and Synchronized Trace categories differently than the LWG. It is also possible that our participants simply did not use those trace types, perhaps because they had never seen them. All our students attended a single institution, while participants in the LWG study attended a variety of institutions, and potentially saw a wider variety of sketching techniques.

In our data, Odd Traces tended to be less structured than Traces. It also was more difficult to determine which values were associated with which variable. Perhaps this lack of clarity was associated with more difficulty in tracking variable values.

4.4.2 Differences in non-tracing sketch types. While the most successful and least successful sketching techniques were consistent between our results and those of the LWG, other sketch types were observed at different rates and were associated with varying success.

This difference may be explained by differences in the problem types. In our data we did not observe certain sketch types, including Keep Tally. Keep Tally was rare in the LWG analysis; it may have been even less common in this data set because a counter variable only appeared in one problem in our test, while the LWG questions contained four problems with counters.

The difference could also be due to the difference in test-taking format. For example, the Position type was much more rare in our data than in that of the LWG. It is possible that creating this sketch required extra effort to copy down an array, rather than writing near a pre-printed array. The additional burden of this task may have made it less appealing. Alternatively, the questions in our experiment may have requires less complex patterns of indexing into arrays than those of the LWG, making it less likely that students sketch to assist with this task.

Table 3: Probability of getting problems right, with and without sketching.

Question	% Correct	% Who Sketched
1 - Predict	63.0	70.4
2 - What code does	78.5	20.0
3 - Predict	66.7	43.7
4 - Predict	75.6	94.8
5 - Predict	53.3	63.0
Fix	65.6	12.6
Order	96.5	3.0
Write	67.6	22.2

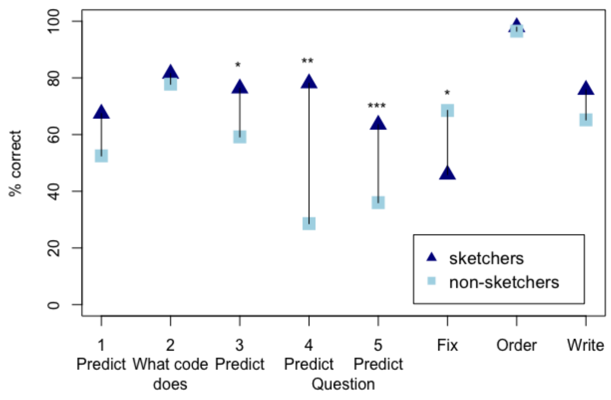

Figure 2: Differences in correctness between sketchers and non-sketchers (* = p <0.10, ** = p <0.05, *** = p <0.01 on Welch two Sample t-test)

4.5 Sketching to avoid split attention effect

Differences in sketches between this study and the LWG study shed light on differences in student cognition between a paper-based exam and a computer-based exam. Students frequently re-wrote pieces of code from questions on their scratch sheets, perhaps to manage issues related to the split attention effect [2]. While students in the LWG study could annotate printed code, in our study student sketches were separated from original problem text on the computer screen. In order to decrease this extrinsic cognitive load, students copied text to be closer to their other sketching.

Rewriting was not a highly successful sketching technique, however, with a success rate about the same as not sketching at all. It is possible that struggling students tried to use Rewrite to relieve some extrinsic cognitive load, but it was not a major assistance.

5 PROFILING SKETCH USE

We extend the LWG analysis to examine use of sketching more deeply, including on additional problem types: fixing code, ordering code, and writing code.

Table 4: Average score on code fixing, ordering, and writing problems when students (N=65) use a particular sketch type from the Leeds Working Group categorization.

Sketch Category	Average score (%)	Data Points
Write (W)	92.7	7
Describe (D)	81.4	14
Blank (B)	77.7	165
Number (N)	71.5	11
Rewrite (R)	65.0	6
Position (P)	40.0	1
Computation (C)	26.7	3

5.1 Some prediction problems trigger sketching more than others

Students chose to sketch the most for code tracing problems (1, 3, 4, and 5). However, even among code tracing problems, some problems were much more frequently sketched than others. 95% of students sketched on question 4, while 44% of students sketched on question 3 (see Table 3).

These differences may be explained by the number of variable states that students had to track in each problem. In question 4, three different variables were updated three times. In question 3, one variable was updated three times, and there were three comparisons. In that question, if students noticed that all comparisons would be false due to a code "error", they did not need to complete a trace. This supports the view that tracing is offloading cognition, which may be more necessary on some problems than others.

5.2 Sketching leads to better predictions

On code prediction problems, sketchers were more successful, sometimes dramatically so (Figure 2). On question 4, those who sketched performed nearly 50 percentage points higher than non-sketchers. For other prediction problems, the difference was 15-25 percentage points. For the problem where students described what a code snippet does, there was almost no difference between sketchers and non-sketchers.

The high rate of sketching on problem 4 suggests that students have some idea when it is a good idea to sketch, at least in cases of high cognitive load.

5.3 Sketching on fix, order, and write code problems

5.3.1 Sketching was associated with lower scores on the fix code problem. Surprisingly, students who sketched on the fix code problem had a lower average score than those who did not sketch (Figure 2) (46% vs 69%, p-value=0.06 on Welch two sample t-test). Only 12% of students sketched on this problem, and the most common sketch type used was Number (a sketch of a single value of a variable). No students chose to sketch a trace on this problem.

Students might have used sketching techniques like Trace fruitfully on the fix code problem, but they chose not to. Maybe they did not perceive the value of sketching when fixing code. Perhaps students' usual strategy when fixing code is to look for patterns, to match the code to their existing schema [21].

5.3.2 There was very little sketching on the order code problem. Only 3% of students sketched on this problem. Both sketchers and non-sketchers were highly successful (average score 97.9% and 96.5% respectively). No students chose to sketch a trace on this problem. Students' low amount of sketching on the order code problem may be further evidence of the low cognitive load of Parsons problems.

5.3.3 Different sketch types were used when writing code. 22% of students sketched on the code reading problem, they scored higher than their non-sketching peers, but not statistically significantly so (p = 0.34 on Welch two sample t-test). The most common sketching technique used was Describe, where students wrote the expected functionality of code on their scratch sheet. Second most common was the Write technique, where students wrote code to solve the problem on paper. This occurred even though students could write their code on the computer. Writers were highly successful, scoring an average of 92.7% (see Table 4). Writing sketches were only used on this problem. It is hard to say whether the increased success of sketchers is related to sketching itself, or an effect of time spent planning before implementing.

5.4 Sketchers take more time

Sketchers took more time than non-sketchers on code fixing (5.5 min vs 12.7 min, p <0.001 on Welch two sample t-test), code ordering (5.4 min vs 12.2 min, p <0.01 on Welch two sample t-test), and code writing problems (7.0 min vs 12.0 min, p <0.001 on Welch two sample t-test). Students who sketched on more code reading problems took more time to complete that part of the test. Correlation between the number of code reading problems sketched and time spent on the code reading questions has an r^2 of 0.64.

This raises the possibility that sketchers score more highly simply because they spend more time problem-solving. However, there is no correlation in our data between time taken and score for any of the problem types (r^2 <0.15 for all).

6 OBSERVING NEW SKETCH TYPES

We replicated the result that a tracing sketch type is highly correlated with correctness on code prediction problems. However, during our data analysis, we noticed that students used several distinct tracing sketch types not distinguished in the Leeds Working Group categorizations. These sketch types were most often used for problem 4, a code prediction problem involving complex variable assignment within the loop body (see Figure 1 for question text). This section explores why certain tracing types might be more representative of the notional machine. We compare the success of students using various techniques and discuss how different populations vary in their use of these sketch types.

6.1 Creating finer distinctions in the LWG sketching taxonomy

We identified four distinct sketching types, shown in Figures 3-6. All of the sketches are traces, involving the tracking of changing variable values over time. All these sketches would be categorized as Trace (T) in the LWG analysis, but we felt their structures were distinct and that they should be distinguished.

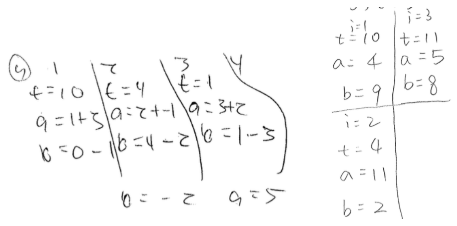

Figure 3: Examples of the "chunk" technique

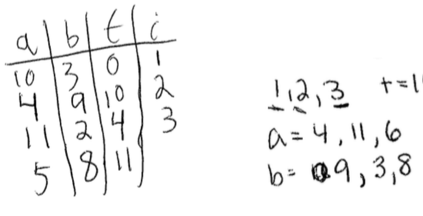

Figure 4: Examples of the "line" technique

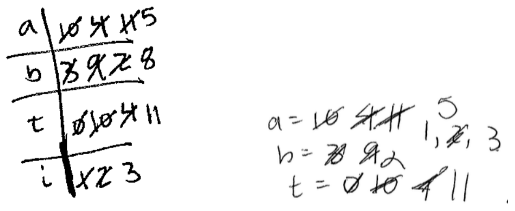

Figure 5: Examples of the "crossout" technique

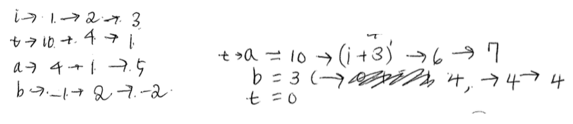

Figure 6: Examples of the "arrow flow" technique

6.1.1 Chunk. This sketch is identified by groupings of assignments, with variable names, equal signs, and the variable value re-written for each loop iteration.

6.1.2 Line. This sketch lists values in a series, following a variable name. The values may simply be written next to each other, or may be separated by commas.

6.1.3 Crossout. This sketch is distinguished by values that are crossed out, usually with a slash. Each variable is written once, with a series of values after it. The value at the end of the line is not crossed out.

6.1.4 Arrow Flow. In this sketch type, variable names are written once, and values are separated by arrows, drawn in the direction of newer values.

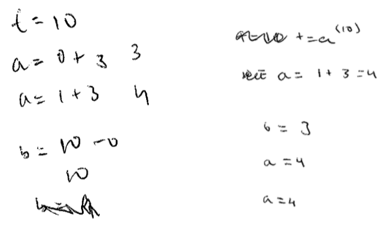

Figure 7: Examples of "incomplete" tracing sketches

6.2 Sketches are different in how they mirror the notional machine

The notional machines of major procedural programming languages share common operations when assignment of primitives is involved. During assignment, a variable's value is overwritten by the new value being assigned. Each variable can hold only one value at a time. Accurate sketching techniques for tracing are a demonstration of this action of the notional machine.

Some sketching techniques from our data seem to mirror the notional machine more clearly than others. Crossout, for instance, demonstrates that previous values are no longer accessible by a strike-through. It is hard to mistakenly use a prior value when it is clearly crossed out. For Line and Arrow Flow, if the sketcher understands that the value at the end of the line is the current value, the sketch communicates this same idea. In contrast to these three techniques, Chunk groups variable values together by the loop iteration when they are assigned. In order to find the most recent value, the student must consider both the current listing of values and the previous listing of values.

From this point of view, Crossout places the least cognitive demand on the sketcher as they act out the notional machine, because information about the most recent value is clearly presented. In contrast, Chunk requires more cognitive effort by the sketcher.

6.3 These tracing sketches were equally successful

We predicted that this cognitive demand would affect students' correctness when using these methods, so that Crossout would have a higher success rate than Chunk.

We coded tracing sketch types used on question 4 (n=135, inter-rater reliability on tracing type used was 93% across 20% of the data). In addition to the four tracing sketch types described in Section 5.1, we also created categories for Incomplete tracing sketches and Other tracing sketches. Incomplete traces did not have a clear sketching structure or include the majority of variable values. Other tracing sketches were traces that were organized and covered all values, but were not from the four main categories we identified. Students overwhelmingly used the chunk tracing method (53.3%), followed by the line tracing method (16.4%).

Difference in correctness on question 4 between students who used the four tracing types was not statistically significant. Students who used one of these methods performed similarly. Differences in representations of the notional machine did not significantly impact students' outcomes in this setting. However, if there were increased time constraints or if students had less programming experience, we may be more likely to see differences between tracing sketches based on this variation in cognitive process.

Table 5: Percentage of correct answers based on tracing sketch type.

Sketch Type	% Correct	Count
Line	92.3	26
Chunk	82.4	77
Crossout	77.8	9
Arrow Flow	50.0	2
Other	50.0	2
None	37.5	8
Incomplete	9.1	11

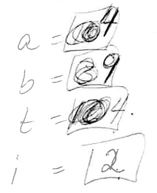

Figure 8: Tracing technique used by the second instructor

6.4 Incomplete tracers and non-tracers were less successful

Students who used no tracing method or used an incomplete tracing method performed much more poorly on this question than students who used a complete tracing method. A statistically significant difference in correctness was found in comparisons between incomplete sketchers and Chunk, Line, and Crossout ($p < 0.001$ on all on Fisher's exact test). A statistically significant difference in correctness was found in comparisons between non-sketchers and those using Chunk and Line ($p < 0.001$ on Fishers Exact Test). The difference between non-sketchers and those using Crossout technique was significant at the lower bar of accepting an alpha value of 0.10 ($p=0.06$).

6.5 Instructors, TAs, and students trace differently

Instructors may influence a student's choice of technique. Participants in this experiment came from two sections of CS1, taught by two different lecturers following the same schedule. Students in these courses were each assigned to a weekly recitation, led by an undergraduate teaching assistant (TA). Both course instructors and 12 of the 17 TAs (70%) were interviewed to determine if they demonstrated tracing techniques similar to those described here (see Table 6).

One instructor self-reported use of Chunk and Line in her classroom, which were the two most commonly-used sketching techniques seen in our data. However, the second instructor self-reported use of a technique not observed in any student sketches,

Table 6: Tracing sketches used by TAs and students

Sketch Type	% of interviewed TAs (n)	% of students (n)
Chunk	16.7% (2)	57.0% (77)
Line	33.3% (4)	19.3% (26)
Crossout	41.7% (5)	6.7% (9)
Arrow flow	8.3% (1)	1.5% (2)
Total	100.0% (12)	84.4% (114)

although 47% of participants were enrolled in this instructor's section. This technique involved a representation of variables as boxes, and the crossing out of prior values as variables took on new values (see Figure 8).

TAs also did not present standardized sketching techniques during recitations and office hours. Interviews with 12 of the 17 TAs showed that they used a wide variety of tracing sketches, although they favored Crossout and Line techniques (see Table 6).

We cannot determine why students used a certain technique without more information, but the data suggests interesting questions. Why did students use the techniques of the first instructor, but never the technique of the second instructor? Although Crossout was favored among TAs, why did students rarely use that technique?

It is possible that Crossout and the method used by the second instructor require more effort to sketch than Line or Chunk. Students may not view the extra effort of striking out values as being worth it, even if it does save cognition later. Even in the effort is similar, students may value sketches that better reflect code syntax.

6.6 Students with prior experience trace differently than first-time coders

Participants in this study were taking an introductory computer science class. However, a demographics survey revealed that 32.6% of participants had prior programming experience. When we categorized students based on programming experience, we found a significant difference in the distribution of tracing techniques used on multiple choice question 4. Students who had prior programming experience used Line much more than first time coders (see Figure 9). Experienced coders still used Chunk the most. Among the four tracing methods, there was a statistically significant difference in tracing technique use between experienced and new coders (p <0.01 on Fisher's exact test).

Similarly, inexperienced coders used Chunk the most, but, unlike experienced coders, they use other coding types less. Instructor 1 stated in her interview that she felt Chunk was more appropriate for students new to programming, while Line was a better fit for students more comfortable with programming. This association seems to be reflected in our data.

Experienced coders used Incomplete much less than first time coders; however, experienced coders were more likely to use no tracing method at all. However, the difference was not statistically significant (p = 0.11 on Fisher's exact test).

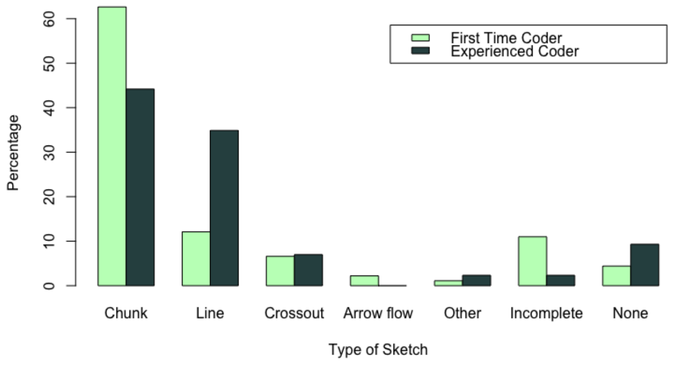

Figure 9: Tracing types used by first time coders and experienced coders

7 CONCLUSION

Our results replicate the LWG's major findings on sketching, extend the LWG analysis to new problem types, and expand the taxonomy of sketches. We re-affirm that tracing is a successful sketch type for use on code reading problems, while not sketching at all has a low success rate. When tracing, completeness of tracing is more predictive of correctness than tracing strategy. Details of tracing sketches do not seem to make much difference.

Sketching is not as frequently used or as successful on code fixing, code ordering, and code writing problems. Sketches like tracing could certainly be used on these problems, such as for testing. Students may not associate sketching with these problem types as much as they do with code prediction. Alternatively, students may have used problem-solving strategies that are not well-represented in sketches, such as pattern-matching with known code patterns.

Our results are consistent with a view of sketching as a technique to distribute cognition and manage cognitive load. When students offload more cognition, as with complete trace sketches, they are more successful. However, such well-ordered sketches may only be possible with strong understanding of the notional machine. Students who do not sketch may lack fundamental knowledge of the notional machine, and are unable to use any sketching technique.

Besides assisting with problem-solving, sketching presents an opportunity for instructor observation and peer interaction. Sketching is visible and can be completed in a group setting. New K-12 computing teachers cite the lack of materials and techniques to aid them in the classroom as a major difficulty [7, 30]. Sketching research can lead to the development of engaging, active pedagogical techniques.

The success of sketching on certain problems, particularly tracing sketches for code prediction problems, supports the idea that all students should be taught a tracing sketch technique. Our results suggest that students' choice of technique is not a straightforward adaptation of their instructor's sketching. If we know more about why students choose a particular technique, we might be able to determine techniques students are most likely to adopt and complete.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, who provided detailed feedback and suggestions that improved this paper.

REFERENCES

[1] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2013. *Computer Science Curricula 2013.* Technical Report. ACM Press and IEEE Computer Society Press. DOI: http://dx.doi.org/10.1145/2534860

[2] Paul Chandler and John Sweller. 1991. Cognitive load theory and the format of instruction. *Cognition and instruction* 8, 4 (1991), 293–332.

[3] Michelene TH Chi, Stephanie A Siler, Heisawn Jeong, Takashi Yamauchi, and Robert G Hausmann. 2001. Learning from human tutoring. *Cognitive Science* 25, 4 (2001), 471–533.

[4] Michelene T. H. Chi, Paul J. Feltovich, and Robert Glaser. 1981. Categorization and Representation of Physics Problems by Experts and Novices. *Cognitive Science* 5, 2 (1981), 121–152. DOI: http://dx.doi.org/10.1207/s15516709cog0502_2

[5] Benedict Du Boulay. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73. DOI: http://dx.doi.org/10.2190/3LFX-9RRF-67T8-UVK9

[6] Robert Ferguson and George M. Bodner. 2008. Making sense of the arrow-pushing formalism among chemistry majors enrolled in organic chemistry. *Chem. Educ. Res. Pract.* 9 (2008), 102–113. Issue 2. DOI: http://dx.doi.org/10.1039/B806225K

[7] Mark Guzdial, Barbara Ericson, Tom McKlin, and Shelly Engelman. 2014. Georgia Computes! An Intervention in a US State, with Formal and Informal Education in a Policy Context. 14, 2 (2014).

[8] Matthew Hertz and Maria Jump. 2013. Trace-Based Teaching in Early Programming Courses. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (2013), 561–566. DOI: http://dx.doi.org/10.1145/2445196.2445364

[9] Mark A. Holliday and David Luginbuhl. 2004. CS1 assessment using memory diagrams. *ACM SIGCSE Bulletin* 36, 1 (2004), 200. DOI: http://dx.doi.org/10.1145/1028174.971373

[10] Edwin Hutchins. 1995. How a cockpit remembers its speeds. *Cognitive science* 19, 3 (1995), 265–288.

[11] Petri Ihantola and Ville Karavirta. 2011. Two-dimensional parsonfis puzzles: The concept, tools, and first observations. *Journal of Information Technology Education* 10 (2011), 2011.

[12] Magdalene Lampert. 1992. Teaching and learning long division for understanding in school. In *Analysis of Arithmetic for Mathematics Teaching*, Rosemary A. Hattrup Gaea Leinhardt, Ralph Putnam (Ed.). Psychology Press, Chapter 4, 221–282.

[13] Raymond Lister, Tony Clear, Dennis J Bouvier, Paul Carter, Anna Eckerdal, Jana Jacková, Mike Lopez, Robert McCartney, Phil Robbins, Otto Seppälä, and others. 2010. Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin* 41, 4 (2010), 156–173.

[14] Raymond Lister, Otto Seppälä, Beth Simon, Lynda Thomas, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, and Kate Sanders. 2004. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, Vol. 36. 119–150. DOI: http://dx.doi.org/10.1145/1041624.1041673

[15] Linxiao Ma. 2007. *Investigating and improving novice programmers' mental models of programming concepts.* Ph.D. Dissertation. University of Strathclyde.

[16] Robert McCartney, Jan Erik Moström, Kate Sanders, and Otto Seppälä. 2004. Questions, Annotations, and Institutions: observations from a study of novice programmers. In *the Fourth Finnish/Baltic Sea Conference on Computer Science Education, October 1–3, 2004 in Koli, Finland.* Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory of Information Processing Science, FINLAND, 11–19.

[17] 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. DOI: http://dx.doi.org/10.1145/572139.572181

[18] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. 2015. Subgoals, Context, and Worked Examples in Learning Computing Problem Solving. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research (ICER '15).* ACM, New York, NY, USA, 21–29. DOI: http://dx.doi.org/10.1145/2787622.2787733

[19] Dale Parsons and Patricia Haden. 2006. Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52 (ACE '06).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 157–163. http://dl.acm.org/citation.cfm?id=1151869.1151890

[20] Roy D Pea. 1993. Practices of distributed intelligence and designs for education. *Distributed cognitions: Psychological and educational considerations* 11 (1993).

[21] Robert S. Rist. 1989. Schema Creation in Programming. *Cognitive Science* 13, 3 (1989), 389–414. DOI: http://dx.doi.org/10.1207/s15516709cog1303_3

[22] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. DOI: http://dx.doi.org/10.1145/2483710.2483713

[23] John Sweller, Jeroen JG Van Merrienboer, and Fred GWC Paas. 1998. Cognitive architecture and instructional design. *Educational psychology review* 10, 3 (1998), 251–296.

[24] Allison Elliott Tew and Mark Guzdial. 2010. Developing a Validated Assessment of Fundamental CS1 Concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education (SIGCSE '10).* ACM, New York, NY, USA, 97–101. DOI: http://dx.doi.org/10.1145/1734263.1734297

[25] Lynda Thomas, Mark Ratcliffe, and Benjy Thomasson. 2004. Scaffolding with Object Diagrams in First Year Programming Classes: Some Unexpected Results. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04).* ACM, New York, NY, USA, 250–254. DOI: http://dx.doi.org/10.1145/971300.971390

[26] Juhani E. Tuovinen. 2000. Optimising Student Cognitive Load in Computer Education. In *Proceedings of the Australasian Conference on Computing Education (ACSE '00).* ACM, New York, NY, USA, 235–241. DOI: http://dx.doi.org/10.1145/359369.359405

[27] Ian Utting, Allison Elliott Tew, Mike McCracken, Lynda Thomas, Dennis Bouvier, Roger Frye, James Paterson, Michael Caspersen, Yifat Ben-David Kolikant, Juha Sorva, and Tadeusz Wilusz. 2013. A Fresh Look at Novice Programmers' Performance and Their Teachers' Expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education - Working Group Reports (ITiCSE -WGR '13).* ACM, New York, NY, USA, 15–32. DOI: http://dx.doi.org/10.1145/2543882.2543884

[28] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in Novice Programmers' Poor Tracing Skills. *SIGCSE Bull.* 39, 3 (June 2007), 236–240. DOI: http://dx.doi.org/10.1145/1269900.1268853

[29] Jacqueline Whalley, Christine Prasad, and P. K. Ajith Kumar. 2007. Decoding Doodles: Novice Programmers and Their Annotations. In *Proceedings of the Ninth Australasian Conference on Computing Education - Volume 66 (ACE '07).* Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 171–178. http://dl.acm.org/citation.cfm?id=1273672.1273693

[30] Aman Yadav, Sarah Gretter, Susanne Hambrusch, and Phil Sands. 2017. Expanding computer science education in schools: understanding teacher experiences and challenges. *Computer Science Education* 26, 4 (2017), 235–254. DOI: http://dx.doi.org/10.1080/08993408.2016.1257418 arXiv:http://dx.doi.org/10.1080/08993408.2016.1257418