How Do Students Solve Parsons Programming Problems? — An Analysis of Interaction Traces

Juha Helminen*, Petri Ihantola, Ville Karavirta, and Lauri Malmi
Department of Computer Science and Engineering
Aalto University
Finland
firstname.lastname@aalto.fi

ABSTRACT

The process of solving a programming assignment is generally invisible to the teacher. We only see the end result and maybe a few snapshots along the way. In order to investigate this process with regard to Parsons problems, we used an online environment for Parsons problems in Python to record a detailed trace of all the interaction during the solving session. In these assignments, learners are to correctly order and indent a given set of code fragments in order to build a functioning program that meets the set requirements. We collected data from students of two programming courses and among other analyses present a visualization of the solution path as an interactive graph that can be used to explore such patterns and anomalies as backtracking and loops in the solution. The results provide insights into students' solving process for these types of problems and ideas on how to improve the assignment environment and its use in programming education.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms

Human Factors

Keywords

Parsons Puzzles, Problem Solving Process, Python

1. INTRODUCTION

Programming is inherently a complex mental process where a programmer solves a domain problem, transforms the solution into an algorithmic form coding it using a programming language, designs an appropriate structure for the program, and finally implements, tests, and debugs the program in an iterative process. In programming education, we train this process heavily by requiring our students to solve many programming assignments. However, the experience has widely been that the results are unsatisfactory and a large number of students have serious difficulties in learning programming [8, 12].

A great challenge in teaching programming is that the whole process of solving a programming task is mostly invisible to teachers because there is no possibility to monitor all students' working and give guidance and feedback for them when they face problems. Even in closed labs with a few dozen students and a tutor, the tutor mostly sees only snapshots of students' work. This has obvious drawbacks. First, we cannot give students enough adequate feedback and guidance when they would need it; even in closed labs they often need to wait a considerable time, until the tutor is available for them, not to mention open labs. Second, as we at best only see a few snapshots, which often have problems or errors, we have a hard time concluding how students ended up with these solutions. Students may also develop bad practices, such as using automatic assessment tools as testers. Poor development habits and misconceptions about programming constructs and concepts acquired early on the introductory programming courses can easily stick. Therefore, we should seek to identify and correct unsound practices and flawed understanding as early as possible.

In this study, we have analyzed how students solve *Parsons programming puzzles*. These are a type of scaffolded program construction tasks where the learner is given a set of code fragments, blocks of a single or multiple lines of code, and the task is to piece together a program from these [13]. We have used a Parsons assignment environment for Python, called jsparsons [5], where learners not only select and order but also indent code fragments. These are called two-dimensional (2D) Parsons problems for the two degrees of freedom. In Python, code indentation has a semantic meaning, as an indented statement falls into the surrounding control structure, which has lower indentation. That is, code blocks are defined by indentation instead of start and end symbols like curly braces (see Tables 3–5 for examples). Compared to the basic Parsons problems, indentation makes such problems better resemble actual programming tasks.

The research reported in this paper has two aims. First, we study methods to visualize and analyze the process of constructing a program. To this end, we have built a tool which performs analyses and allows interactive investigation of Parsons problems' solution paths. Second, by means of the different analyses and visualizations, we study students' problem solving process in 2D Parsons problems. Students solve Parsons assignments in an environment that gives automatic feedback and guides them towards a solution. In addition, the system logs all student interaction, i.e., all reorderings of the code fragments, as well as, feedback requests. Using this kind of data we can explore many interesting questions, such as:

- How do they arrive at the solution? Can we observe

*Corresponding author.

common solving patterns in an assignment or across assignments? How similar or different are the solution paths across students? What types of ineffective solving patterns do students exhibit? Do students' solution paths share common incorrect states, where the rows are ordered or indented incorrectly? How can these states be characterized?

- Do students backtrack or go in circles in their solution paths, i.e., do the solution paths have loops where students return to an earlier state in their solution path? How can these loops be characterized?

- How do students use automatic feedback? When and how often do they ask for feedback? How can the situations when they use feedback be characterized?

2. RELATED WORK

2.1 Parsons Programming Puzzles

Originally, Parsons problems were conceived to provide an engaging learning environment with immediate feedback that allows focused rote learning of syntax without being sidetracked by complex program logic [13]. In addition to online learning environments (e.g. Cort [4], ViLLE [17], and js-parsons [5]), they can be used in traditional pen-and-paper exams where actual programming, i.e., straightforward writing of code from scratch, is problematic [2, 11]. There are many variants and flavours of Parsons problems [5]:

Extra code fragments that are not part of the correct solution can be used to make the problems more challenging. These *distractors* are often modified from a correct line with the aim of revealing misconceptions related to the syntax [13].

A context can be provided by having fixed lines of code before, in the middle, and after the code fragments to be rearranged [4]. This allows making the programs larger and, thus, more concrete and meaningful.

User-defined blocks raise complexity and emphasize problem solving and program logic in the problems. This can be realized by letting learners insert code block delimiters, such as, curly braces in Java.

As discussed in Section 1, the js-parsons tool implements a Python-specific variant of users defining blocks where code fragments must be both re-ordered and indented correctly [5]. The tool is further described in Subsection 3.1.

An interesting question is which types of skills Parsons problems test and how they relate to the commonly assigned tasks of tracing, explaining, and writing code. Initially, it was postulated that Parsons problems lie somewhere between tracing and writing code [21]. Denny et al. found a notable correlation between Parsons scores and code writing scores but a low correlation between tracing and writing, and Parsons problems and tracing [2]. This suggests that tracing and writing require different skills and that Parsons problems are similar to writing code. Whereas, Lopez et al. found strong correlation between tracing and writing, and reading and writing code [11]. Moreover, they found evidence of the existence of a hierarchy of programming-related skills and indications that Parsons problems might be a lower skill than tracing or writing code. However, they note that the difficulty of a task may also be a function of its size and the programming constructs involved than merely the task type. Indeed, as they also note, the one and only Parsons problem they used was rather simple and could mostly have been solved using shallow heuristics, especially, since the code blocks were pre-determined with curly braces. Further studies have found more evidence of a hierarchy where some skill in tracing precedes explaining and some skill in these two precede writing code while all are still believed to develop in parallel reinforcing each other [9, 10, 20]. Nevertheless, Venables et al. do note that the strength of the relationships varies considerably according to the nature of the task [20]. Finally, Lister et al. give some results of a found correlation between the scores of a Parsons problem and scores in tracing and writing in an exam [9]. Overall, the topic warrants some more studies with clearer focus on Parsons problems and a clear separation of the different types and mediums of Parsons problems which may have very different characteristics in relation to the skills being practiced.

2.2 Programming Assignment Trace Analysis

There is a growing body of research into programming behavior analysis based on recorded interaction traces. Blikstein has logged students' actions in the NetLogo programming environment [1]. In the modeling assignments, he identified such behaviors as copy-pasters who would switch away from the environment for long periods of time and then suddenly the code would grow notably. Kiesmüller et al. have studied students' problem solving strategies in the finite state machine -based visual programming environment Kara [7]. They have built a real-time identifier for previously observed problem solving approaches based on methods from speech recognition. Jadud, Rodrigo, Tabanao et al. have studied novice compilation behavior within the BlueJ programming environment [6, 19]. They have presented a visualization for a programming session recorded at compilation events and quantified how much a student struggles in a session based on compilation events. They have shown that this measure correlates negatively with the exam score. Recently, Piech et al. logged compilation events in Eclipse and modeled programming assignment development paths using machine learning [15]. The different groups of development paths correlated with students' performance and their predictive power was stronger than that of the assignment scores. At a larger granularity, analyzing submissions of an automatic assessment system, Edwards et al. have found quantitative evidence, e.g., that starting early relates to better performance [3]. Spacco et al. have presented a schema for representing program evolution with data captured from Marmoset [18] which captures program code at every save. Poncin et al. have applied process mining techniques from business process analysis to investigate software repositories of students' capstone projects [16].

With regard to Parsons problems, in the original paper, the authors suggest that in future versions of the tool they would like to be able to record learners' interaction with the tool in order to analyze their patterns of error [13]. Subsequently, the js-parsons tool has implemented this and the authors intend this feature to allow analysis of how the assignments are solved [5]. However, we are not aware of any previous work that has performed these analyses, or with regard to Python, of anything similar.

3. DATA COLLECTION

3.1 Method and Tool

The js-parsons tool provides a web environment for solving 2D Parsons problems in Python as described in Section 1. In this study, we used a mode where the input, i.e., the building blocks for the code, is given in random order on the left and the learner is to construct the solution on the right by drag-and-dropping code fragments to their place. Code fragments can be freely inserted in the solution between other fragments, moved around, indented, and also removed from the code back to the input area (see Figure 1).

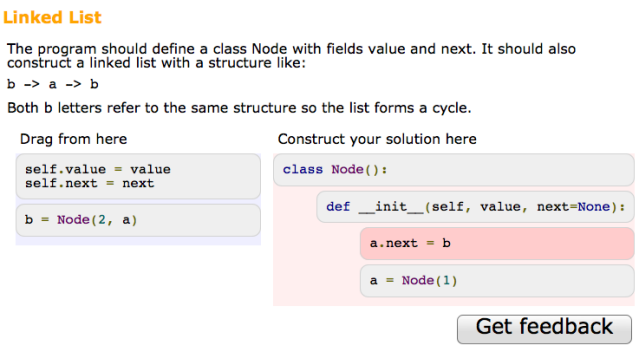

Figure 1: Parsons problem in js-parsons. Student has requested feedback, and the line in incorrect position is highlighted in red.

Feedback can be requested at any time and an unlimited number of times. There are three types of feedback: 1) there are too few lines, 2) the order or 3) the indentation is incorrect. The first two feedback types can occur simultaneously but feedback on indentation is given only after all the right fragments have been added and are in correct order. Previously, js-parsons would give feedback on ordering by highlighting in red the first incorrect fragment counting from the top [5]. However, we felt that this type of feedback would encourage adding lines linearly from top to bottom with a trial-and-error strategy of using repeated feedback requests to select the fragments. Thus, in this study, we modified the feedback so that we highlight a minimal set of fragments that need to be moved to fix the order, and the learner is informed that these fragments are in wrong positions relative to the others (see Figure 1). Incorrect indentation is pointed out by highlighting in red the start of the first incorrectly indented fragment. The size of whitespace used for indentation in Python can vary. For the feedback, indentation is normalized and any particular absolute indentation is not forced but feedback is given on the relative correctness of indentation in the learner's code. In addition to the use of color to highlight errors, a message window pops up. It is important to note that for this type of detailed feedback to be possible, the problem must be puzzle-like in the sense that there is a single correct combination, ordering, and indentation of code fragments, a unique solution.

The tool records a full trace of the learner's interactions during the problem solving session. Any changes in the input, solution, and all feedback requests are recorded with time stamps. We used this trace as the basis of our analyses.

As discussed in Section 2.1, the research is inconclusive on the skills needed to solve Parsons problems. However, we argue that it is not clear that 2D Parsons problems, as described here, are so simple that solving them lies beneath actual programming tasks such as tracing, explaining, and writing code. We argue that, depending on the exact task, they require similar skills and will invoke similar solving patterns and difficulties, and are thus an interesting, while simplified, data source to learn about the program construction process.

3.2 Assignments and Learners

We collected data on the solving of five different problems from students of two different programming courses at Aalto University, Finland. The problems are described below. In some problems, lines of code have been placed together as a fragment in order to force a unique solution. None of the problems included distractors. A student was able to advance to the next problem only after solving the current one.

P1: Find max (8 lines in 7 code fragments): Construct a function that finds the maximum value in a list.

P2: Draw triangle 1 (3 lines): Construct a function that prints a text triangle.

P3: Linked list (7 lines in 6 code fragments): Construct a program that defines a `Node` class and creates two objects to form a linked list with a cycle. See Figure 1 and Table 3.

P4: Draw triangle 2 (6 lines): Construct a function that prints an upside-down triangle. See Table 4.

P5: Sublist test (7 lines): Construct a function that tests whether a list is a sublist of the other. See Table 5.

In Fall 2011, these assignments were used on a Web Software Development (WSD) course. They were part of a compulsory exercise round on Python but students could pass this round without solving these assignments. Still, almost all students solved all of them. The assignments were given in the order described above.

In Spring 2012, the assignments were used on a CS2 course taught with Python. On that course, the assignments were optional, additional exercises. For this course, we switched the order of the first two assignments.

Before solving the assignments, students were asked two questions: *Have you programmed with Python before (yes/no)* and *How much have you programmed before (Python or some other programming language)*. The results of this background survey are summarized in Table 1. Based on the survey, 44% of the students on the WSD course had no previous Python experience whereas for CS2 this was only 11%. Parsons problems have not been used at Aalto University before so it is unlikely that students had previously been exposed to the concept.

4. ANALYSIS AND RESULTS

The recorded interaction traces comprise a lot of data. To aid our analyses, we implemented a tool that does preprocessing and provides different quantitative measures and visualizations of the data. We conceptualized a solution as a graph where the nodes are different states of the code on the right as in Figure 1 and edges are transitions from a state to another invoked with the different possible operations: inserting a code fragment to the (partial) solution on the right,

Table 1: Learners' background on WSD and CS2 courses as evaluated by themselves. Programming experience choices were: 1) I am new to programming or have done very little programming. 2) I know basic programming. I have taken basic courses in programming and/or learned similar skills in my work or hobbies. 3) I have experience in programming. I have taken several courses in programming on various topics and/or have learned similar skills in my work or hobbies. 4) I am an experienced programmer with much practical experience. Programming is my profession or a hobby I am passionate about.

familiar with python	programming experience	WSD	CS2
no	1	4 (7.3%)	1 (25%)
	2	13 (23.6%)	1 (25%)
	3	23 (41.8%)	-
	4	12 (21.8%)	2 (50%)
yes	1	-	2 (6.1%)
	2	22 (31.0%)	12 (36.4%)
	3	35 (49.3%)	16 (48.5%)
	4	14 (19.7%)	3 (9.1%)

moving a code fragment within the solution and thus changing the order or indentation, and removing a code fragment from the solution. Each successfully completed solving session is thus a path from a state of empty code to a state with the correct solution, and the number of steps in the solution path is the number of edges traversed. The minimum is the number of code fragments in the solution which is reached when they are all added directly to their correct place. In addition, at each state along the path the student may have requested feedback. Code can also be rearranged in the input area but students were instructed to build the solution on the right and, looking from data, code reordering in the input area was relatively rare and code cannot be indented there, so we chose to focus only on the solution area.

Because the assignments were optional and not rewarded on the CS2 course, the number of students solving the assignments was quite low. Thus, we chose to focus our analysis efforts on the WSD course only. However, we have used the data from CS2 to validate some of our observations as discussed in Section 5. Furthermore, at first on the WSD course, P1 had an error in that the solution was not unique but that it was possible to build two different solutions with the given code fragments but still only one was accepted. Therefore, we had to omit a large portion of this data. It is also worth noting that P2 was a trivial problem with only three code fragments. So we have focused on P3, P4, and P5 in our analyses. Table 2 gives a summary of the general nature of the data collected as medians and their median absolute deviations (MAD). The table indicates that P4 and P5 were likely the most difficult assignments for the students.

4.1 Use of Automatic Feedback

In general, automatic feedback was used sparingly in solving the assignments. The median of how many times a student asked for feedback in a solution ranged from 1 to 3 in the different assignments and there was little variance. Indeed, the median absolute deviation ranged from 0 to 2. Except for

Table 2: Average steps taken and time spent in solving the assignments. P1, P2, P3, P4, and P5 had 7, 3, 6, 6, and 7 code fragments, respectively.

course	assign.	n	steps med	steps mad	time (sec) med	time (sec) mad	time/steps med	time/steps mad
WSD	P1	73	13	4	185	76.0	13.9	5.3
	P2	142	3	0	26	8	8.3	2.3
	P3	140	10	4	142.5	58	12.4	4.7
	P4	137	9	3	219	98	19	7.6
	P5	136	10	3	247	109.5	19.1	8.0
CS2	P1	29	10	3	116	44	9	3
	P2	31	3	0	37	13	11	3.0
	P3	27	9	3	87	16	9.8	2.8
	P4	20	7.5	1	145	64	20.9	5.7
	P5	17	7	0	177	72	19	7.4

the trivial P2, 35-53% of the students were able complete the assignment on their first feedback request and, thus, most requested feedback more than once. Some few individuals requested feedback dozens of times, up to 62 times in a solution. On closer inspection, these students exhibited trial-and-error-like behavior where the student would request feedback after almost every modification they made and they took little time to think about their next move.

In all the assignments, in over 90% of the solutions, students asked for feedback for the first time only after all the lines belonging to the solution had been added. In most cases (54.8% of all solutions for P1, 90.8% for P2, 82.1% for P3, 78.8% for P4, and 82.4% for P5), this was *immediately* after adding the last line of code. The number is significantly lower for P1. We think this is on the one hand because students may not have at this point yet fully realized that feedback could be requested at any time and an unlimited number of times and on the other hand because an approach where code was indented only after all of it having been added was common as described in the next subsection.

Investigation into the states where feedback was requested reveals that the correct, complete answer is unsurprisingly the most common such state. In the last three assignments, the other most common feedback states were the same as the most common incorrect states described in Subsection 4.3.

4.2 Common Patterns

Overall, the variance of the solution paths across students was notable. As a quantitative measure of this, the solution paths in P3, P4, and P5 had in total 453, 444, and 781 different states, respectively. P5 had a higher count because it had 7 draggable code fragments compared to the 6 for the other two. The number of states that were included in many solution paths is quite low: for P3, P4, and P5 there were only 24, 22, and 14 states that appeared in at least every tenth solution, respectively. Worth noting is that the few low-performing students clicking through with a trial-and-error attitude grow the set of different states with nonsensical states not visited by others.

We examined whether any common patterns emerge across solutions by constructing an aggregate graph of all the solution paths for each assignment (see Figure 2). Each node represents one state and its size is relative to the number of solutions that have it. Start node is labeled with the number of solution paths the graph is built from and a solution is

labeled with an *F*. Node outlines are colored according to the state correctness: black for correctly ordered, red for wrong order, and magenta for states with all code fragments in the correct order but incorrectly indented. Nodes themselves are colored from white to black to indicate the number of solution paths where feedback has been requested in that state. Each edge represents a transition from a state to another in response to changing the code in the solution area by, e.g., moving a code fragment. The width of the edge is relative to the number of solutions that have this transition. Edges are also labeled with this number. Edges are colored according to which drag-and-drop operation they represent: black for adding code, red for changing order, magenta for increasing indentation, pink for decreasing indentation, and brown for removing code. We implemented a tool to interactively browse these graphs that additionally shows the code corresponding to the state when hovering over nodes. To identify common patterns, we focused on transitions performed by most students by filtering out edges with weights less than some varying threshold value, and examining paths that were left.

For all the assignments we could see one primary overall pattern which was the simple add all the code fragments linearly, that is, in top-down order directly to their correct place. In the first assignment, P1, it was also quite common to first add the code fragments with no indentation and then do the indentation separately. This was less common in subsequent assignments and may be due to the students being inexperienced with the interface for solving the exercises. Also, we did not see this with the CS2 course. Because the assignments studied were rather simple and had few lines of code, it is not surprising that we see the pattern mentioned above. However, the variations to this are interesting. Indeed, we can see that many students preferred to first add code fragments that defined a block or control structure such as the `for` loop statement or the `if` branching statement. This is most evident in P5 (aggregate graph in Figure 2) whose solution included two `if` statements and one `for` statement. Students much preferred starting to construct the program with adding the `for` and `if` statements before the others. This is a clear deviation from the straightforward linear approach (the left path in the figure) and an indicator of some structured control flow- or block-driven thinking when constructing the program.

Another way to look at the pattern in which students solve the problems is shown in Tables 3–5. The tables show in which order the code fragments were added to the solution. For example, Table 4 informs us that after the first step, 98.5% of the solutions had the `def`-statement. After adding another line, all of the solutions contained this function signature and most solutions, 73.7%, also had the `for`-line. These tables illustrate the same pattern of preferring to add control structures before other statements.

4.3 Common Difficulties

Common difficulties can be observed by examining the most common[1] incorrect states for each assignment. The most interesting such observations are in P3 (see Table 3 for the code). The object instantiation code was commonly indented inside a wrong block: the same level as the code inside the constructor (in 39% of solutions), the same level as

[1]We considered states present in at least 10% of the solutions to be common.

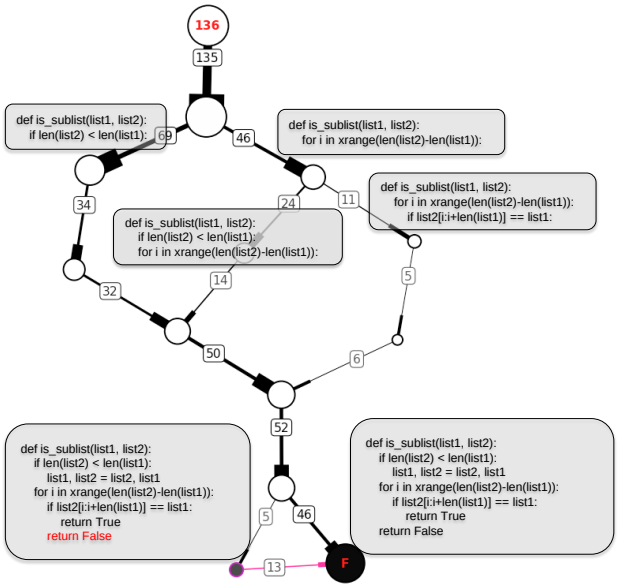

Figure 2: Solution strategies of P5 filtered so that transitions performed by less than five students not visualized. The resulted graph covers 29 % of transitions of all students and 27% of all solution paths.

Table 3: Lines added in P3 on the WSD course.

Code	S1	S2	S3
class Node():	64.3	88.6	97.1
def __init__(self, value, next=None):	35.0	88.6	95.7
self.value = value	0.0	12.9	68.6
self.next = next			
a = Node(1)	0.7	5.0	11.4
b = Node(2, a)	0.0	0.0	3.6
a.next = b	0.0	0.0	0.0

Table 4: Lines added in P4 on the WSD course.

Code	S1	S2	S3
def draw_triangle(h):	98.5	100.0	100.0
stars = (2*h-1)*'*'	0.0	17.5	41.6
for n in xrange(h):	0.7	73.7	94.2
spaces = n	0.0	5.8	40.9
print spaces * ' ' + stars	0.0	0.7	16.1
stars = stars[2:]	0.7	2.2	7.3

Table 5: Lines added in P5 on the WSD course.

Code	S1	S2	S3	S4
def is_sublist(list1, list2):	99.3	100.0	100.0	100.0
if len(list2) < len(list1):	0.7	53.7	77.9	86.8
list1, list2 = list2, list1	0.0	4.4	36.0	60.3
for i in xrange(len(list2)-len(list1)):	0.0	35.3	55.1	87.5
if list2[i:i+len(list1)] == list1:	0.0	4.4	14.0	33.8
return True	0.0	1.5	1.5	6.6
return False	0.0	0.7	14.0	19.9

the constructor definition (33%), and one step deeper than code inside the constructor (11%). Particularly interesting is the last one where the code really makes no sense since the code is indented even though there is no surrounding control structure. It seems these students were just trying to get through with no consideration of whether it made sense.

In assignment P5, there was only one common incorrect state. In that state, the `for`-loop and code inside it was indented one step too deep and thus contained within the wrong block.

Assignment P4 was difficult, and the incorrect states included almost all permutations of the code fragments. Table 6 illustrates this. It shows the positions of code fragments in states where students had requested feedback and all the lines had been added but were ordered incorrectly. From the table we can see, for example, that a common mistake was to have the `print` statement as the last line (69.9% of times, column 6 in the table) while it's correct position was second to last. The values are normalized so that every student's solution regardless of its length or number of feedback requests has equal weight.

Table 6: Positions of code fragments in students' solutions when feedback was requested and the order was incorrect. From each line of the table, one can read the percentage of solutions where the code fragment was in the 1st, 2nd, etc position. For example, 0.2% of solutions had the `def` line as the 3rd line.

Line of Code	1	2	3	4	5	6
def draw_triangle(h):	99	0.7	0.2	0	0	0
stars = (2*h-1)*'*'	0	45	16.4	22.4	9.3	6.9
for n in xrange(h):	0	43.8	44.7	7.4	3.6	0.4
spaces = n	0.8	4.9	29.8	35	28.5	1
print spaces * ' ' + stars	0	0	1.3	7.3	21.8	69.6
stars = stars[2:]	0.2	5.6	7.6	27.9	36.7	22

4.4 Loops and Backtracking

Using the graph visualization introduced in Subsection 4.2, we also examined individual solution paths in addition to the aggregate graphs. Compared to the aggregate graphs, the node size and the edge width have no meaning in these, the start node is labeled with an S, and edges are labeled with the time in seconds between the transition from the state to the next and low values are greyed out to highlight longer pauses (see Figure 3). This state-centric view of the student's solution allowed us to spot the quite common ineffective behavior of revisiting states. Table 7 shows data on how many of the solution paths visited states earlier visited in the same path, that is, the solution paths had loops. Clearly, loops were most common in P4. This is not surprising, since it was also the assignment where students had many common incorrect states.

To better understand the student behavior in these loops, we examined and categorized all the solution paths with loops.

Backtracking happens when the student reverts to an earlier state by undoing the operations exactly in the reverse order. One might assume that in these cases returning to the earlier state was intentional.

Circular loop(s) occurs when the student visits multiple states and then returns to an earlier state via different intermediate states. In these cases it is difficult to say if the student returned to the earlier state intentionally or by accident.

Table 7: Solutions with loops. Count is the number of solutions with at least one state revisit. Length of a loop gives the median and median absolute deviation of the number of steps between state revisits.

Assignment	Count	%	Length of a loop	
			med	mad
P1	19	26.0%	4	3
P2	1	0.7%	3	0
P3	29	20.7%	5	4
P4	45	32.8%	3	2
P5	28	20.6%	3	2

Let us call the shortest path from start to the solution state within a solution path the *trunk*. Combinations of the two types of loops above are possible and can be further categorized based on how they are related to each other and the trunk.

Separate sidetrack(s) , where loops originating from the trunk do not start from the same or consecutive states along the trunk, are illustrated in Figure 3.

Concentrated sidetracks where many loops originate from a smaller number of states in the trunk are illustrated in Figure 3. We call these states concentration points.

Jumbled combinations where the number of concentration points is extremely high, there are nested loops, and the loops are lengthy are illustrated in Figure 4. In some cases drawing the line between this category and *concentrated sidetracks* is difficult.

5. DISCUSSION

The overall aim in our research is to build a tool that would allow us to monitor students' programming process, collect data about the problems they face in the process, and give tailored feedback to them. Though it is technically possible to log all students' interaction when they are using some programming environment, the problem is to get data that has adequate granularity. Automatic assessment tools, such as Web-CAT [3], provide only the submissions. Logging all editing data, on the other hand, would provide us with lots of too detailed data, like writing code sentences, misspellings and correcting them, most of which is not relevant to us. What we would need is data which includes all changes students make in the level of inserting, removing, and reordering syntactically correct statements, expressions, methods and other larger programming concepts, as well as all interaction they have when compiling, testing, and debugging their problems. Here, we have studied Parsons problems in the js-parsons environment. This has enabled us to examine this type of data in terms of puzzle-like program construction sessions.

Nearly all students were able to solve all our Parsons problems. However, how they solved them varied. Thus, if these Parsons problems were used to assess some programming

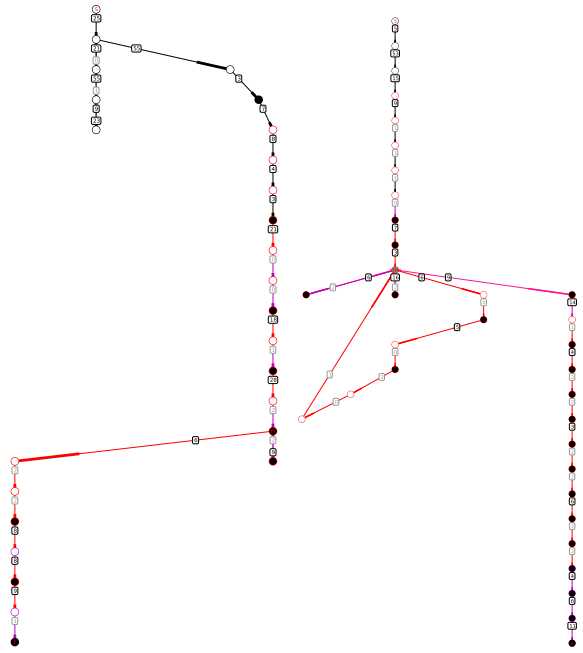

Figure 3: Separate sidetracks on the left and concentrated sidetracks on the right.

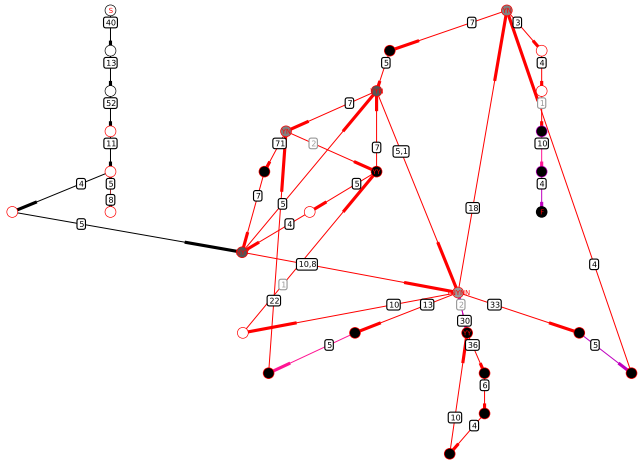

Figure 4: A jumbled combination of loops.

skills, we assume that assessing solving strategies is more meaningful than merely checking the final state. In our previous work, we observed how four experts solved algorithmic 2D Parsons puzzles. They seemed to follow a top-down strategy where they first added the function signature to the solution, then added loops and ordered them correctly if needed, then added the `if`-statements, and finally the remaining lines [5]. As discussed in Subsection 4.2, we found that some students did indeed follow this pattern while others did not. It would be interesting to further analyze whether they did not use the structured approach because they already had the full or partial solution figured out. Nonetheless, even though we can observe the tendency of using this approach, there are a lot of small variations in the solution paths and some guesswork must be present.

To validate the common solving strategies identified on the WSD course, we briefly checked the data collected on the CS2 course. It seems the same patterns were there, but since we only have little data from that course, we cannot definitely say that they were exactly the same. In the future, we need to collect more data and analyze patterns more thoroughly from multiple courses.

In Subsection 4.4, we examined loops in students' solution paths. Having no loops in a solution does not guarantee that the learner was always moving towards the solution, but loops clearly indicate that the strategy was not optimal. We assume that separate sidetracks could be slips or at least something where the learner realized a problem but was able to proceed immediately after returning to the previous state. It is, however, difficult to evaluate what the learners with jumbled state graphs were thinking while solving the assignment. Concentrated sidetracks are more interesting because it looks like a learner got somewhat stuck to the states from where multiple loops originate. In future, we could provide automated feedback when such behavior is identified. This could further be combined with feedback given when we identify *extreme movers*, that is, students who ignore the given feedback or do not use it effectively [14]. This would hopefully guide students away from falling into these ineffective patterns.

As is typical when new educational tools are introduced, we also collected comments on the assignments from students. The opinions of Parsons problems were quite varied ranging from boring to fun. The only emerging theme was that students see these kinds of assignments good for teaching novel solutions to (algorithmic) coding tasks.

We recognize that the current assignments we have, are in many sense artificial compared with actual programming tasks. We therefore direct our future work to extend the analysis to cases, which allow more realistic programming, such as allowing distractors, i.e., code fragments not part of the solution, allowing students to copy lines instead of restricting the number to one, and to construct expressions. We argue that most of the analysis methods we have used, would be readily available for such enhanced assignments. Essentially, this would move the assignments towards visual programming. and thus allow us to investigate closely how students solve programming tasks, what kind of strategies they have, what kind of errors they make and how do they cope with the errors. Another, interesting direction for research would be implementing automatic feedback on observations made from recorded interactions. Finally, one weakness with the analyses in this study is that we can see anomalies and common mistakes in the solutions but we can only speculate where they stem from. Indeed, in further research, we also need to gather data on what students are thinking, what are their reasons for doing what they did using methods such as simulated recall or talk aloud protocols in order to augment our analysis results.

6. CONCLUSIONS

In this paper, we have demonstrated the use of various methods to extract meaningful information from Parsons problems' solving sessions based on a novel data source of automatically recorded detailed interaction traces. We have shown how to visualize the process of solving these problems as an interactive graph that can be used to explore patterns and anomalies in the solving process. By analyzing empirical

data collected from students of two programming courses, we have identified some elements of poorly proceeding solutions, such as, loops in the solution paths. In future work, we plan to experiment with giving automatic feedback on these types of episodes whose recognition requires knowledge of the whole solution path and not just the current state. Moreover, we plan to extend the tool for more realistic programming tasks, collect data from this, and perform similar analyses.

7. REFERENCES

[1] P. Blikstein. Using Learning Analytics to Assess Students' Behavior in Open-Ended Programming Tasks. In *Proceedings of the 1st International Conference on Learning Analytics and Knowledge*, pages 110–116, 2011.

[2] P. Denny, A. Luxton-Reilly, and B. Simon. Evaluating a New Exam Question: Parsons Problems. In *Proceedings of the Fourth international Workshop on Computing Education Research*, pages 113–124, 2008.

[3] S. Edwards, J. Snyder, M. Pérez-Quiñones, A. Allevato, D. Kim, and B. Tretola. Comparing Effective and Ineffective Behaviors of Student Programmers. In *Proceedings of the fifth international workshop on Computing education research*, pages 3–14, 2009.

[4] S. Garner. An Exploration of How a Technology-Facilitated Part-Complete Solution Method Supports the Learning of Computer Programming. *Journal of Issues in Informing Science and Information Technology*, 4:491–501, 2007.

[5] P. Ihantola and V. Karavirta. Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations. *Journal of Information Technology Education: Innovations in Practice*, 10:1–14, 2011.

[6] M. Jadud. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, pages 73–84, 2006.

[7] U. Kiesmüller, S. Sossalla, T. Brinda, and K. Riedhammer. Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, pages 274–278, 2010.

[8] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. McCartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas. A Multi-National Study of Reading and Rracing Skills in Novice Programmers. *ACM SIGCSE Bulletin*, 36(4):119–150, 2004.

[9] R. Lister, T. Clear, D. Bouvier, P. Carter, A. Eckerdal, J. Jacková, M. Lopez, R. McCartney, P. Robbins, O. Seppälä, et al. Naturally Occurring Data as Research Instrument: Analyzing Examination Responses to Study the Novice Programmer. *ACM SIGCSE Bulletin*, 41(4):156–173, 2010.

[10] R. Lister, C. Fidge, and D. Teague. Further Evidence of a Relationship between Explaining, Tracing and Writing Skills in Introductory Programming. *ACM SIGCSE Bulletin*, 41(3):161–165, 2009.

[11] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth international Workshop on Computing Education Research*, pages 101–112, 2008.

[12] 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. *ACM SIGCSE Bulletin*, 33(4):125–180, 2001.

[13] D. Parsons and P. Haden. Parson's Programming Puzzles: a Fun and Effective Learning Tool for First Programming Courses. In *Proceedings of the 8th Australasian Conference on Computing Education*, pages 157–163, 2006.

[14] D. N. Perkins, S. Schwartz, and R. Simmons. Instructional Strategies for the Problems of Novice Programmers. *Teaching and Learning Computer Programming: Multiple Research Perspectives*, 1990.

[15] C. Piech, M. Sahami, D. Koller, S. Cooper, and P. Blikstein. Modeling How Students Learn to Program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 153–160, 2012.

[16] W. Poncin, A. Serebrenik, and M. van den Brand. Mining Student Capstone Projects with FRASR and ProM. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 87–96, 2011.

[17] T. Rajala, M.-J. Laakso, E. Kaila, and T. Salakoski. VILLE — a Language-Independent Program Visualization Tool. In *Seventh Baltic Sea Conference on Computing Education Research*, pages 151–159, 2007.

[18] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh. Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[19] E. Tabanao, M. Rodrigo, and M. Jadud. Predicting At-Risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the seventh international workshop on Computing education research*, pages 85–92, 2011.

[20] 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.

[21] J. Whalley and P. Robbins. Report on the Fourth BRACElet Workshop. *Bulletin of Applied Computing and Information Technology*, 5(1), 2007.