Investigating Strategies used by Novice and Expert Users to Solve Parsons Problems in a Mobile Python Tutor

Geela FABIC*, Antonija MITROVIC[a] & Kourosh NESHATIAN[a]
[a]*Computer Science and Software Engineering, University of Canterbury, New Zealand*
geela.fabic@pg.canterbury.ac.nz

Abstract: We present PyKinetic, a mobile tutor for Python. The tutor is aimed at novices and meant to be a complement to traditional lectures and labs. The first type of activities implemented in the tutor is Parsons problems, which present code snippets to be ordered by the student to produce the desired output. As a starting point towards an intelligent tutor, we conducted a pilot study to evaluate the interface and usability of PyKinetic, and to identify and contrast strategies used by novice learners with those of experts. Great feedback and enthusiasm was received for the prospect of PyKinetic and interesting strategies were revealed from both groups. The study revealed that experts, as can be expected, outperformed novice users and used superior problem-solving strategies. In future work, we will improve PyKinetic's problems, feedback, activities and extend PyKinetic to provide instruction on optimal problem-solving strategies.

Keywords: Mobile Python tutor; Parsons problems; novice/expert differences; problem-solving strategies

1. Introduction

Learning programming is challenging: the student has to learn the syntax and semantics of the programming language and understand its purpose in context, perform problem solving tasks, logical thinking, as well as develop design skills and strategies (Linn and Dalbey, 1985). Apart from learning programming concepts, the student must also understand concepts behind programming like how code structures are being compiled and translated, as well as how programs are executed. Novice programmers find it rather difficult to grasp these concepts, which might lower their motivation to learn more and to practice. It takes about ten years of experience for a novice programmer to become an expert (Winslow, 1996).

Intelligent Tutoring Systems (ITSs) are knowledge-based systems that simulate the behavior of good human teachers and provide individualized feedback to students (Woolf, 2010). ITSs have been proven effective in supporting student learning in different domains (Koedinger et al., 1997; Heift and Nicholson, 2001; Melis et al., 2001, Mitrovic, 2003, 2012; van Lehn et al., 2005; Weber and Brusilovsky, 2001). When learning a programming language, having access to an ITS is valuable for students to practice, as it is impossible to have a human tutor available at all times. The activities included in the ITS can be designed to focus on increasing engagement, improving students' self-efficacy and motivation for learning. Some students may also find that using an ITS is less socially awkward since it does not involve directly communicating with a human tutor. Hence, this may motivate some students to use an ITS more frequently in their own time and will therefore contribute to opportunities of gaining deeper understanding of the domain.

Python is widely used as a programming language in universities nowadays to teach introductory programming (Guo, 2013). This project aims to develop *PyKinetic* (Fabic, Mitrovic and Neshatian, 2016), a mobile tutor hoping that it would appeal better to a new generation of students, compared to desktop or Web-based educational tools. Apart from the thriving popularity of smart phones and mobile applications, a mobile tutor could potentially target engagement.

We present a prototype of PyKinetic, aimed to be developed as a constraint-based intelligent tutoring system (Ohlsson, 1994). PyKinetic will be a complement to traditional lecture and lab-based introductory programming courses. The current version of PyKinetic contains only one type of activity – *Parsons problems*. Parsons problems (Parsons and Haden, 2006) are exercises requiring the student to rearrange a given set of randomized lines of code to produce an expected outcome. The prototype currently contains two types of Parsons problems, with and without distractors (extra lines of code). In the future, we plan to add additional types of learning activities.

As an initial step towards an intelligent tutor for Python, we performed a study with PyKinetic, in order to identify problem-solving strategies used by novices and experts. Our hypothesis was that experts would outperform novices in speed and efficiency in solving problems, and use optimal problem-solving strategies. The motivation of the study is to enhance PyKinetic to teach students not only about Python, but also to provide instruction about optimal problem-solving strategies. In the following section, we present research done on Parsons problems, educational systems for Python and evaluations of problem-solving strategies used in solving Parsons problems. We then introduce PyKinetic, followed by the experiment design and the findings. Section 7 discusses the problem-solving strategies used by novices and experts. Lastly, we present our conclusions and compare problem-solving strategies observed in our study with those of other studies.

2. Related Work

Parsons problems were originally designed to promote a fun way for students of an introductory course in Turbo Pascal to improve their skills in syntactic constructs (Parsons and Haden, 2006). These programming activities are suitable for novices, as they contain syntactically correct code that only needs to be put in the right order. A variation includes puzzles with syntactically incorrect or unnecessary lines of code (referred to as *distractors*) which students need to eliminate.

Denny et al. (2008) considered five variants of Parsons. The first two variants contain no distractors, with the difference that one of them includes scaffolding such as curly braces and indentation (since this was used in the context of Java), while the second variant does not provide any. The next two variants are composed of paired options for each line of code given in a randomized order but the paired options are clearly placed right next to each other. These variants were available with and without scaffolding. The last variant contains pairs of options for each line of code, but these pairs are provided in a random order. It was not specified whether the last variant was presented with or without scaffolding but this variant ended up being discarded as it was perceived to be unreasonably difficult. For example, 7 lines of code for the puzzle becomes 14 and having these 14 lines of code in a completely randomized order may be viewed by students to be overwhelming to attempt.

There is no widespread agreement on how Parsons problems compare to other types of exercises typically used in introductory programming courses, such as code reading, tracing, writing and explaining. Code tracing falls into lower categories in Bloom's taxonomy, while code writing requires higher order skills (Thompson et al., 2008). Some researchers find Parsons problems are easier than code tracing (Lopez et al., 2008), while others view Parsons problems to lie in between code tracing and code writing (Lister et al., 2010). Denny et al. (2008) found a moderate positive correlation between scores on Parsons problems and code writing questions, but only a weak correlation between Parsons problems and code tracing questions. Therefore, they suggested that Parsons problems were similar to code writing. There are also opinions that the position of Parsons problems in the hierarchy of programming skills can vary, depending on their type (with or without distractors) and complexity (Ihantola and Karavirta, 2011). Other possible factors could include the interface used (on paper or online), and scaffolding and feedback provided.

There are some Python learning environments developed as mobile applications. An example is Quiz&Learn Python[1], which is a game to test and improve knowledge on Python 2.x programming available on Android and iOS devices. The aim of the game is to answer 20 multi-choice questions and to answer them correctly within one minute for each question as fast as possible to gain more points. There are four help options that can be used only once each for every game: remove two incorrect answers, skip a question, debug the code and stop the timer. Remove incorrect answers

[1] http://www.villekaravirta.com/projects/quizlearn-python/

removes two incorrect choices out of the four given. Choosing to skip a question, skips the current question to move on to the next without answering it. Choosing the help option to debug the code gives the users an access to a debugger which shows a line by line visualization of the execution of the given code snippet. The last option (stop the timer) gives the user unlimited time to answer a question. The game ends when the user answers a question incorrectly, has ran out of time, or has successfully answered all 20 questions. The application is developed using Apache Cordova, Zepto.js, Topcoat, SASS, Node.js and PostgreSQL. Based on observations while using the application, it seems that it is presented more as a game rather than a tutor with game elements. It seems to be more focused on gaming features rather than providing pedagogical aspects to support learning.

There are some educational environments for Parsons problems. Ihantola and Karavirta (2011) present js-parsons, a JavaScript library for developing Parsons problems. The library supports "two dimensional" Parsons problems, which allow students to drag lines of code (LOCs) from a set on the left-hand side of the screen and drop it on the solution space on the right-hand side. The second dimension feature is that indentations are supported and students can change the indentations of the LOCs in their solutions. The library is language independent, and can be used to develop Parsons problems for any programming language. Since js-parsons is a JavaScript library, it can be used to develop Parsons problems on webpages designed for personal computers or mobile webpages for tablets and smartphone devices.

There are limited results in literature about problem-solving strategies used by novices and experts for Parsons problems. Ihantola and Karavirta (2011) report on a small study involving four experts. The study was conducted in JSParsons, a Web environment for Parsons problems built using the js-parsons library. The study presented ten Parsons problems with distractors, which required indentations to be specified. For each problem, the name of the algorithm was provided (such as insertion sort). All experts used the same strategy, starting with method signatures, then proceeding with loops and conditionals, and only at the end dealing with initialization of variables and indentation. Another study conducted with students (Helminen et al., 2012) found that students followed a top-to-bottom strategy for simple Parsons problems. In two problems, the first step in 98.5-99.3% of the solutions was to position the function signature. The same researchers also developed MobileParsons (Karavirta et al., 2012) for iOS and Android mobile devices. The interface presented the problem area on top and the solution area below in portrait mode, and side by side in landscape mode. MobileParsons was further developed to allow limited editing of lines (Ihantola et al., 2013).

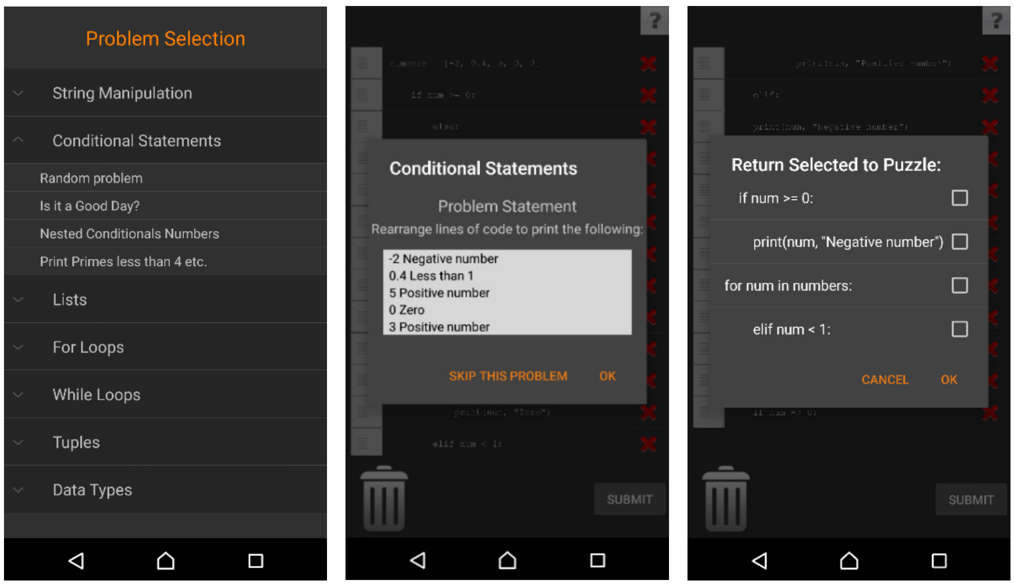
3. PyKinetic

PyKinetic is a Python tutor developed using Android SDK to be used on smartphones. The tutor is aimed to serve as an additional resource for novice learners to enhance their Python 3.x programming skills. The prototype currently contains 53 Parsons problems, covering the following topics: *String Manipulation*, *Conditional Statements*, *Lists*, *For Loops*, *While Loops*, *Dictionaries*, *Tuples* and *Data Types*. The learner needs to rearrange given LOCs to form a correct code snippet that would produce the expected result. There are two types of problems, with or without distractors. The number of LOCs per problem ranges from 3 to 16, with a maximum of 5 distractors. The learner can expand any topic to see available problems (Figure 1, left), and select either a specific problem or ask for a random problem. The selected problem is then presented to the learner, together with the problem statement (Figure 1, middle). The learner can view the problem statement at any time during problem solving (by clicking on the "?" icon on the top-right hand corner). Distractors can be removed by tapping on the red X on the right of each LOC. Deleted lines can be retrieved by tapping on the trash icon and selecting desired LOCs (Figure 1, right).

In this prototype, the problem space containing LOCs also serves as the solution space. This is different to other implementations of Parsons problems (Ihantola and Karavirta, 2011; Helminen et al., 2012; Karavirta et al., 2012; Ihantola et al., 2013), where LOCs need to be dragged across from the problem area to the solution area. We decided to combine the problem and solution into a single area in order to maximize the use of space.

There are problems of varying difficulty and complexity in the tutor. Each problem is assigned the complexity level, ranging from 1 (the easiest problems) to 9. Most problems are only code snippets, but some are functions and include function calls.

The learner can submit his/her solution to be checked at any time. The tutor contains correct solutions for problems including alternative solutions, and the student's solution is matched to them. The prototype currently only provides simple feedback, informing the learner that the solution is correct, or specifying that there are still some distractors left, or LOCs missing. Feedback also informs the student whether the order of LOCs is right, when LOCs are selected correctly. We plan to enhance the diagnosis process in the next version by developing a constraint-based model of the domain (Mitrovic, 2012).



Rearrange lines of code to print the following:

```
-2 Negative number
0.4 Less than 1
5 Positive number
0 Zero
3 Positive number
```

```
if num >= 0:
print(num, "Negative number")
for num in numbers:
elif num < 1:
```

Figure 3. Topic/problem selection screen (left); an example Parsons problem (middle); Retrieving LOCs from trash (right)

4. Experiment Design

The novice participants were 8 volunteers (4 male, 4 female) recruited from an introductory programming course at University of Canterbury. The five expert participants were tutors teaching the same course. The study consisted of individual, one-hour long sessions. The pilot study was conducted in September 2015, by which time the students had learnt about seven topics covered in PyKinetic (problems on Dictionaries were not included in the pilot). The version of PyKinetic used in the study contained 21 problems in total: for each topic, there were two problems with distractors and one without. The problems used in the study had 3 - 16 LOCs, with a maximum of 5 distractors. Four problems were forced to the landscape mode since they contained long LOCs, while the rest were in the portrait mode.

After providing informed consent, the participants interacted with PyKinetic. The novices were free to choose problems as they wished, but were asked to attempt at least one problem from each topic. We used the think-aloud protocol (Ericsson and Simon, 1993), asking the participants to verbalize their thoughts while interacting with the tutor. The screen of the device used for the study was recorded including audio verbal comments of participants. At the end, participants filled a questionnaire, the first part of which included questions about their background, while the second part included multi-choice and open questions about PyKinetic.

We conducted sessions with experts later on, and asked them to attempt the problems that the majority of novice participants attempted, in order to compare the problem-solving strategies used. The experiment design was similar to the setup by (Ihantola and Karavirta, 2011). In their study, they have observed strategies used by experts in solving Parsons problems. Ihantola and Karavirta also conducted a study with novices to identify other strategies (Helminen et al., 2012).

5. General Findings

When asked how much experience the participants had with Python, using the Likert scale from 1 (*Not so experienced*) to 7 (*Highly experienced*), the mean reply of novice participants was 2.12 (sd = 1.25), with only one novice judging his/her experience as 5, and the rest as either 1 or 2. The mean on the same question for experts was 5.4 (sd = 1.34), ranging from 4 to 7.

We eliminated data about problems which the participants only viewed but performed no actions on, and also data about two problems that were found to be buggy. A problem is considered as attempted if the participant made at least one action on it, either by dragging and/or deleting LOCs, viewing the trash, or submitting the solution. A move is defined as the moving of LOCs when attempting a problem. In the way the Parsons problems were setup in the tutor, each move could affect the positions of other LOCs. The analysis was made simpler by counting a move as one regardless of the difference between the starting and ending positions. However, a move is considered as an abandoned move if it was dropped on the same position where it was dragged from.

We used the Mann Whitney U test to analyze similarities and differences between the two groups, with the significance level of 0.05. Table 1 reports the averages (standard deviations are given in parentheses) for the number of abandoned and completed problems. The table also reports the averages for attempted problems: submissions, moves, abandoned moves, time taken, problem complexity, the number of LOCs/distractors, and the number of times problem statement was viewed. The experts have not abandoned any problems, while two novices abandoned two problems each, and two other novices abandoned a single problem each. The experts solved more problems in fewer submissions/moves and in a shorter time, as expected. The only significant difference on the distributions of the two groups was found for the number of submissions per problem (p = .002), and there was a marginally significant difference on the number of moves per problem.

Table 1. Overall Results (** denotes significance on the .01 level)

Measure	Novices	Experts	U, p
Abandoned problems	.75 (.89)	0 (0)	ns
Completed problems	10 (3)	12.8 (3.7)	ns
Submissions	2.72 (1.63)	1.29 (.08)	U = 0, p < .005**
Moves	13.65 (11.51)	7.3 (1.45)	U = 8, p = .093
Abandoned moves	.62 (1.02)	.26 (.17)	ns
Time taken (min)	4.02 (2.6)	2.82 (1.1)	ns
Problem complexity	3.79 (.78)	4.36 (.55)	ns
Distractors	1.66 (.67)	2.03 (.34)	ns
LOCs	8.75 (1.52)	8.87 (1.3)	ns
Problem statement viewed	3.1 (.85)	3.7 (.8)	ns

We also categorized problems by topic (for the seven Python topics used in the study), as well as by the number of LOCs and distractors, and calculated the same measures. Using the number of LOCs, we divided problems into long (11-16 LOCs), medium (7-10 LOCs) and short (3-6 LOCs). The significant and marginally significant differences found are reported in Table 2. The experts were faster in solving problems of most types, apart from While loops, but the only significant difference in time was for Conditionals. The reason why the experts needed more time for the While loop problems is that they attempted more complex problems of this category (in terms of the problem difficulty, and the numbers of distractors and LOCs – all three differences are significant). The experts also attempted more complex problems on Data types, and were significantly faster (in terms of time and the number of submissions) in problems containing many distractors. They needed fewer submissions to complete problems on Lists (marginally significant difference) and also fewer submissions for long problems.

The highest number of errors for both groups was for the problems on Lists. One source of confusion was related to indexing lists (e.g. *my_list[:2:-1]*). All novices and two experts commented that they were used to using only one colon indexing a list. This was probably one of the reasons for the marginal difference between average submissions for this category.

Both groups were advised that the problems were presented in the increasing order of difficulty. We observed that most novices started with easier problems, while the experts randomly picked a problem from each topic without focusing on their difficulty. There was a significant difference for difficulty and the number of LOCs for short problems. The experts also needed significantly fewer moves for problems with a medium number of distractors (2-3 distractors). In addition, the experts viewed the problem statement significantly more often in the case of problems with few distractors (0 or 1). A potential reason for this difference is because experts use better strategies: many novices used trial and error (as discussed in the following section), while experts are likely to think about the problem more and review its requirements.

Table 2. Results by Problem Category (* denotes significance at the .05 level)

Problem Category	**Measure**	**Novices**	**Experts**	**U, p**
Conditionals	Time	6.1 (2.96)	4.26 (.71)	6, .045*
Lists	Submissions	3.6 (2.98)	1.33 (.33)	7, .065
While Loops	LOCs	5.25(2.74)	10.3 (1.56)	38, .006**
While Loops	Distractors	.75 (.8)	2.7 (.67)	38, .006**
While Loops	Difficulty	1.87 (1.25)	4.6 (.89)	38, .006**
Data Types	Difficulty	3.46 (2)	5.53 (.96)	35, .03*
Long problems	Submissions	2.76 (1.58)	1.18 (.25)	5, .03*
Short problems	LOCs	4.53 (.68)	5.17 (.47)	35.5, .019*
Short problems	Difficulty	1.78 (.97)	2.37 (.44)	34, .045*
Many Distractors	Time	6.76 (2.77)	3.18 (.52)	1, .003**
Many Distractors	Submissions	3.85 (2.35)	1.22 (.22)	.5, .002**
Medium Distractors	Moves	11.6 (9.82)	5.97 (.74)	6.5, .045*
Few Distractors	Distractors	0 (0)	.21 (.04)	40, .002**
Few Distractors	Viewed	1.65 (.49)	2.96 (.95)	38, .006**

6. Questionnaire Responses

Overall, the participants were enthusiastic about the tutor, as seen from the questionnaire responses, summarized in Table 3. The participants from both groups found PyKinetic intuitive, easy to use and fun (the average ratings ranged from 5 to 5.6). Some participants seemed to appreciate the interface and commented: "*It's nice how it pops up showing you what to do*" and "*Oh wow, that's cool how you can like slide them up... that's nice*."

When asked whether they improved their skills by interacting with PyKinetic, the average response from novices was significantly higher than that of experts (U = .5, p = .002). It is not surprising that the experts' responses to this question are much lower, as the problems were designed for novices. A few novices seemed surprised that they learned something from the tutor: some comments were "*I'm actually learning something here!*", "*Oh cool I didn't know you could do something like that*." and "*I'm learning stuff while doing it so that's always a plus*."

Both groups seemed to perceive the provided problems having the right amount of difficulty (the experts were asked whether problems are at the appropriate level of difficulty for novice learners). The lowest rating was received from the novices about the amount of feedback provided by the tutor

(2.88). This was expected, since the prototype only provides simple feedback which is only available upon submission. It is interesting that experts scored the feedback much higher. There was a statistically significant difference on feedback rating (U = 34, p = .045).

When asked whether they would use the tutor again, seven out of eight novices agreed (the novice who disagreed specified he/she was not interested in learning more about programming). Two experts also stated they would like to use the tutor again since they gained new knowledge from the tutor, specifically on indexing lists. One participant mentioned "*I can really see myself practicing Python with this on the bus or if I'm waiting for someone*." Both groups were also asked to select programming skills they used in the tutor (reading, syntax and structure and/or logical and semantic reasoning skills). Half of the novices responded they used all those skills, while the other novices selected 2/3 skills. All experts responded they used all the skills.

Table 3. Summary of questionnaire responses

Question (1 Lowest to 7 Highest)	**Novices**	**Experts**
Was the tutor's interface intuitive and easy to use?	5.13 (0.83)	5.6 (0.55)
Was the tutor fun to interact with?	5.13 (0.99)	5 (1.41)
Would you say you have learned some new things and/or enhanced your skills by interacting with the tutor?	5.75 (0.89)	2.4 (1.34)
Do you think it is beneficial that this tutor is developed on a mobile platform?	5.25 (1.04)	4.8 (1.92)
Were problem statements clear enough to understand what needed to be done?	4.5 (1.41)	4.8 (1.92)
Please rate the average difficulty of the problems in the tutor.	4.5 (0.53)	4 (0.71)
Do you think there is enough feedback given when attempting a problem?	2.88 (0.99)	4.2 (0.84)

7. Problem-Solving Strategies

We watched the video recordings of the participants' interactions with PyKinetic and manually observed and identified strategies made by the participants. A wide range of strategies was observed, some of which were used by participants in both groups. An example is to focus on a particular type of LOC and move it (referred to as *Selecting a LOC*). The participants usually looked for variable declarations, function calls and print statements, possibly because variable declarations and function calls are normally located somewhere at the beginning of a program, whilst print statements are normally positioned at the end. One participant made a comment along these lines: s/he just started a problem, noticed a print statement and mentioned the following while dragging the LOC in position: "*Print statements at the end*." This strategy was used at least once by each novice. One expert used this strategy. However, it is important to note that this strategy was not used in all problems; it was observed that the participants' strategies changed depending on the nature of the problem and its expected output.

A more specific version of this strategy was used for problems with functions, when the participants moved the function statement first, followed by the docstring. This strategy was very evident in both groups: five novices and four experts used this strategy for all problems that contained functions. The only situations when this strategy was not used were when those statements were already in place (please note that LOCs were presented in a random order), or when the participant was clearly missing the relevant declarative knowledge. The latter was observed only with novices who used sub-optimal strategies (discussed in Section 7.1).

Another strategy used by both novices and experts was to move distractors (except the very obvious ones) to the end of the solution. Some of the statements novices made during this strategy were "*just in case I still need it*" or "*I don't want to delete the other print lines yet just in case I do need them, but I'll put them down the bottom*." Only two experts used this strategy since experts were generally better at eliminating distractors. Having said that, it seemed that the experts were only doing so because they were too focused on their model solutions to deal with distractors immediately.

All of these strategies require domain knowledge: knowing relative position for specific types of statements, or being able to identify distractors. However, the majority of other observed strategies were used exclusively by one group of participants; those strategies clearly show the difference in domain knowledge between novices and experts. We present those strategies in the following subsections.

7.1 Strategies Used by Novices

Half of the novices grouped LOCs on the basis of their indentations. Such a strategy shows lack of knowledge, as novices were relying on a superficial feature rather than trying to understand the meaning of LOCs. The reliance on the indentations as scaffolding was also mentioned by a participant: "*Sometimes with the loops ... the indentations give away a lot and you can just you know ... without having to read much on what they mean*." This strategy allowed novices to eliminate distractors. The strategy was also useful for arranging the LOCs logically, especially with conditional statements. After applying this strategy, the novices either tried to reason about the LOCs in each group, or used the trial and error strategy. One participant mentioned "*Okay let's put all the indentations at the same...*" then tried to read the code, to find the correct lines. Another novice mentioned: "*So I'm like trying to find the systematic way of like sorting it*." Following this, the novice also mentioned "*So now I'm gonna work out which ones would make sense*."

A common strategy used by novices was trial and error. After solving parts of the problem the participant was knowledgeable about, the participant then tried to solve the rest of the problem by exploring possible solutions, which resulted in multiple submissions. For example, the participant would move a single LOC and submit the solution immediately, in order to eliminate wrong solutions. In some of the situations, the novices asked the researcher for help. This strategy was used when the novices were struggling with problems, therefore illustrating lack of knowledge. Additional evidence can be observed from their utterances, such as "*I'm just gonna get to try all of them and figure out why*" and "*This is one of the questions that is probably more complex than my brain ... whether or not I give up ... I don't know*". Three out of eight novices used trial and error, and two other novices commented that they could see that trial and error can be used as a strategy.

One novice used a unique strategy, when he/she deleted all the LOCs, and then retrieved the necessary ones from the trash. The participant eventually abandoned the problem, so this strategy might be due to high cognitive load.

7.2 Strategies Observed in Experts

A common strategy used by experts was to build the solution from top to bottom (referred to as the top-down strategy). For example, some experts mentioned that function statements have to be first, so they looked for this line and moved it first, then the docstring and other LOCs, until the return or print statement. This strategy shows that experts have a model of the solution, and are working towards matching it. All experts used this strategy, but not always exclusively. One expert alternated between this and another strategy, which consisted of combining syntactically and logically similar LOCs with similar indentations and then logically placing them in the correct order (e.g. similar print statements with similar indentations placed at the bottom).

While the experts were searching for LOCs according to their model solution, three of them were at the same time deleting distractors which were syntactically incorrect lines of code. The other two, on the contrary, left such LOCs and deleted them at the end, although it was clear they understood those LOCs were distractors. Generally, the experts were good at identifying distractors.

8. Discussion and Conclusions

We reported on a pilot study with a prototype of a mobile Python tutor which contained Parsons problems. Our primary goal was to investigate problem-solving strategies used by novices and experts. The study was conducted with 8 novice students and 5 experts. The participants were generally enthusiastic about the prospect of using PyKinetic as an additional tool to learn Python, and found the problems of appropriate nature and complexity. We received good suggestions for further improvement of PyKinetic, such as adding a short tutorial for first-time users, improved feedback and hints on solving problems. The enthusiasm from the participants was encouraging, with seven out of eight novices and two out of five experts interested to use the tutor again.

Feedback received also included suggestions for the interface. Overall, the interface was considered to be intuitive and user-friendly. As mentioned earlier, Parsons problems were presented in PyKinetic in either portrait or landscape mode, which gave us additional insights about the interface. It was observed that in problems presented in the landscape mode, most LOCs were obscured, which seemed to increase extraneous cognitive load for many novices. Some participants commented that the problems in the landscape mode seemed more difficult because the full view of the problem was not available.

We have observed several effective problem-solving strategies used by both novices and experts such as using declarative knowledge to focus on particular LOCs and position them first, and moving distractors to the end of the code. The strategies used by experts demonstrated a higher level of knowledge, as they mostly used the top-down strategy. One expert used an optimal strategy of grouping LOCs with similar indentations, syntax and semantics then logically placing them in their respective positions. We have also observed several strategies when dealing with distractors. Experts appeared to be better in identifying distractors compared to novices.

As mentioned in Section 5, our experiment design was similar with the study conducted by Ihantola and Karavirta (2011). The number of experts in their study were similar to ours. Most experts in our study followed a top-down strategy, solving the problem from the function statement through to the return or print statement. Ihantola and Karavirta reported a similar top-down strategy. However, they have not observed the experts to move all lines perfectly (in the correct order). This is maybe because of the algorithmic nature of their problems compared to ours, which focused on honing basic Python programming skills for novices. Nevertheless, we have confirmed their findings on the top-down strategy observed in experts. This shows that experts solving Parsons puzzles are working towards a mental model solution.

The novices used sub-optimal strategies such as trial and error. None of the novices used the top-down strategy; this contradicts the findings reported by Helminen et al. (2012), where majority of the novices were observed to follow the top-down strategy. The reason for this may be the noticeable difference between the length and complexity of the problems used in their study (five problems with 3-8 LOCs without distractors), compared to our study involving 21 problems with 3-16 LOCs and 0-5 distractors per problem. Helminen et al. (2012) also focused on analyzing only three out of the five problems, which made their data set smaller. However, they have also observed a more specific strategy for Selecting a LOC which was to select a *for* loop or an *if* statement first (Helminen et al., 2012).

Another strategy we have observed for novices was to group LOCs by indentation, which is based on superficial scaffolding feature rather than on code logic and semantics. One expert was observed to have used an optimal variation of the strategy to group LOCs by indentation. The expert demonstrated a strategy of grouping syntactically similar statements with the same indentation while also positioning LOCs in place and removing distractors. Both groups were also observed to have strategies on dealing with distractors.

A limitation of this study is the low number of participants. PyKinetic is still in the early ages of development and several evaluation studies will be designed and conducted using the next versions of the tutor. Based on our observations and feedback received from the study, we plan to improve PyKinetic in various aspects: problem authoring, feedback and activities included in the tutor. The buggy problems discovered in the study have since been fixed, and we have added context for problems. We also aim to develop additional types of activities for the tutor such as Parsons problems with missing keywords, erroneous examples, and predicting output.

As mentioned in Section 1, we plan to extend PyKinetic to provide instruction about optimal problem-solving strategies. For example, the system could offer instruction on specific topics the students are struggling with, or the system could refer the student to other potential sources. The system could also advise students about more effective problem-solving strategies, observed in experts. Lastly, we plan to include support for self-explanation, an important meta-cognitive skill which improves learning outcomes, and also to introduce game elements to maximize engagement (Mayer and Johnson, 2010).

References

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. In *Proc. 4th Int. Workshop on Computing Education Research* (pp. 113-124). ACM.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol Analysis: Verbal Reports as Data* (Revised Ed). Cambridge: MIT Press.

Fabic, G., Mitrovic, A., & Neshatian, K. (2016). Towards a Mobile Python Tutor: Understanding Differences in Strategies Used by Novices and Experts. In *Proc. 13th Int. Conf. on Intelligent Tutoring Systems,* Zagreb, Croatia, June 7-10, 2016. (Vol. 9684, p. 447). Springer.

Guo, P. J. (2013). Online Python tutor: embeddable web-based program visualization for CS education. In *Proc. 44th ACM Technical Symposium on Computer Science Education* (pp. 579-584). ACM.

Heift, T., & Nicholson, D. (2001). Web delivery of adaptive and interactive language tutoring. *Artificial Intelligence in Education,12(4)*, 310-325.

Helminen, J., Ihantola, P., Karavirta, V., & Malmi, L. (2012). How do students solve parsons programming problems?: an analysis of interaction traces. In *Proc. 9th International computing education researc*h conference (pp. 119-126). ACM.

Ihantola, P., & Karavirta, V. (2011). Two-dimensional parson's puzzles: The concept, tools, and first observations. *Journal of Information Technology Education, 10 (IIP),* 119-132.

Ihantola, P., Helminen, J., & Karavirta, V. (2013). How to study programming on mobile touch devices: interactive Python code exercises. In *Proc. 13th Koli Calling Int. Conf. on Computing Education Research* (pp. 51-58). ACM.

Karavirta, V., Helminen, J., & Ihantola, P. (2012). A mobile learning application for parsons problems with automatic feedback. In *Proc. 12th Koli Calling Int. Conf. Computing Education Research* (pp. 11-18). ACM.

Koedinger, K. R., Anderson, J. R., Hadley, W. H. & Mark, M.A. (1997). Intelligent Tutoring goes to school in the big city. *Artificial Intelligence in Education*, 8(1), 30-43.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist, 20(4)*, 191-206.

Lister, R., Clear, T., Bouvier, D. J., Carter, P., Eckerdal, A., Jacková, J., ... & Thompson, E. (2010). Naturally occurring data as research instrument: analyzing examination responses to study the novice programmer. *ACM SIGCSE Bulletin*, 41(4), 156-173.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008, September). Relationships between reading, tracing and writing skills in introductory programming. In *Proc. 4th Int. Workshop on Computing education research* (pp. 101-112). ACM.

Mayer, R. E., & Johnson, C. I. (2010). Adding instructional features that promote learning in a game-like environment. *Journal of Educational Computing Research, 42(3)*, 241-265.

Melis, E., Andres, E., Budenbender, J., Frischauf, A., Goduadze, G., Libbrecht, P., ... & Ullrich, C. (2001). ActiveMath: A generic and adaptive web-based learning environment. *Artificial Intelligence in Education, 12*, 385-407.

Mitrovic, A. (2003). An intelligent SQL tutor on the web. *Artificial Intelligence in Education*, 13(2-4), 173-197.

Mitrovic, A. (2012). Fifteen years of constraint-based tutors: what we have achieved and where we are going. *User Modeling and User-Adapted Interaction*, 22(1-2), 39-72.

Ohlsson, S. (1994). Constraint-based student modeling. In *Student modelling: the key to individualized knowledge-based instruction* (pp. 167-189). Springer Berlin Heidelberg.

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proc. 8th Australasian Conf. Computing Education* (pp. 157-163). Australian Computer Society, Inc..

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. In *Proc. 10th Conf. Australasian computing education-Volume 78* (pp. 155-161). Australian Computer Society.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J.A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A. & Wintersgill, M. (2005). The Andes Physics Tutoring System: Lessons Learned. *Artificial Intelligence in Education*, 15(1), 147-204.

Weber, G., & Brusilovsky, P. (2001). ELM-ART: An adaptive versatile system for Web-based instruction. *Artificial Intelligence in Education,* 12(4), 351-384.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17-22.

Woolf, B. P. (2010). *Building intelligent interactive tutors: Student-centered strategies for revolutionizing e-learning*. Morgan Kaufmann.