A Mobile Learning Application for Parsons Problems with Automatic Feedback

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

ABSTRACT
In this paper, we present a tool that facilitates the learning of programming by providing a mobile application for Parsons problems. These are small assignments where learners build programs by ordering and indenting fragments of code. Parsons problems are well-suited to the mobile context as the assignments form small chunks of learning content that individually require little time to go through and may be freely divided across multiple learning sessions. Furthermore, in response to previous analysis of students using a web environment for Parsons problems, we describe improvements to the automatic feedback given in these assignments.

Categories and Subject Descriptors
K.3.1 [Computers and Education]: Computer Uses in Education—Computer-assisted instruction (CAI), Distance learning; K.3.2 [Computers and Education]: Computer and Information Science Education—Computer science education

General Terms
Human Factors

Keywords
Parsons Puzzle, Parsons Problem, Python, Mobile Learning, mLearning

1. INTRODUCTION
Learning to program requires much practice and various exercise types and tools have been designed to facilitate this. Parsons problems are one such exercise type where the task is to reconstruct a program by selecting and reordering code fragments (e.g. lines) which are initially in a random, incorrect order [15]. Parsons problems were first developed to support rote learning of syntax but they are also used, for example, in pen-and-paper exams [5, 12] and to teach algorithmic thinking [11]. Parsons problems and educational software used to assess them automatically are discussed in Section 2.

Mobile touch screen devices are a natural platform to solve Parsons problems. The type of interaction in the assignments, dragging and dropping code fragments, is well-suited to touch screens. Also, being relatively short, Parsons problems make an ideal candidate for quick and sporadic learning sessions.

Unfortunately, automatically assessed Parsons problems on mobile devices are not well supported. Although some of the learning environments that support Parsons problems are web-based, those are difficult or impossible to use with the browsers of mobile devices. Typically, the content does not fit to the smaller screen size, touch screen actions (e.g. drag and drop) are not supported, or offline use is not possible. This leads us to our first research question “How to adapt Parsons problems to mobile devices with smaller screens and touch interaction?”

While answering this question, we introduce MobileParsons, an application that supports automatically assessed Parsons problems on iOS- and Android-based mobile devices. MobileParsons is built on top of the open source js-parsons [11] platform. Special requirements set by the mobile environment, design choices made, as well as the overall functionality of the mobile interface are explained in Section 3.1.

The automatic feedback in the initial version of js-parsons is based on linear comparison of code fragments and their indentations. Starting from the beginning of the model solution, lines are compared one-by-one and the first fragment that does not match the model solution is highlighted. This may encourage a strategy of linearly adding fragments along with repeatedly requesting feedback to test the correctness of the new fragment. Moreover, this may encourage learners to think about the program in a line-by-line manner instead of thinking about the program as a whole. Unfortunately, this goes against the way experts do and how beginners should learn to comprehend a program. Experts, for example, view programs as instantiations of abstract programming plans, schemas, such as, iterating through a container and counting items [18]. To tackle this, we have modified the feedback and it has already been utilized on large courses [8]. However, the details of the new feedback have not been described before. Even with this improved feedback, many students were found to use a trial-and-error problem solving strategy with excessive use of feedback. Moreover, we found some students wandering in the state space of the problem so that while solving the assignments they ended up with the same incorrect code repeatedly [8]. Thus, our second research question is related to improving the feedback: “How to both provide meaningful feedback on (partial) solutions to Parsons problems and discourage undesired problem solving behavior?” How we approached this is explained in Section 3.2.

2. RELATED WORK

*Corresponding author.

1http://github.com/vkaravir/js-parsons/
2.1 Mobile Learning Tools

A recent review of mobile learning research [22] showed computer science to be one of the most common disciplines applying mobile learning. Indeed, mobile systems for programming and programming education have received increasing interest both academically and commercially. Early work by Hurst et al. [9] had students view algorithm animations as videos on mobile devices. A more recent example is mJeliot [16], which integrates quiz-style questions shown for students on their mobile devices with the Jeliot program visualization tool. mJeliot is an Android application. Tillmann et al. [21] proposed that “computer programming, and thus the teaching of programming, can and should be done directly on the mobile devices themselves”. They also introduced TouchDevelop, a Windows Phone application that includes a programming environment, language, and a code editor. Scratch [14] was available for iOS for awhile but was removed due to App Store policies. The source code for it is available online3 although installing it to a device requires Apple Developer Licence. Another example is Quiz&Learn Python4, which is a mobile quiz game on Python execution. The application also includes a line-by-line visual debugger. Quiz&Learn Python is an iOS application. Finally, Codea5, formerly known as Codify, is an application that provides a Lua programming environment for iOS devices.

2.2 Parsons Problems

There exist a few interactive environments to solve and get automatic feedback from Parsons problems. Parsons and Haden, when introducing the concept, used a generic web framework for multiple choice questions (MCQ) called Hot-Potatoes [15]. There are also other online (learning) platforms that support MCQs. In general, MCQ frameworks do not suit well to solving Parsons problems reordering code fragments in such environments is clumsy [11].

Garner [7] has described a standalone application called CORT that is specifically designed to support Parsons problems. CORT extends the concept of Parsons problems by supporting a fixed context (i.e. static code around the code that is to be sorted) as a starting point for a solution. Feedback in CORT is based on learners copying-and-pasting their solutions to an interpreter to see what happens.

Two recent web-based platforms for Parsons problems are js-parsons [11] and ViLLE [17]. Both provide automatic feedback from students’ solutions and their drag-and-drop style user interface to sort code fragments is implemented in JavaScript. While js-parsons is open source, ViLLE is not. Feedback in ViLLE is a score which we assume is based on running some unit tests on the program. We could not find documentation but it appears that zero score is given to code that fails to execute due to, e.g., a syntax error. If the program can be executed, ViLLE allows learners to visualize the execution line-by-line, js-parsons brings another dimension to the traditional Parsons problems by allowing code to be indented in addition to ordering code fragments. Famously, in Python, whitespace is significant and indentation is indeed used to define blocks which makes problems in js-parsons a step more complex compared with those in the other implementations. In addition, js-parsons collects a lot of log data on how students construct their solution for possible later analysis.

2.3 Issues with Automatic Feedback

In automatic assessment systems that give immediate feedback to learners, trial-and-error behavior with frequent feedback requests is a typical problem. To tackle this, many different approaches have been used [1, 10].

The most obvious approach is to limit the number of submissions allowed (or, feedback requests). Another approach is to limit the details of feedback. In Web-CAT, for example, limiting the feedback given in the last 24 hours before deadline eliminated the rush of last hour before the deadline [6]. Yet another way to limit feedback is used in Marmoset [20] that divides feedback into two phases: immediate feedback includes the results of public tests, whereas results of secret tests are given only after the deadline. QuizPACK [4] and TRAKLA2 [13] use quite a different approach of randomizing the assignment for each try. Finally, requiring some minimum time between successive feedback requests can be used to prevent trial-and-error behavior. According to Ala-Mutha [1], Cellidh has this feature.

In the context of Parsons puzzles, our previous analysis [8] revealed the problems with feedback usage already mentioned in Section 1: students who had episodes of no progress in the form of loops in the solution path and students who requested feedback excessively. Figures 1 and 2 show examples of these behaviors in students’ solution paths (taken from the analysis tool presented in [8]). In the figures, each node represents a state of the code fragments and indentations, and edges correspond to operations that change the code and thus cause a transition from a state to another. Block denotes the end, the target node of an edge. The graphs give a lot of information about the progress of a student, such as, the time taken to do the operation (numbers on the edges), type of operation (color of the edges), and correctness of the state (border color of the nodes). What is important for this paper, is the shape of the graph in Figure 1 with loops, and the frequency of feedback requests (nodes with black fill color) in Figure 2. In this paper, we present a new mobile application with improvements to the feedback in the js-parsons environment that are aimed to address these issues.

3. DESIGN AND IMPLEMENTATION

In this section, we describe the design and implementation of MobileParsons. First, in Section 3.1 we explain how the mobile user interface was designed and how we were able to utilize js-parsons in our work. After that, in Section 3.2, we describe how the feedback of js-parsons has been improved. Much of the feedback improvements have been pushed to the js-parsons codebase so that they are now available on both js-parsons and MobileParsons.

Indeed, we decided to build MobileParsons on top of js-parsons. Figure 3 is a screenshot from a Parsons problem in js-parsons. Initially, all code fragments are on the left and a student is asked to construct the solution to the area on the right. All fragments are draggable and it is also possible to place a fragment between two consecutive fragments, in
3.1 Mobile Application

We redesigned the whole user interface for the mobile environment. While the desktop version was, after a small fix to handle touch events, functional on smartphones, it was not really usable. The lines of code were too small to be able to reliably move the lines wanted. To make the interaction easier while still having the required functionality, we took the following steps to save space (that is, pixels) by redesigning the components on the screen.

• Main Controls: The main controls of the Parsons problems are to get feedback, move to the next assignment, and move back to the collection listing. All these take a significant amount of space in the desktop version shown in Figure 3. We eliminated the page title completely, as we did with the next exercise button. In the desktop version, the next exercise button appears next to the feedback button once the problem is
solved. For the mobile version, closing the feedback of a correctly solved problem automatically moves to the next assignment. For requesting the feedback, there is a small button in the top-right corner. On the top-left corner, there is a small button to move back to the home page. Note, that while both of these buttons are small, the area where they respond to touch is significantly larger than the visual appearance.

• **Assignment:** Depending on the task, the assignment can take a lot of space. To save space, we moved the assignment in its own tab that can be toggled visible/invisible with a click of the tab. This tab is positioned on the left and is initially visible when starting a new puzzle. Figure 4 shows an example of the assignment tab.

• **Interface instructions:** In the desktop version, both the given code and the area where the solution should be constructed had labels above them indicating their role. We consider only the instructions on where the solution should be constructed as important for the assignment. These instructions are now embedded in the target area and are visible until the first line is dragged there.

• **Lines of code:** Lines of code are the target of the main interaction in the assignments, thus we did not significantly decrease their size. We merely changed the font to a slightly smaller size on small screens, as well as reduced padding. The most significant change is switching the position of the input code and target area depending on the device orientation. When the device is in landscape orientation, the code areas are side by side (see Figure 4). In portrait orientation, the code areas are on top of each other (see Figure 5). This arrangement tries to minimize the need for scrolling when dragging the lines.

It should also be noted that the user interface changes the size of the components depending on the screen size. Thus, the application is usable on devices ranging from small smartphones to larger tablet devices.

In addition to the redesign of the user interface components, the additions to the feedback introduced in Section 3.2 needed to be included in the UI. In general, feedback was shown in a rather concise way already with the use of colors in js-parsons, so we did not change this behavior. Requesting feedback still shows a pop-up window with textual feedback and highlights the incorrect parts in the code.

As a new feature, if a student uses feedback excessively, it is disabled for a certain duration. The circumstances for doing this are described later in Section 3.2.3, here we only focus on the user interface. When feedback is disabled, the feedback button turns gray, as does the orange border at the bottom of the header. The border becomes a progress bar, turning back to orange from left to right. Once the whole border is orange, the feedback button turns orange and feedback can yet again be requested. Since this is a rather subtle indicator, the feedback message shown will also point out that the feedback has been disabled and will be available once the border is orange again.

As another new feature compared with js-parsons, when the student has already requested feedback, a Last Feedback tab is shown which shows the feedback the learner got the last time. Furthermore, whenever the learner ends up with code that has been seen before, this tab changes to a Hint tab to inform that they are going in circles. This feature is further explained in Section 3.2.2. Opening the tab shows the current state as well as the state where the student continued the last time (see Figure 6). Also, the line modified last time is highlighted. Like with the draggable code fragments, the code areas are positioned next to each other in landscape mode and on top of each other in portrait mode.

### 3.1.1 Technology

Like the desktop version, the mobile environment is implemented using web technologies. We rely heavily on established
Figure 6: The hint shown when the solution path forms a loop, i.e., the learner is going in circles in terms of the constructed code. Blue highlights illustrate the change the student made to the code when they last had it. Feedback pointing out errors is given for both states of the code (red highlights).

JavaScript libraries such as jQuery\(^8\) and jQueryUI\(^9\) to build on top of. Both of these libraries were already used by js-parsons. Adapting the layout for different screen sizes and device orientations is done using CSS media queries. To create native applications for mobile devices, we used PhoneGap\(^10\) which provides access to device’s APIs for HTML applications and, more importantly, makes it possible to wrap HTML applications as native applications. While PhoneGap supports many different mobile platforms, we have thus far tested and used the application on iOS and Android phones and tablets. We chose these platforms partly due to their popularity and partly due to having access to the devices and development tools for these platforms.

Exercise data, i.e. the assignment collections, are fetched from the server. The collection data is refreshed on user request. This data is also stored locally on the device so that assignments can be solved offline. This feature is implemented using HTML5’s local storage.

3.2 Automatic Feedback

In this section, we describe how we have improved the feedback in both js-parsons and MobileParsons. Improvements described in Section 3.2.1 have already been used with students (see [8]) while improvements presented in Sections 3.2.2 and 3.2.3 are based on problems discovered in subsequent analysis of students’ solutions to these types of assignments [8]: poorly progressing solutions where there were episodes of no progress in the form of loops in the solution path and excessive use of feedback. To address these issues, we followed the design guidelines of Baker [2]. His conditions were conceived for adapting an educational system to deal with students who game the system. Modifying them to fit our case, the conditions for the improvements were: 1) the design must aid learners who exhibit the aforementioned undesirable behavior and 2) the design should have minimal effect on those who do not.

3.2.1 Feedback from Partial Solutions

The primary principle in designing the new type of feedback was that learners should be able to construct their answers to Parsons problems in any order and still get meaningful feedback even from partial solutions. For example, a student solving an assignment whose solution is a recursive program may start by constructing the termination condition for the recursion, and continue by adding some initialization routines to the beginning. Both of these may be in a sense correct but there are still lines missing from the beginning, middle, and end. On the other hand, some code fragments might have been misplaced while most could be in order. The question is how to provide guidance with feedback on such partial solutions by informing which parts of the solution are correct and which are not. In Parsons problems, we have the advantage of knowing the solution. Thus, we can pinpoint where the error lies even in partial solutions that are not executable code. Therefore, just like these Parsons problems are about ordering and indenting code fragments, we preferred to provide feedback in terms of how one could fix their partial solution by reordering and reindenting the code fragments.

The feedback algorithm we implemented is based on longest common subsequences (LCS) [3]. To give feedback about the order of the lines, we search for a longest common subsequence shared by the model and the student solution that has the greatest number of consecutive code fragments in the student’s solution. By labeling the fragments in the model solution with increasing numbers from top to bottom and using these same identifiers for the respective fragments in the student’s solution, the problem can be reduced to finding the longest increasing subsequences. After calculating the LCS, we highlight the lines from the student’s solution that do not belong to the LCS. Intuitively, the highlighted lines are the smallest set of fragments in the wrong positions relative to the others. The ordering of the code can be corrected by moving only the lines that are highlighted. Another important aspect of the algorithm is selecting the LCS with most consecutive code fragments in the student’s solution to ensure that students are not advised to break correct snippets containing multiple code fragments. An example of this is shown on the right in Figure 7 (compare this with the feedback for the same solution in the old version shown in Figure 3). There are two possible line combinations to highlight as incorrect: first and second or the ones shown in the figure. The feedback in the figure is preferred since the
second line follows immediately after the first line in the correct solution. When all the lines are in correct order, we give feedback about the indentation similarly to what js-parsons did before. That is, when the correct code fragments have been selected and are ordered correctly in the solution, the start of the first fragment with incorrect indentation, if any, is highlighted.

![Figure 7: An example of the feedback on an incorrect ordering of code fragments. On the left is the expected correct solution. On the right, an incorrect partial ordering is shown as well as the feedback that the student would get from that partial solution.](image)

3.2.2 Poorly Progressing Solutions

To help students who embarked on aimless exploration resulting in loops (revisiting previous states of the solution along the solution path), we implemented functionality to recognize such behavior. When this occurs, a tab with the label “Hint” appears that will give students information that they have visited the same state, i.e. had the same exact code as their (partial) solution, before as well as show where they continued with their solution the last time. Feedback on both the current state they are in and the state they continued the last time is given. Using the feedback from these two different but closely-related states we hope the student then to be better able to reflect on their error in the partial solution. Indeed, there is a gentle suggestion for the student to really stop and think at this point. An example of this kind of a hint tab is shown in Figure 6. We considered forcing the hint to students instead of placing it behind a tab, but decided against it based on our design guidelines. In addition, we do not want this functionality to become irritating to the students. Furthermore, we only make the hint available if both the current state and the state they moved to the last time have more than one code fragment. We also omit the hint when the student makes only one change before immediately changing the code back to to its previous state.

Had this hint functionality been available for the students in our earlier study of how js-parsons is used [8], 15–23% of students’ solutions would have received it. The results for the three assignments analyzed are presented in Table 1. The table shows that quite a few students could possibly have benefited from the newly-implemented additional guidance.

3.2.3 Frequent Use of Feedback

To encourage students to think about the feedback rather than using it for trial-and-error, we decided to deactivate the feedback button if the frequency of feedback requests gets too high. This means that whenever a user requests feedback, we investigate what he has done during the previous 45 seconds, and

- If there is another feedback request during the previous 5 seconds, feedback is deactivated for 15 seconds.
- If there are two other feedback request during the previous 20 seconds, feedback is deactivated for 15 seconds.
- If there are three other feedback request during the previous 45 seconds, feedback is deactivated for 15 seconds.

Time penalties of all rules that apply are summed together. This means that the feedback functionality can be deactivated for 15, 30 or 45 seconds. This is done in order to give greater penalty for several rapid successive feedback requests compared with a single offence. No other history information than what has been done during the previous 45 seconds is used. This is to allow students to correct their trial-and-error behavior and not to get penalized after that.

We used the previously collected data and analysis tools [8] to empirically design and test the penalty conditions. We tested different values and decided the limits after reviewing the solution graphs (such as those in Figures 1 and 2) of both penalized and non-penalized solutions. We tried to optimize the limits so that those which, based on our analysis of individual solutions, looked like trial-and-error would get penalized and others would not. Each solution was only examined until the first time penalty since after that it is impossible to speculate how the student would have continued. Table 2 summarizes the penalties imposed for the three assignments we used as our test data. The last column in the table gives statistics on how many times the students would have been eventually penalized requested feedback before the penalty would have been imposed. The median for this was four, while the medians of feedback requests of all solutions in P3, P4, and P5 varied between 1 and 3. Indeed, while the majority of the students would never get penalized, those who do would do so soon after they start to use feedback.

### Table 1: Percentages of students’ solutions in the previously collected data that would have, at least once, received the hint related to looping behavior.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solutions</th>
<th>Solutions with hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>140</td>
<td>23 (16.4%)</td>
</tr>
<tr>
<td>P4</td>
<td>137</td>
<td>32 (23.4%)</td>
</tr>
<tr>
<td>P5</td>
<td>136</td>
<td>20 (14.7%)</td>
</tr>
</tbody>
</table>

4. DISCUSSION

Parsons problems provide a tool that can be used with several possible learning goals in mind. Lower-level concepts we can focus on are the syntax and constructs of a programming language. Originally, Parsons problems were envisioned as a tool for allowing the learning of syntax with many repetitions. This can be accomplished by accompanying code fragments...
Table 2: The results of simulating the feedback disabling formula on the collected data. MAD stands for median absolute deviation. Each solution was examined only until the first penalty if any and thus each either got a single type of penalty or none. The numbers in the last column also include the feedback request that was denied.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solutions penalized</th>
<th>Penalties imposed</th>
<th>Feedback requests before penalty (med/mad)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>15s 30s 45s</td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td>37 (26.4%)</td>
<td>23 13 1</td>
<td>4/1</td>
</tr>
<tr>
<td>P4</td>
<td>40 (29.2%)</td>
<td>28 10 2</td>
<td>4/1</td>
</tr>
<tr>
<td>P5</td>
<td>30 (22.1%)</td>
<td>20 8 2</td>
<td>3.5/0.5</td>
</tr>
</tbody>
</table>

with distractors\(^{12}\) that have syntax errors. Especially this use case, where assignments may be short but many in number suits well to mobile learning. On the other hand, an approach to familiarize learners with different language constructs is to design assignments that use the same constructs in various different programs and contexts. The key is to design the programs in such a manner that they will highlight both the primary function but also the edge cases in using the constructs.

Possible higher-level learning goals in designing Parsons problems can revolve around program design and logic. Parsons problems can be used to make learners read and trace many different types of programming solutions to specific problems in different domains. This provides a medium for familiarizing learners with common prototypical programming patterns, schemas and programming plans, such as, how a counter or an accumulator can be implemented. This idea was a driving force in the design of the CORT system [7].

On the other hand, Parsons problems encourage learners to exercise their skills in tracing code without running the code. This is especially true for the mobile use case because there typically will not be any runtime environment readily available and instead learners will need to rely on their understanding of how the code functions and their existing mental models of the concepts and constructs involved.

Finally, we may target specific known misconceptions, such as those listed by Sorva [19], by designing distractors that allow alternative solutions which are a likely manifestation of these. Indeed, we may try to induce cognitive conflict and foster learning by isolating specific problems with distractors that, when used, force the learner to stop, think, and try to resolve the flaw in their understanding. For this purpose, a future improvement would be to incorporate some additional learning material in order to support this conflict resolution.

The new features in MobileParsons aimed to address aimless exploration and frequent use of feedback have not been tested or evaluated with students. We see a future evaluation of these aspects important. However, we believe we have progressed in the right direction. The problems addressed have been identified when analyzing real student solutions. Furthermore, the same code has been used when empirically designing the details of the features. As a result, both the additional hints on poorly progressing solutions and the time penalty on excessive use of feedback could have a positive impact on how students solve these assignments.

From a technical perspective, the mobile application can be integrated with the existing server implementation via an API in order to allow distributing and creating new collections of Parsons problems in that environment. We see this possibility as a potential fertile ground for collaborative creation, use, and evaluation of Parsons problems.

With regard to the mobile implementation, making the same code work on both iOS and Android platforms was surprisingly straightforward. However, as is typical for cross-platform development, making the application work smoothly on both platforms proved to be tricky. Especially some incorrect ghost events on Android 2.3 and failing HTTP requests in Android 4 caused significant problems. Overall, the experience with PhoneGap was encouraging enough that we have considered the possibility of making a Windows Phone version in the near future.

Finally, another aspect of developing native applications for mobile platforms, especially for iOS, is the difficulty of distributing such applications. All apps need to go through the Apple App Store review process to be available for students. On Android, there is no such limitation. Our goal is to submit the application to both Apple App Store and Google Play as a free application. We see going through the official distribution channels as an important step in distributing educational mobile apps. At the time of writing, however, this has not yet been done.

5. CONCLUSIONS

In this paper, we have introduced a new mobile learning application for Parsons problems called MobileParsons. In addition, we have described significant improvements to the existing js-parsons system to address issues found in previous analysis of students’ problem solving process in these assignments. The system now provides more guidance with advanced feedback on the order of lines, subtle feedback when students return to a state in the solution they have visited before, as well as impose a time penalty for frequent use of feedback. As future work, we would find interesting to experiment with different approaches of making MobileParsons more engaging and addictive, such as, assigning scores to solutions, rewarding learners for achievements such as completing many assignments or using little time to complete assignments, or facilitating social interaction and competition between learners.

6. REFERENCES


