Journal of Information Technology Education: Innovations in Practice

Volume 10, 2011

Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations

Petri Ihantola and Ville Karavirta
Department of Computer Science and Engineering, Aalto University, Helsinki, Finland

petri.ihantola@aalto.fi; ville.karavirta@aalto.fi

Executive Summary

Parson's programming puzzles are a family of code construction assignments where lines of code are given, and the task is to form the solution by sorting and possibly selecting the correct code lines. We introduce a novel family of Parson's puzzles where the lines of code need to be sorted in two dimensions. The vertical dimension is used to order the lines, whereas the horizontal dimension is used to change control flow and code blocks based on indentation as in Python. Python blocks have no explicit begin/end statements or curly braces to mark where the block starts or stops. Instead, indentation is used to define starts and stops of blocks and functions.

In addition, we introduce tools supporting two-dimensional Parson's puzzles: (1) MIT licensed JavaScript widget to embed our puzzles to any HTML, and (2) server to create, share, and solve puzzles.

We have observed how experienced programmers solve our puzzles. Such users often start by dragging the method signature to the beginning and continue by defining majority of the control flow (i.e., loop statements, assignments, conditional statements). Only after these are done, details, including initialization of variables and handling of corner cases, are dragged to correct positions in the middle of the previously structured code. This shows that even experts are not able to solve puzzles linearly, i.e., line by line, starting from the first. Thus, user interfaces (UIs) should minimize the work needed to insert a line between two adjacent lines of existing code. In some of the existing Parson's Puzzle UIs this is not the case.

Another observation we made is that too often users don't ask or use automated feedback. Why this happens needs further investigations. Perhaps experienced users are too proud to ask a tool to help them (especially when being observed), or perhaps users don't recognize when they are stuck and should ask for help. Providing constant feedback is one way to tackle this problem. However, the obvious downside of the constant feedback is that solving an exercise can become trial-and-error repetition.

Keywords: programming, automated feedback, Parson's puzzles, open source, problem solving strategies

Introduction

It is commonly agreed that students' active participation and exercises are essential for learning programming. The problem, especially on large courses, is that teachers rarely have enough time to

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

Editor: Janice Whatley

give the high quality feedback they would like to. This is where automatic assessment can help. By automating the workload in assessment, the teacher can focus on improving other aspects of the course (Carter et al., 2003).

Being able to program includes a wide spectrum of skills. For example, ITiCSE 2004 working group (Lister et al., 2004) stated that being able to read and trace code is a precursor skill to writing code of a similar complexity. Since then, the BRACElet group has investigated the other intermediate levels of building programming knowledge. They have found that the ability to explain a program in a way that demonstrates the ability of seeing the forest for the trees – relational answers in the terms of the SOLO taxonomy (Biggs & Collis, 1982) – is one such level (Lister, Simon, Thompson, Whalley, & Prasad, 2006). Moreover, Lopez, Whalley, Robbins, and Lister (2008) demonstrated that there is a hierarchy of skills from reading to explaining and from explaining to writing. A later follow-up study by Lister et al. (2009) made the same conclusions.

Programming related assignments can be designed to measure different skills. For example, *"What is the value of a given variable after the following code is executed?"* is a simple tracing question, whereas "*Explain in plain English what the program does.*" requires higher skills. "*Write a program that sorts a list"* is an example of a code construction assignment.

Parson's puzzles are simplified code construction assignments where the lines of code are given in the wrong order and the task is to sort and possibly select the correct lines (Parsons & Haden, 2006). Originally, Parson's puzzles were developed to provide an engaging learning environment with immediate feedback. Parson's puzzles are widely used and there are tools supporting them.

Anecdotally, in the original article these puzzles were called Parson's puzzles. This is slightly confusing as the first author is Parsons, not Parson. Since then the name of these puzzles has varied between Parsons' puzzles, Parson's puzzles, Parsons puzzles/problems and simply Parsons. We follow the typing from the title of the original paper.

Automatic assessment of Parson's puzzles is straightforward as it can be done without executing the code. In addition to online learning environments, Parson's puzzles can be used in traditional paper exams where coding exercises (i.e., programming with a pen and a paper) are often problematic. Interestingly, in the context of paper exams, points from Parson's puzzles correlate well with open ended code writing question (Denny, Luxton-Reilly, & Simon, 2008). Moreover, in the same study, points from neither of these question types correlated with points from tracing exercises. However, in another setup, Lopez et al. (2008) found Parson's puzzles to be lower level than tracing exercises including loop constructs. Lopez et al. speculate this could be due to different difficulty levels or complexities of tracing and Parson's puzzle exercises in their study.

In this paper, we introduce a new family of Parson's puzzles inspired by the Python programming language. We also introduce an open source tool to embed such puzzles on web pages. In our tools we have tried to address some of the reported usability issues of the other systems also supporting Parson's puzzles. In addition, we provide a website where teachers can browse existing puzzles, create new puzzles, and create collections with several puzzles for their students. Finally, we observed experts and report how they solve complex Parson's puzzles (e.g., insertion sort as a puzzle). For example, if the exercise is too complex to be solved linearly, experts often create the control flow first and add initializations of variables after that.

The rest of this paper is organized as follows. In the next section, we introduce different types of Parson's puzzles and existing tools supporting them. The third section presents our two-dimensional variant and the tools we have implemented. In the fourth section, we describe our observations of experts, related to the problem solving strategies. The next section discusses the observations, how to design puzzles, and why the current assessment approach of Parson's problems is not always sufficient. Finally, we conclude our work.

Parson's Puzzles

Different Types of Puzzles

We believe that the position of Parson's puzzles in the hierarchy of programming related skills can vary – Parson's puzzles are neither strictly close to writing nor strictly close to reading/tracing questions. There are many variants of Parson's puzzles and different variants can measure and teach different skills. Moreover, as also speculated by Lopez et al. (2008), the complexity of the puzzles may affect the level of skills needed to solve the puzzles. Based on the previous research, we identified the following possibilities to construct Parson's puzzles:

- **Extra lines** (i.e., distractors) can be added to make a puzzle more challenging (Parsons & Haden, 2006). When distractors are used, there are two approaches to construct an initial state. First approach is to randomly order all the lines, including distractors. Another approach is to group the distractors with the correct alternative and always keep them next to each other in the initial random order. In this approach, distractors can be designed to separate problems in problem solving and in syntax. According to Denny et al. (2008), grouping reduces cognitive load not relevant to problem solving or to programming.
- **User-created blocks** are supported by letting users insert curly braces or indent the code. This gives a lot more freedom and, if combined with distractors without line grouping, exercises can get too difficult (Denny et al., 2008).
- **Context** provides a fixed code around the code to be sorted. It allows larger, and often more concrete, examples to be shown to students (Garner, 2007).

All these can be combined to create different kinds of assignments. Some can test problem solving whereas others can be targeted to syntactical problems.

Tools

The original Parson's problems (Parsons & Haden, 2006) were created using a generic drag-and-drop exercise framework called Hot Potatoes (see http://hotpot.uvic.ca/). Exercises created with the tool can be exported to HTML+JavaScript pages. An example of such exercise is presented in Figure 1. Exercises are solved by dragging lines from right to left. When feedback is requested, lines in (absolutely) correct positions are highlighted. One problem of this UI is that inserting a line between two existing lines is cumbersome. Student may need to move all lines after the insertion point to create a *free slot* where the new line can be inserted.

ViLLE (Rajala, Laakso, Kaila, & Salakoski, 2007) is a Java application/applet originally developed for program visualization. Recent versions include Parson's puzzles, which allow context to be created around the editable code. Such an exercise is illustrated in Figure 2. Distractors are not supported. The feedback is an error message in case the resulting code does not compile or a number of points in case it does. In this case, the student can also step through a visualization of the execution of his/her solution.

CORT (Garner, 2007) has been used with Visual Basic programs so that students move lines from left to a part-complete solution on the right. Moving the lines is done by selecting a line and clicking arrow buttons to move it left or right. To get feedback, student can copy the code into Visual Basic interpreter and execute the code. CORT supports both distractors and context.

Binary Tree Postorder Traversal

Order the codelines by dragging and dropping from right to left.

Check

1	def traverse_postorder(tree_node):
2	visit(tree_node)
3	traverse_postorder(tree_node.right)
4	if tree_node is not None:
5	traverse_postorder(tree_node.left)

Figure 1: A Parson's problem in Hot Potatoes.

Sort code lines

```
def main():
    a = 1
    b = -1
    a = b
    tmp = a
    b = tmp
```

Exercise description

Order the codelines so that it swaps contents of variables a and b.

Figure 2: A code sorting exercise in ViLLE.

Of these tools, Hot Potatoes is freeware, ViLLE is freely available for non-commercial purposes, and CORT does not seem to be publicly available. The source code is not available for any of these tools. As we will see in the next section, this presented a big problem for us.

Two-Dimensional Parson's Puzzles

We introduce a new family of Parson's puzzles inspired by the Python programming language. Python uses neither curly braces nor begin/end pairs to group lines. Instead, code blocks are defined by their indentation. Listings 1 and 2 illustrate this. Listing 1 is a program that returns the largest, non-negative value of a list. Listing 2 is almost the same, but because of the different indentation of the return statement, return is always executed at the end of the first iteration and the first value of the list is returned.

We propose a two-dimensional variant of Parson's puzzles where lines of code are not only sorted but also placed on a two-dimensional surface. The vertical dimension is used for ordering the code like in traditional Parson's puzzles. The horizontal dimension is used to define code blocks based on indentation – just like in Python.

```python
def find_max(L):
    max = 0;
    for item in L:
        if item > max:
            max = item
    return max
```

Listing 1: Python example

```python
def find_max(L):
    max = 0;
    for item in L:
        if item > max:
            max = item
        return max
```

Listing 2: Another Python example

Initially, we wanted to extend one of the existing tools to support the proposed variant of puzzles. However, since the source code for none of them was available, we were unable to do so. Furthermore, we felt the user interfaces of the existing systems were some- what clumsy. The original Parsons' article identified several needs in puzzles built with Hot Potatoes (Parsons & Haden, 2006). For example, the mechanism of drag-and-drop is cumbersome. To insert a new line between two consecutive lines, the user needs to move all the lines after the insertion position to create a free slot.

We trusted we could improve the user experience in Parson's puzzles and had no choice but to implement yet another tool. The next subsections introduce our tool as well as our online environment for creating, sharing, and solving two-dimensional Parson's puzzles.

Our Parson's Tool

We designed JSParsons based on student feedback reported in the original Parson's puzzle article and our own experience on using Hot Potatoes and ViLLE. Our tool is open source under the MIT license (available at https://github.com/vkaravir/js-parsons). JSParsons is a JavaScript widget. This allows puzzles to be embedded into any HTML document. The most novel feature of JSParsons is the support for two-dimensional drag-and-drop of the code lines. To make creating matching indentations easier, an adjustable grid is used to mark the allowed positions of code lines.

JSParsons supports two visualization modes. In the basic mode, shown in Figure 3, lines of code are sorted and indented in one area. Distractors are not supported in this mode. In left to right mode (Figure 4), distractors are supported and there are two areas for code lines similar to the original Parson's puzzles. In this mode, lines need to be dragged to the solution area on the right and can be inserted between any two lines. A line can be removed from the solution by dragging it back to the left area.

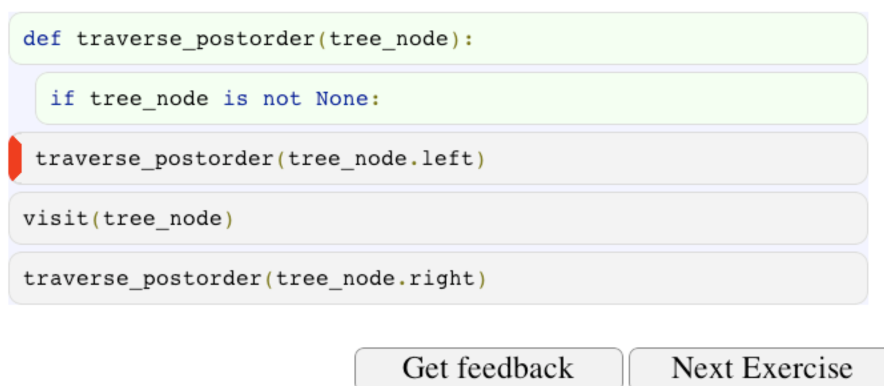

Figure 3: Example of a two-dimensional Parson's puzzle in basic mode.

Feedback is given to student on request. There are various types of feedback (Figure 3 shows examples of the two first types):

- Lines in correct/incorrect position are colored green/red.
- Lines in correct position but incorrectly indented are highlighted with a red border on the left.
- In left to right mode, the background of the solution is colored green/red if the solution has correct/incorrect number of lines.

Drag from here

```
traverse_inorder(tree_node)
if tree_node is not None:
traverse_inorder(tree_node.right)
```

Construct your solution here

```
def traverse_inorder(tree_node):
  if tree_node is None:
    traverse_inorder(tree_node.left)
    visit(tree_node)
```

Get feedback | Next Exercise

Figure 4: An exercise in our system with distractors.

Creating new exercises with the widget requires a few lines of JavaScript: specifying the indented lines of code of the model solutions, distractors, and optional arguments (e.g., visualization mode). Another option for creating puzzles is to use our online environment and the exercise editor described next.

Online Environment for Parson's Puzzles

In addition to the JavaScript tool, we have a website (http://parsonspuzzles.com) where puzzles can be solved and where teachers can create puzzles online. Main features of this site are:

- Teachers can browse existing puzzles and create collections with several puzzles. Each collection can be accessed with a unique URL a teacher can pass to his/her students. Furthermore, public collections appear in a list on the main page. This allows students to solve the puzzles and other teachers to find suitable collections.
- Teachers can create new puzzles by using an online editor (see Figure 5). All puzzles are licensed under creative commons, which authors need to agree.

New Puzzle

Quick instructions: Every line you input into the textarea below will be one line in the puzzle. Indentation of all the lines needs to be valid according to python indentation rules.

Distractors, i.e. lines that are not part of valid solution, are marked by adding `#distractor` to the end of the line. Combined lines can be created by adding \n into the string. Before saving your puzzle, Click *View puzzle* to show what the puzzle would look like.

Title: Swap Variables

Description: The solution should swap contents of variables a and b

Code:

```
1 tmp = a
2 a = b
3 b = tmp
4 b = a #distractor
```

Position: Ln 4, Ch 18 Total: Ln 4, Ch 36

Save | Cancel | View Puzzle

Figure 5: Editor for creating two-dimensional Parson's puzzles

Another novel feature is the ability to record how puzzles are solved. We hope this data to be useful in further research into how Parson's puzzles are solved.

Problem Solving Strategies

The goal of our preliminary analysis described here is to form an hypothesis on how experienced users solve algorithmic, two-dimensional Parson's puzzles. This is why we choose to follow qualitative research approach in this study. Follow-up quantitative studies are needed to verify if most experts are actually following the approached we identified. In addition, more research is needed to understand if experts' strategies related to Parson's puzzles differ from what (novice) students do.

Research Method

To collect data about the problem solving strategies, we created a collection of ten algorithmic Parson's puzzles (see the Appendix). The exercises included two simple tasks to introduce the widget, four exercises on tree traversal algorithms, and three on sorting algorithms.

We observed four senior teachers and teaching assistants solving the problems. All the participants were familiar with the data structures and algorithms in the questions. Solving the exercises took them 20–30 minutes. We asked the participants to speak out loud what they were thinking while solving the puzzles. At the end, we asked their comments. We wanted to keep the atmosphere and discussion open so we did not record voice or video. Instead, we were both observing and taking notes. We were afraid some of our participants would have felt stressed about their performance if they were recorded. In addition, to improve validity of our results, we discussed our observations afterwards with the interviewees. Furthermore, the system recorded the solution sequences, so we could trace them when analyzing the data.

The Strategy

Most of our puzzles were algorithmic. That is, a name of an algorithm was given and the task was to sort the lines of this algorithm. Many of the algorithmic puzzles were about sorting algorithms. All the participants followed the following strategy of five steps when they solved sorting algorithms:

1. Find the function signature. This was actually the first step in all sorting exercises.
2. Find two loop statements and add them after the signature.
3. Check that the loops are in correct order, change if needed, and indent the first three lines correctly.
4. Take the conditional *if*-statements and the lines where variables are initialized and insert them to correct positions. This was typically the first time feedback was requested to ensure the solution was going to the right direction.
5. Add the remaining lines and check the indentation.

The rest of the puzzles were more trivial. Especially in the Hello World and Swap puzzles the participants simply took all the lines in correct order. Participants clearly had the full solution in their minds before they started to construct the solution.

Other Observations

Participants were allowed to ask feedback from the tool while they were solving the puzzles. Feedback from an unfinished solution might be valuable if they believed that the first few lines of

their solution were already correct. Still, participants rarely requested the feedback before they felt their solution was ready. Clearly, requesting feedback more often would have helped them solve the puzzles. One of the interview comments we got was that "*I would likely use more of feedback and trial-and-error method to solve the exercises if no one was monitoring and taking notes.*"

Two of the participants commented that because they already knew the algorithms, they learned Python (which they were not familiar with). However, one of the participants commented, "*This is more difficult than writing code when the expected solution does not match one's own mental model of the algorithm*" (we had two versions of selection sort: traditional taught on our DSA course and a more pythonic version of list sorting).

Discussion

In this section, we discuss strategies of experts to solve complex Parson's puzzles built with our widget. In addition, we report the problems we faced when creating new puzzles.

Solving Puzzles

According to Lister et al. (2004) and McCracken et al. (2001), many novice programmers have problems in reading and in writing programs. One explanation is that novice programmers miss programming schemas or plans (Detienne, 1990; Soloway & Ehrlich 1986). For example, experienced programmers know how to iterate over an array to find the best (e.g., smallest) element. When reading programs, experts can recognize those patterns immediately. They are also able to apply and combine them when writing programs.

Muller (2005), among others, has suggested that algorithmic patterns (i.e., plans) should be explicitly taught to novice programmers. We hope that, with assignments similar to what we have in the Appendix, Parson's puzzles can teach these patterns.

The strategy described in the next sub-section can be interpreted as a schema or a template applied by experts. A loop, an inner loop, and a comparison are often needed for sorting. This is why users select them first. After these steps, users fill in the rest, which is also when experts typically start thinking.

Even experts did not solve the problems linearly (i.e., where lines are moved into the correct position in their textual order; the 1st line, the 2nd line, etc). We suspect that students would also not solve the problems linearly. Thus, it is important that a line can be easily added between any two lines – a feature not supported by Hot Potatoes.

Our expert participants used the feedback of the tool less than we anticipated. However, it is not clear how much from this behavior was because of us observing. Interestingly, Isohanni & Knobelsdorf (2010) found their students were also not using the feedback in their tool. This was especially true when students were struggling and would have needed the help. The lesson we can learn from this is that some feedback on syntactic indentation errors could be shown continuously without the student needing to request it. The continuous feedback is also supported by the findings of Kirschner, Sweller, and Clark (2006) who conclude that novices should be given more guidance when they do not have sufficient prior knowledge.

One concern raised when interviewing the participants was that while the puzzles would be helpful for some students, avoiding trial-and-error and the use of Wikipedia to find the solutions might be difficult – especially if these are used for grading purposes. This is a common problem in education nowadays, one where no perfect solution exists. As a solution, we suggest that when designing the puzzles (especially when they are not algorithmic like ours), the problem should be something specific enough to not be already available online.

Designing Puzzles

Ambiguous solutions are perhaps the biggest problem of automatic assessment of complex Parson's puzzles. Ambiguity means that the functionality of two different programs can be the same. In the QuickSort in Listing 3, for example, lines 5–7 are interchangeable. This is a problem if the assessment of Parson's puzzles is based on the correct order of the lines. To avoid this, in JSParsons draggable elements larger than a single program line can be created. In rare cases, this kind of grouping approach can lead into too large elements trivializing the whole exercise. In addition, the author of an exercise needs to decide when such elements are needed.

```
def qsort(L):
    if len(L) <= 1:
        return L
    pivot = L[0]
    less = [x for x in L if x < pivot]
    equal = [x for x in L if x == pivot]
    greater = [x for x in L if x > pivot]
    return qsort(less) + equal + qsort(greater)
```

Listing 3: QuickSort in Python (quoted from http://www.codecodex.com/wiki/Quick sort)

Another problem we faced, not solvable by the grouping approach, is the logically different solution strategies related to the use of distractors. These alternative solution strategies can be divided between the following three categories:

- Distractors can be placed in a position where they never get executed. Multiple distractors and at least one with a conditional statement similar to the correct one led often to this problem. In other words, distractors were used to create conditional blocks that are never executed.
- Distractors may also have no real effect even when they get executed. For example, distractors modifying variables that are not used after a certain point in the program are problematic.
- Meaningful but still different solution strategies are also possible. For example, adding both "return" and "if tree node is None:" as distractors to the example of Figure 3 creates at least two valid solution strategies demonstrated in Listings 4 and 5.

The first two categories are clearly bad programming and therefore we argue that automatic assessment should mark those as incorrect. However, the challenge is to identify these in order to give good feedback. Feedback should tell that, although the functionality is correct, there is still something wrong in the code.

It is not clear how to deal with the last category. One argument against accepting these strategies is that the model solution was designed to be idiomatic, whereas the alternative is not.

We propose that Parson's puzzles should be constructed in an environment that tests all the possible combinations, alerts the author about alternative solutions, and leaves it for the author to decide which of the solutions should be accepted. Functionally identical variants could be identified based on unit tests provided by the author.

```
def traverse_postorder(tree_node):
    if tree_node is None:
        return
    traverse_postorder(tree_node.left)
    traverse_postorder(tree_node.right)
    visit(tree_node)
```

Listing 4: A correct solutions to the post order traversal

```
def traverse_postorder(tree_node):
    if tree_node is not None:
        traverse_postorder(tree_node.left)
        traverse_postorder(tree_node.right)
        visit(tree_node)
```

Listing 5: Another correct solutions to the post order traversal

Conclusions and Future Research

In this article, we have:

- Described a new two-dimensional subcategory of Parson's puzzles.
- Reported our initial observations on how experts solve complex, two-dimensional Parson's puzzles. For example, users try to solve exercises without using the feedback that would help them. In general, this could imply that the authors of automated learning environments should design which of the feedback needs to be actively pushed for learners and which of the feedback should be available only when requested by learners.
- Presented an open source tool for embedding Parson's puzzles into, for example, learning environments.
- Introduced an online environment where people can create, share, and solve two- dimensional Parson's puzzles.

In the future, we will use the interviews of the experts to create more algorithmic Parson's puzzles. We will then use these with students to evaluate the suitability of two-dimensional Parson's puzzles in teaching algorithms. We have already implemented a way to record all user actions in JSParsons and send the logs to a server. Mining this data will hopefully give us a better understanding of how exercises are really solved and how the tool may affect learning of programming

Two-dimensional puzzles are more complicated when compared to similar puzzles where only the order of the lines needs to be solved. How to extend the concept of two-dimensional puzzles to other programming languages provides interesting usability challenges for future research. In Java, for example, our two dimensional UI could dynamically modify the code by inserting and removing curly braces based on the indentation.

References

Biggs, J., & Collis, K. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.

Carter, J., English, J., Ala-Mutka, K., Dick, M., Fone, W., Fuller, U., & Sheard, J. (2003). ITICSE working group report: How shall we assess this? *SIGCSE Bulletin, 35*(4), 107-123.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008). Evaluating a new exam question: Parsons problems. *ICER '08: Proceedings of the Fourth international Workshop on Computing Education Research*, ACM, New York, NY, USA, pp. 113–124.

Détienne, F. (1990). Expert programming knowledge: A schema-based approach. In J.-M. Hoc, T. R. G. Green, R. Samurcay, & D. J. Gilmore (Eds.), *Psychology of programming* (pp. 205–222). London: Academic Press.

Garner, S. (2007). 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. Retrieved from http://proceedings.informingscience.org/InSITE2007/IISITv4p491-501Garn260.pdf

Isohanni, E. & Knobelsdorf, M. (2010). Behind the curtain: Students' use of vip after class. *ICER '10: Proceedings of the Sixth international workshop on Computing education research,* ACM, New York, NY, USA, pp. 87–96.

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist, 41*(2), 75–86.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Thomas, L. (2004). A multi- national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119–150.

Lister, R., Clear, T., Simon, Bouvier, D. J., Carter, P., Eckerdal, A., . . . Thompson, E. (2009). Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *SIGCSE Bulletin, 41*(4), 156–173.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: Novice programmers and the solo taxonomy. *SIGCSE Bulletin, 38*(3), 118–122.

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *ICER '08: Proceedings of the Fourth International Workshop on Computing Education Research,* ACM, New York, NY, USA, pp. 101–112.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B-D., . . . Wilusz, T. (2001). A multi-national, multi- institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bulletin, 33*(4), 125–180.

Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research.* ACM Press, New York, NY, USA, pp. 57–67.

Parsons, D., & Haden, P. (2006). Parson's programming puzzles: A fun and effective learning tool for first programming courses. *ACE '06: Proceedings of the 8th Australian Conference on Computing Education*, Australian Computer Society, Inc., Darlinghurst, Australia, pp. 157–163.

Rajala, T., Laakso, M.-J., Kaila, E., & Salakoski, T. (2007). ViLLE — A language-independent program visualization tool. In R. Lister & Simon (Eds.), *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, Vol. 88 of CRPIT, ACS, Koli National Park, Finland, pp. 151–159.

Soloway, E., & Ehrlich, K. (1986). Empirical studies of programming knowledge. In C. Rich & R. C. Waters (Eds.), *Readings in artificial intelligence and software engineering* (pp. 507–521). San Francisco, CA, USA: Morgan Kaufmann Publishers.

Appendix

Solutions to puzzles discussed in the section of problem solving strategies are presented here. If there are multiple lines inside one box, this means that those lines were one draggable element in the puzzle. Possible distractors are the last lines with `#distractor` comment string at the end of the line.

When puzzles were presented, all the lines (including distractors if turned on) were shuffled and presented in a random order. Distractors were shown like all the other lines, i.e. without any hints which lines were distractors.

First Exercise

```
def helloWorld():
    for i in range(5):
        print "Hello"
        print "World"
```

Swap

```
temp = a
a = b
b = temp
```

Binary Tree Levelorder Traversal

```
def traverse_levelorder(tree_node):
    queue.put(tree_node)
    while not queue.empty():
        T = queue.get()
        if t is not None:
            visit(t)
            queue.put(t.left)
            queue.put(t.right)
queue.put(t) #distractor
```

Binary Tree Postorder Traversal

```
def traverse_postorder(tree_node):
    if tree_node is not None:
        traverse_postorder(tree_node.left)
        traverse_postorder(tree_node.right)
        visit(tree_node)
if tree_node is None: #distractor
traverse_postorder(tree_node) #distractor
```

Binary Tree Inorder Traversal

```
def traverse_inorder(tree_node):
    if tree_node is not None:
        traverse_inorder(tree_node.left)
        visit(tree_node)
        traverse_inorder(tree_node.right)
if tree_node is None: #distractor
traverse_inorder(tree_node) #distractor
```

Binary Tree Preorder Traversal

```
def traverse_preorder(tree_node):
    if tree_node is not None:
        visit(tree_node)
        traverse_preorder(tree_node.left)
        traverse_preorder(tree_node.right)
if tree_node is None: #distractor
traverse_preorder(tree_node) #distractor
```

Selection Sort

```
def selectionSort(a):
    for i in range(0, len(a)):
        min = i
        for j in range(i+1, len(a)):
            if a[j] < a[min]:
                min = j
        temp = a[min]
        a[min] = a[i]
        a[i] = temp
if a[i] < a[j]: #distractor
min = I #distractor
```

Insertion Sort

```
def insertionSort(a):
    for i in range(0, len(a)):
        temp = a[i]
        j = i
        while j > 0 and a[j-1] > temp:
            a[j] = a[j-1]
```

```
        j = j-1
    a[j] = temp
temp = a[i]
j = i-1
```

Selection Sort

```
def selection_sort(list):
    l = list[:] #create copy of the list
    sorted = []
    while len(l):
        lowest = l[0]
        for x in l:
            if x < lowest:
                lowest = x
        sorted.append(lowest)
        l.remove(lowest)
    return sorted
while(true): #distractor
```

Biographies

Petri Ihantola is a researcher, lecturer and PhD student from Aalto University, Finland. He has also worked as a software engineer based in Finland, Ireland and Switzerland. His research interests include software test automation, visualizations and automated feedback in CS education. In addition, Mr. Ihantola enjoys beautiful code, real ales, trekking, and dark chocolate.

Ville Karavirta is a researcher and lecturer at Aalto University, Finland. He received his D.Sc. (Tech) diploma in 2009 from the same university (known as Helsinki University of Technology at that time). He is passionate about using visualizations and web and mobile technologies to improve education, especially to teach programming and computer science in general. When not working, you can probably find him eating good food, sleeping, doing sports, or moonlighting as an entrepreneur.