Automatically Detectable Indicators of Programming Assignment Difficulty

Petri Ihantola
Aalto University
Department of Computer Science and Engineering
Helsinki, Finland
petri.ihantola@aalto.fi

Juha Sorva
Aalto University
Department of Computer Science and Engineering
Helsinki, Finland
juha.sorva@aalto.fi

Arto Vihavainen
University of Helsinki
Department of Computer Science
Helsinki, Finland
avihavai@cs.helsinki.fi

ABSTRACT

The difficulty of learning tasks is a major factor in learning, as is the feedback given to students. Even automatic feedback should ideally be influenced by student-dependent factors such as task difficulty. We report on a preliminary exploration of such indicators of programming assignment difficulty that can be automatically detected for each student from source code snapshots of the student's evolving code. Using a combination of different metrics emerged as a promising approach. In the future, our results may help provide students with personalized automatic feedback.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

Keywords

automated assessment; programming assignments; assignment difficulty; personalized feedback

1. INTRODUCTION

In typical CS and IT curricula, introductory programming courses are among the first that students take. The rest of the curricula build on the skills learned in those courses, and indeed success in introductory courses affects whether students' continue with their studies or not [7]. It is not surprising that ways to improve introductory-level programming education has been under study for decades [21, 29].

Some decades ago, programming was a skill needed by a select few. It was often learned and taught in an *ad hoc* fashion, as educators often sought to replicate the ways in which they had themselves happened to learn to program, and there was no pressure to educate large numbers of graduates. From the 1980s onwards, programming has become increasingly mainstream, to the point that some countries have included programming education in primary school (e.g. [9]). At the same time, the practices and tools used to teach programming have evolved; these include novice-friendly programming environments [16] and microworlds [10], languages for beginners [20, 24], and program visualization tools [26], among others. Nevertheless, this evolution is arguably lagging behind the demand for more programmers and better programming pedagogy.

Pedagogical approaches to programming have been proposed in which students in read code and study many worked examples [17, 18]; software tools have also been designed to support activities such as code-reading and multiple-choice questions that help develop knowledge of important concepts. Such activities can be very useful in a complementary role, but if the goal is to learn to write programs, the pedagogy should be aligned with that goal [4] and must eventually include activities in which the students practice writing programs. In a university course or similar formal learning context, this means that the pedagogy needs to include *programming assignments.*

As with any form of practice, two key aspects of a programming assignments are feedback and student motivation. These aspects are connected, as good feedback can not only help the learner with the topic of the assignment but also increase the learner's motivation. Poor feedback, on the other hand, can make a learner less inclined to persist with a programming task or an entire course.

Feedback can be partially or fully automatic [28, 14]. Automatic assessment systems and intelligent tutoring systems bring benefits such as easy accessibility and low cost per student, which makes them particularly attractive in large classes with hundreds or thousands of students — and even more so in the context of modern massive online courses. The downside of many automatic solutions is that they fall short of a human tutor in terms of quality of feedback. A part of this problem is that the feedback provided by automatic systems is usually not personalized to take the learner and the learner's present knowledge into account. In order to provide better automatic feedback, we need to be able to judge how the individual learner (or group) relates to the assignment at hand. For instance, do they find it difficult? Trivial? Helpful? Ideally, a reasonably reliable estimate of such factors could be elicited automatically.

SIGITE'14, October 15–18, 2014, Atlanta, Georgia, USA.
Copyright 2014 ACM 978-1-4503-2686-5/14/10 ...$15.00.
http://dx.doi.org/10.1145/2656450.2656476.

2. RESEARCH QUESTION

This article presents a study which is a preliminary exploration of factors that may influence the difficulty of programming assignments and metrics for automatically assessing those factors. More specifically, we explore the research question: *How do a learner's programming background and automatically analyzable programming behavior relate to the perceived difficulty of different programming assignments?*

The work is motivated by the current state of automated assessment systems for programming assignments: We believe that an appropriate next step in the automated assessment of programming assignments is the ability to provide feedback that is adjusted to fit particular students' needs and struggles. One aspect of this development is that feedback should be adjusted to match students' perceptions of assignment difficulty. Ideally, such feedback could be provided without constantly prompting students to assess the difficulty of the various assignments they work on.

The remainder of this article is organized as follows. First, we review the related literature in Section 3 below. Sections 4 and 5 outline our research methodology and present our empirical results and related discussion. Section 6 discusses some limitations of our work and possibilities for expanding on this exploratory study; Section 7 concludes the article.

3. RELATED WORK

Related work is explored through four themes. We start with theories of learning as we discuss the relationship between task difficulty, practice and motivation, which leads to the second theme of feedback. This in turn brings us to the third theme: software for automatic assessment. Finally, we consider those empirical studies within computing education research which resemble ours in that they have measured students' difficulties with programming assignments.

3.1 Task difficulty and practice

Expertise is not innate. It commonly grows through *deliberate practice*, that is, effortful activity whose purpose is to optimize improvement [11]. The importance of deliberate practice is reflected in programming courses around the world, which are designed around assignments that afford students with the opportunity to practice their programming skills on increasingly complex tasks.

Practice can be more or less effective. Ideally, the difficulty of an assignment matches the learner's existing knowledge and skills so that the learner is challenged to make use of their full cognitive capacity but is not overwhelmed by *cognitive load* [22]. Although the ideal is difficult to meet, not least because learners' prior knowledge varies, teachers may consider their students' expected learning trajectories and sequence assignments accordingly. Models of instructional design have been proposed to support these endeavors (e.g., the 4C/ID model [27]). The design and sequencing of assignments may be viewed as a form of *scaffolding* that aids the learner to make progress within their *zone of proximal development* [31] as they practice on tasks that they could not do without the help of the scaffolding.

Task difficulty impacts students' motivation in several ways. For instance, as per expectancy–value theories of motivation [2], assignments that are too easy are likely to have low perceived utility, while hard ones have a higher cost of completion, which reduces motivation unless they have been carefully designed to sustain interest. Excessive difficulty also contributes towards poor *self-efficacy* [3], which hampers further learning.

Another form of scaffolding that impacts motivation is the feedback that learners receive.

3.2 Feedback and motivation

Hattie and Timperley [12] argue that three main roles of feedback are to help a learner understand 1) the goals of learning, 2) the learner's own progress towards those goals, and 3) the activities that are needed to make better progress. For present purposes, the second role—the progress made by the learner—is the most salient.

A teacher or educational environment can help a student reflect on their progress by providing feedback that relates the student's performance to a particular goal or subgoal. Constructive feedback can improve self-efficacy. Constructive does not always imply positive, however, and feedback on progress should take into account the student's background and prior performance as well as the difficulty of the task. A beginner completing a difficult task should be applauded, but as Borich and Tombari argue on the basis of the literature, teachers who "show surprise at [students'] success, give excessive unsolicited help, or lavishly praise success on easy tasks are telling students that they lack ability" [5]. Such feedback can be detrimental to self-efficacy and motivation. Inappropriate feedback may also quickly cause students to learn to distrust the feedback-giving teacher or environment. The matter is, of course, complicated by the fact that an activity is not equally challenging to all learners.

3.3 Automatic feedback

An on-campus lab with an instructor and a small number of students is a setting that is well-suited to good, individualized feedback [8]. When such labs are not an option, or as a supplementary measure to them, feedback may be worked into course materials and programming assignments, which can be delivered online.

There is a robust field of research that seeks to improve the automatic assessment of students' solutions to programming problems [14]. Typically, automatic feedback is provided after students' take an action such as submitting a solution for assessment; the feedback often consists of information on the correctness of the solution and perhaps some additional information about observed deficiencies. The feedback may also praise the student for getting a good score or exhort them to make an improved attempt.

Two weaknesses of the typical approach discussed above are: 1) The feedback is "passive", as it is only presented when the student requests it, e.g., by submitting a solution, instead of being proactively offered, say, when the student is experiencing difficulty. 2) Feedback messages are based on the features of the submitted solution only, and are not influenced by other relevant factors such as the student's background or the difficulty of the task for the particular student. For instance, an experienced student may receive excessive accolades for a trivial assignment, which then undermines any praise received for more challenging ones.

3.4 Programming assignment difficulty

In this subsection, we briefly review some work similar to ours, that is, projects whose purpose has been to evaluate the difficulty of programming tasks.

Alvarez and Scott studied the relationship between the student-estimated difficulty of programming assignments and a number of metrics [1]. They used a survey that asked students to estimate difficulty twice, first after initially familiarizing themselves with an assignment and again after finishing it. The highest correlations to estimated difficulty were found using code metrics such as lines of code and the amount of control flow statements within the code.

Several threads of research exist that have utilized data recorded from the students' programming process. Although these studies generally have not focused on assignment difficulty, some of them have explored related phenomena. For instance, Jadud [15] proposed a formula for quantifying compilation errors, which has been used to identify students' course and assignment outcomes. In another study, Rodrigo and Baker [25] sought to identify students that are frustrated using both log data as well as observations from external observers. It is plausible that compilation errors and frustration do correlate positively with difficulty.

Another approach could be to estimate the cognitive load of students: cognitive load depends on both the intrinsic difficulty of a learning task and the prior knowledge that students bring to it. One way to estimate cognitive load is to use a suitable questionnaire. This approach, which is being explored in a programming context by Morrison et al. [19], has the benefit of using validated instruments and a solid theoretical basis, but since it requires a survey with multiple items, it is not suitable for our purposes. Another method also based on cognitive load is featured in a recent pilot study [6], in which the concept of "thrashing" was operationalized by measuring mouse clicks in an IDE; thrashing was taken to be an indication of (excessive) cognitive load. The results of the study demonstrated that different programming languages lead to different patterns of thrashing, which may be indicative of differences in difficulty.

In the present study, we seek to explore new metrics for automatically identifying which programming assignments different students find difficult. Our intention is to take one step closer to providing better, individualized, motivating automatic feedback that takes into account not only the student's program but also task difficulty as experienced by the particular student.

4. DATA AND METHODS

The data used in this study comes from an open online programming course offered by the University of Helsinki during Spring 2014. It is a six-week Java course in which students are taught procedural programming for the first three weeks and object-oriented programming for the second three-week period. The course is taught using an assignment-intensive teaching style, where majority of the work is done within a programming environment. Details of the course have been previously published in [30].

After each assignment, students could provide numeric feedback on the difficulty of the assignment. The difficulty was given on a scale from 1 to 5, where one stands for "easy" and five for "hard". In addition, the programming environment used in the course stored key-level snapshot data, that is, each key-press by a student while working on a programming assignment was recorded. None of the questions were mandatory, and students could turn off the key-level snapshot data gathering at will. At the beginning of the course, the participants were asked to provide details on their programming experience.

The data set used in this study contains information on 417 students. This is after we included only students who had provided details on their programming background, provided feedback on assignment difficulty on at least three occasions, and kept the key-level snapshot recording enabled. Overall, the included students submitted 31255 solutions to assignments and provided details on the difficulty of an assignment 11161 times. That is, in about 36% of the submissions, the participant also provided feedback on the assignment difficulty.

The snapshot data was processed to include a time stamp as well as information on compilation state, i.e., whether the source code in each snapshot compiles. This data was aggregated to provide information on the process that each student took to solve an assignment. More specifically, for each assignment that a student works on, we aggregated details on (1) the time spent on the assignment, (2) the number of keystrokes made, (3) the percentage of keystrokes and time in a non-compiling state, (4) the number of lines of code, and (5) the number of control-flow elements in the program (e.g. *if, else, while, for, return*). By "time spent on the assignment" we mean the overall time spent modified with by truncating any pause of over five minutes between keystrokes to only five minutes. The number of keystrokes and the percentage of time/keystrokes in a non-compiling state are also potential indicators of struggling to make progress; if a student spends more time in a state where the code does not compile, or takes more steps than others while solving the problem, the assignment may seem more difficult overall. Line and control-flow element counts measure code complexity, and have previously been observed to be decent indicators of perceived difficulty [1].

We used quantitative analysis to identify factors that explain programming assignment difficulty. Correlations between the students' perceived difficulty and factors were computed using the R statistics package [23].

5. RESULTS AND DISCUSSION

This section describes our results in three parts. First, we discuss the effect of programming experience on perceived assignment difficulty. Then, we consider the relationships between perceived difficulty and individual factors: time, number of keystrokes, compilation state, lines of code, and the number of control-flow elements. Finally, we look at combining the various factors.

5.1 Programming Experience

In order to evaluate the effect of prior programming experience, the students were split in two groups on the basis of on their background. Of the participants, 230 reported no previous programming experience, while 187 described at least some experience with programming. Figure 1 displays the average perceived difficulty of each assignment for the groups, as well as a combined metric for all participants. As a Shapiro-Francia test revealed that the populations do not follow a normal distribution, a Wilcoxon signed-rank test was used as the paired difference test to measure whether the population means differ.

For the population with at least some existing programming experience, the median difficulty of the assignments is 1.735, while for the population with no previous program-

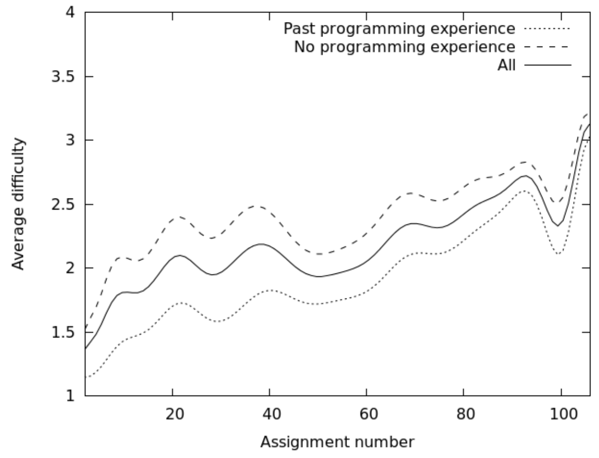

Figure 1: The means of students' estimates of the difficulty of the assignments. The curves have been smoothed for ease of viewing.

Table 1: Correlation coefficients between perceived assignment difficulty and various metrics, grouped by assignment type (math-oriented vs. open-ended vs. visually supported). Correlations marked with an asterisk are statistically significant (p < 0.01).

Factor	All	Math	Open	Vis
All participants				
Time	.49*	.48*	.30*	.48*
Number of keystrokes	.44*	.42*	.27*	.39*
% states not compiling	.16*	.10	.03	.33*
% time not compiling	.03*	.09	.11	.20*
Lines of code	.36*	.27*	.24*	.26*
Control-flow elements	.35*	.26*	.22*	.38*
Programming experience				
Time	.54*	.45*	.36*	.45*
Number of keystrokes	.50*	.47*	.35*	.37*
% states not compiling	.15*	.10	.04	.27*
% time not compiling	.01	.11	.02	.13
Lines of code	.44*	.34*	.33*	.34*
Control-flow elements	.43*	.30*	.19	.36*
No programming experience				
Time	.46*	.48*	.25*	.53*
Number of keystrokes	.40*	.38*	.22*	.45*
% states not compiling	.16*	.10	.03	.36*
% time not compiling	.03	.08	.22	.20*
Lines of code	.33*	.25*	.18*	.23*
Control-flow elements	.31*	.25*	.25*	.45*

ming experience, the median difficulty of the assignments is 2.375. The populations are statistically different ($p < 0.01$), and thus there is, as one would expect, a difference in how the two populations perceive the difficulty of the assignments. However, as can be observed from Figure 1, the between-group difference in mean perceived difficulty diminishes towards the end of the six-week course. This trend suggests that the course taught skills that the more experienced students already had to some extent, and that the beginners partially caught up with the more experienced students.

5.2 Individual factors

Initially, an analysis was carried out to determine the correlations between difficulty and various other factors, each of which was considered separately. These factors were: time, number of keystrokes, proportions of keystrokes and time in a non-compiling states, lines of code, and count of control-flow elements. Table 1 displays these correlations for all participants as well as beginners and experienced students separately. For each factor, the table shows four values: one for all assignments, one for mathematically oriented assignments, one for open-ended assignments, and one for assignments with visual elements such as a given GUI that helps evaluate one's progress.

While a majority of the observed correlations are statistically significant, the correlation values are mostly medium-sized ($0.3 < r < 0.5$). In only two of the cases, the individual factors show a high correlation with difficulty ($r > 0.5$); both factors being time. The pattern of correlations appears to be largely similar for students with and without prior programming experience.

The correlations that we found between perceived difficulty and the number of lines of code as well as the number of control-flow elements were lower than the corresponding results reported earlier by Alvarez and Scott [1]. In their study, lines of code and control-flow elements had the highest correlations with perceived difficulty, whereas in our data, time on task and the number of keystrokes had somewhat higher correlations.

As Table 1 further shows, in most cases we found only low, largely insignificant correlations between perceived difficulty and the factors related to compilation status. The assignments with visual programming support constitute an exception to this trend, as a low-to-medium positive correlation was observed in these assignments.

5.3 Combined factors

In the previous section, we considered individual indicators of assignment difficulty one at a time. To get an initial understanding of how these factors interact in the data set at our disposal, we applied a recursive partitioning to construct a decision tree of assignment difficulty. The model is based on the metrics presented in the previous section.

The model was built using the ctree implementation of R[1]. This method guarantees that the size of the tree is appropriate so that no pruning or cross-validation is necessary. A general description of the method is provided by Hothorn et al. [13].

The resulting decision tree is depicted in Figure 2. At each end node (leaf), a range of difficulty values is shown. This is the range of all the assessments of difficulty by students whose development snapshots matched the decision nodes leading to the end node. As can be seen from the figure, the tree is dominated by the amount of time that the student spent on the assignment. Although program size, complexity, and the degree to which the student maintained their program in a compilable state had an effect, students generally reported time-consuming exercises to be difficult.

6. LIMITATIONS AND FUTURE WORK

Our data comes from a particular programming course taught in a particular way in a Nordic country with a rather homogeneous population and high quality of education. Our results may be context-dependent. Indeed, as a part of the

[1]`http://www.inside-r.org/packages/cran/party/docs/ctree`

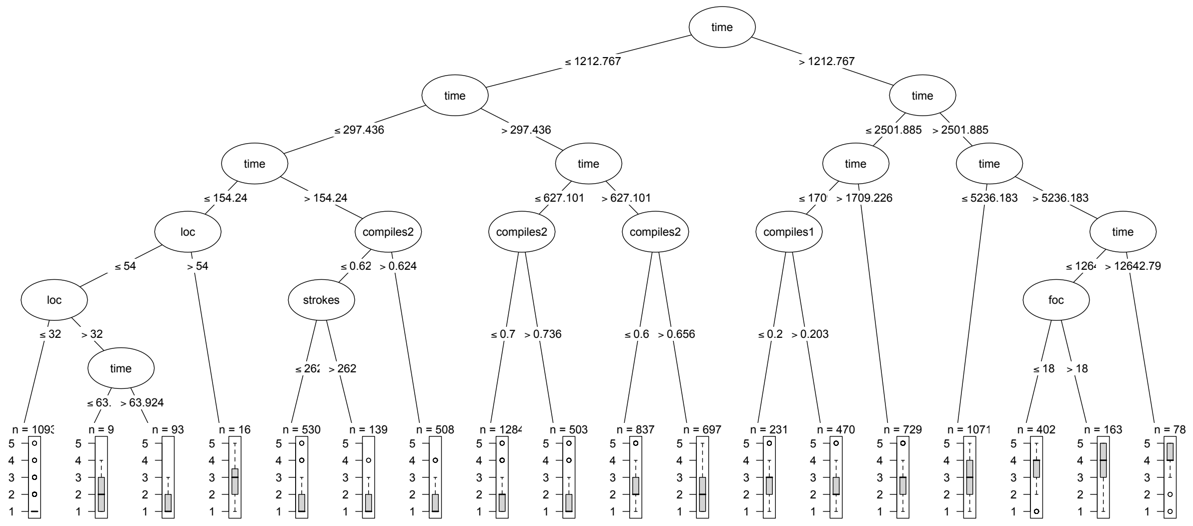

Figure 2: A decision tree of assignment difficulty constructed using recursive partitioning. *Loc* stands for lines of code and *foc* for flow-of-control count. *Compiles1* and *compiles2* stand for the proportion of compiling states and the proportion of time during which a program compiles, respectively.

present work, we re-examined the factors such as lines of code and number of control-flow elements that had the highest correlations with difficulty as reported by Alvarez and Scott [1], and while our work confirms that these metrics correlate positively with perceived difficulty, we found lower correlations for these factors than the other study. This difference in results suggests that the phenomena we have studied are at least in part, and perhaps quite significantly, dependent on context. Although a decision tree such as the one in Figure 2 may be useful in tailoring feedback in a particular context, other contexts need to be addressed separately. Future work may explore different programming courses, introductory or otherwise, and determine the extent to which our findings are transferable.

Our study draws on students' subjective assessments of difficulty. However, different students may have different interpretations of what "difficult" means. As pointed out in Section 5.3, much of "assignment difficulty" can be explained by the time it takes from students to do the assignment; similarly, Alvarez and Scott [1] reported that the size of the program was an important factor. A challenge in research such as ours, and one that we have not addressed in the present work, is teasing apart difficulty and workload, or at least determining the extent to which the distinction between the two is important for providing good feedback. The decision tree approach used above would easily accommodate additional variables, if necessary.

Another limitation is that although each assignment received a difficulty rating from at least 60 different students, providing the ratings was voluntary and we cannot rule out the possibility of an inherent self-selection bias. It is possible that the students that provided ratings are different from other students.

In this work, we split students in two groups on the basis of their prior programming experience. Future work could replace this simple model with a more fine-grained one so as to explore the variation among the experienced students.

To enable a critical examination of our results and facilitate follow-up studies, we have published our data set at `http://bit.ly/1oZnEKG`.

7. CONCLUSIONS

In this article, we have explored factors that relate to the perceived difficulty of programming assignments. The factors, excluding past programming experience, can be automatically detected from a stream of programming events—or keystrokes—that are performed within a programming environment. Our analysis suggests that the time spent on an assignment and the amount of programming events both have a medium to high correlation with perceived difficulty. Barely any correlation was found between perceived difficulty and the number of states that fail to compile or the length of time that the student's program is in a non-compiling state.

Although automatic feedback remains a far cry from what a good human tutor can provide, many students do not have convenient access to good human tutors, and any advance in automatic feedback is welcome. Our results show that metrics related to perceived difficulty can be automatically extracted from data that describes students' programming process. Automatic feedback systems can be adjusted to take task difficulty into account, which may improve the quality of feedback.

We conclude this article with some observations about the use of key-level data of student behavior. As a basis for assessing difficulty, this data has the benefit that it can be collected *in situ* and makes it possible to provide early, proactive feedback that the student does not need to explicitly request by submitting an assignment. An additional benefit, whose implications for automatic feedback may be explored in future work, is that such data provides details about students' programming process. Since, as Hattie and Timperley put it, "feedback is effective when it consists of information about progress, and/or about how to proceed" [12], key-level data has the potential to further enhance feedback.

8. REFERENCES

[1] A. Alvarez and T. A. Scott. Using student surveys in determining the difficulty of programming assignments. *J. Comput. Sci. Coll.*, 26(2):157–163, Dec. 2010.

[2] E. M. Anderman and H. Dawson. Learning with motivation. Routledge, 2011.

[3] A. Bandura. Self-efficacy: Toward a unifying theory of behavioral change. *Psych. Review*, 84(2):191, 1977.

[4] J. Biggs and C. Tang. *Teaching for Quality Learning at University*. McGraw-Hill, 3rd edition, 2007.

[5] G. D. Borich and M. L. Tombari. *Educational Psychology: A Contemporary Approach*. Longman Publishing/Addison Wesley, 2nd edition, 1997.

[6] S. Buist. Extending an IDE to support input device logging of programmers during the activity of user-interface programming: Analysing cognitive load. Bachelor of Science dissertation, The University of Bath, 2014.

[7] A. Christopher Strenta, R. Elliott, R. Adair, M. Matier, and J. Scott. Choosing and leaving science in highly selective institutions. *Research in Higher Education*, 35(5):513–547, 1994.

[8] M. Clancy, N. Titterton, C. Ryan, J. Slotta, and M. Linn. New roles for students, instructors, and computers in a lab-based introductory programming course. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 132–136, New York, NY, USA, 2003. ACM.

[9] Computing At School. Computing at school web site. `http://www.computingatschool.org.uk/`, n.d.

[10] S. Cooper, W. Dann, and R. Pausch. Alice: A 3-D tool for introductory programming concepts. *J. Comput. Sci. Coll.*, 15(5):107–116, Apr. 2000.

[11] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psych. Review*, 100(3):363, 1993.

[12] J. Hattie and H. Timperley. The power of feedback. *Review of Educational Research*, 77(1):81–112, 2007.

[13] T. Hothorn, K. Hornik, and A. Zeileis. Unbiased recursive partitioning: A conditional inference framework. *Journal of Computational and Graphical Statistics*, 15(3):651–674, 2006.

[14] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, pages 86–93. ACM, 2010.

[15] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research*, ICER '06, pages 73–84. ACM, 2006.

[16] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[17] M. C. Linn and M. J. Clancy. The case for case studies of programming problems. *Communications of the ACM*, 35(3):121–132, March 1992.

[18] R. Lister. Concrete and other neo-Piagetian forms of reasoning in the novice programmer. In J. Hamer and M. de Raadt, editors, *Proceedings of the 13th Australasian Conference on Computing Education (ACE '11)*, volume 114 of *CRPIT*, pages 9–18, Perth, Australia, 2011. Australian Computer Society.

[19] B. B. Morrison, B. Dorn, and M. Guzdial. Measuring cognitive load in introductory CS: Adaptation of an instrument. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 131–138, New York, NY, USA, 2014. ACM.

[20] S. Papert. *Teaching Children Thinking (LOGO Memo)*. Massachusetts Institute of Technology, A.I. Laboratory, 1971.

[21] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson. A survey of literature on the teaching of introductory programming. In *ITiCSE Working Group Reports*, ITiCSE-WGR '07, pages 204–223, New York, NY, USA, 2007. ACM.

[22] J. L. Plass, R. Moreno, and R. Brünken, editors. *Cognitive Load Theory*. Cambridge Univ. Press, 2010.

[23] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[24] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, and Y. Kafai. Scratch: Programming for all. *Commun. ACM*, 52(11):60–67, Nov. 2009.

[25] M. M. T. Rodrigo and R. S. Baker. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the Fifth International Workshop on Computing Education Research*, ICER '09, pages 75–80, New York, NY, USA, 2009. ACM.

[26] J. Sorva, V. Karavirta, and L. Malmi. A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education*, 13(4):15:1–15:64, Nov. 2013.

[27] J. J. G. van Merriënboer and P. A. Kirschner. *Ten Steps to Complex Learning: A Systematic Approach to Four-Component Instructional Design*. Lawrence Erlbaum, 2007.

[28] K. VanLehn. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist*, 46(4):197–221, 2011.

[29] A. Vihavainen, J. Airaksinen, and C. Watson. A systematic review of approaches for teaching introductory programming and their influence on success. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 19–26, New York, NY, USA, 2014. ACM.

[30] A. Vihavainen, M. Luukkainen, and J. Kurhila. Multi-faceted support for MOOC in programming. In *Proceedings of the 13th Annual Conference on Information Technology Education*, SIGITE '12, pages 171–176, New York, NY, USA, 2012. ACM.

[31] L. S. Vygotsky. *Mind in Society: The Development of Higher Psychological Processes*. Harvard University Press, 1978.