Using intelligent tutoring on the first steps of learning to program: affective and learning outcomes

Maciej Pankiewicz[1], Ryan Baker[2], Jaclyn Ocumpaugh[2]

[1] Warsaw University of Life Sciences, Warsaw, Poland
[2] University of Pennsylvania, Philadelphia, USA
maciej_pankiewicz@sggw.edu.pl

Abstract. There exist several online applications for automated testing of the computer programs that students write in computer science education. Use of such systems enables self-paced learning with automated feedback delivered by the application. However, due to the complexity of programming languages, even the easiest tasks made available through such systems require understanding of several programming concepts and formatting. Therefore, a student's initial work in an introductory computer science course may be highly challenging, especially for students with no previous programming background.

To address this challenge, a highly-decomposed micro-task module has been developed and made available on an automated assessment platform with programming assignments. Impact of its introduction has been examined within an introductory programming university course with 239 participants. We investigated the micro-task module's impact on student affect, student performance on the platform, and student learning outcomes. Results of the experiment show that students in the experimental group (with micro-tasks enabled) significantly less frequently reported frustration, confusion and boredom, needed less time to solve tasks on the platform and achieved significantly better results on the final test.

Keywords: Computer science education, Programming micro-tasks, Affect.

1 Introduction

Despite decades of research on helping students learn how to program [1, 2], many introductory students do not advance to more complicated topics and coursework [3]. Even when supported by educational tools and scaffolds, students with no programming background struggle in the first weeks of the course [4], where the complexity requires students to self-regulate through the acquisition of several cognitive skills that are introduced simultaneously [5-7]. These courses are also difficult to plan for because of their heterogeneous student populations. While some students have never programmed before or have only used graphical programming languages such as Scratch, other students have more experience. *Intelligent tutoring systems* (ITS) could be of help to these problems. However, developing ITS for this domain seems to be a challenge. A review

of intelligent tutoring systems for programming [8] finds only one example of an intelligent tutoring system for a modern programming language in the years since 2010 [9].

By contrast, the number of automated assessment platforms used in computer science education constantly increases [10]. The purpose of these platforms is to actually execute the code created by the student against a set of several test cases to provide immediate feedback on its correctness. Educational benefits often result from the fact that they may be used as a guide that helps students track if they are achieving their learning goals, simultaneously providing teachers more insight into student progress [10].

However, even when immediate feedback is available, students who do not understand why things have been marked as incorrect, may experience confusion, frustration or anxiousness and these negative affect states can negatively impact student outcomes [11-13]. If key aspects of intelligent tutors could be embedded into these platforms, it might be a feasible way to improve student outcomes. In this paper, we investigate a tool that breaks down the earliest steps of learning to program using micro-tasks (much like the earliest ITS for programming [1]), embedded within a test suite platform. We conducted a controlled experiment to investigate whether adding the micro-tasks to the test suite platform improves student learning and affect.

2 Methods

The online platform with the micro-tasks module. The online application *RunCode* [14] is a platform for automated execution and testing of programming code. It has been used since 2017 by students of computer science at the Warsaw University of Life Sciences. Usage of the platform is not mandatory, but students willingly use it (91% in Winter Semester 2021-2022). The *RunCode* app provides students with 146 programming tasks covering the following programming topics: types and variables (33), conditional statement (25), recursion (28), loops (17) and arrays (43). When a student submits a solution (programming code) to the task, the code is compiled, and tested. The student is then provided with several types of feedback, including the total score, the information from the compiler (if the code didn't compile) and detailed results for each test case executed on the submitted code (if the code compiled).

Most programming assignments in introductory courses require students to write several lines of code so that a program is ready to be executed against a defined set of test cases. With every line, statement, operator and punctuation, the complexity of the code increases. When faced with so many programming language components, students may find it hard to keep track of each detail of the created code. To prevent this problem, the platform provides students with a *micro-tasks* module designed to improve their understanding of a particular (single) element of programming code in more detail. Micro-tasks are coding tasks that focus on one particular aspect of the code and usually require the user to enter only one line of the code. They are designed to help students understand how different programming components work one-by-one. To achieve that, the line of code entered by the student is combined with a larger code structure. If the line of code is not correct, the student receives feedback on that specific element, in

order to help them relate the errors that occurred to that one element being tested at the time. In contrast, when writing larger pieces of code, even a small syntax mistake may lead to a large number of compiler errors, which makes understanding the dependencies in the code much more difficult.

A total of 103 micro-tasks were created, spanning the topics of: types and variables (46), conditional statement (7), recursion (23), loops (9) and arrays (18). In the following sections, we use *micro-task* only when referring to a programming task that is made available in the introduced micro-task module. An example of such a task could be: "use the = operator and assign an appropriate value to the variable age." We refer to all the regular coding tasks available on the platform as a *task*. An example of such a task could be: "create a function that returns true if a given number is odd." To evaluate the impact of the micro-tasks module introduced to the system, students were randomly assigned to a control or experimental group. Sets of tasks for each topic were published later in the day after the corresponding classes. Students in the control group could access tasks on the platform immediately after they were published. For the experimental group, access to the main tasks was unlocked after solving a set of micro-tasks. Micro-tasks were not available to students in the control group.

Participants. This study's participants consisted of first-semester computer science students (N=276) taking the *Introduction to Programming* course in Winter Semester 2022-2023 at the Warsaw University of Life Sciences. This course is required for computer science students at this university. The programming language used in this course is C#. A total of 239 students (28% female) consented to participate, with 174 submitting at least 1 solution on the platform during the study period (86 students in the control and 88 in the experimental group). In total, 31,011 submissions were collected (7,284 micro-tasks, and 23,727 regular programming tasks). In addition to these tasks, data consists of a single-item self-assessment of skills, results of the pre-test conducted during the first class session and the post-test conducted during the 10th class session.

Self-Assessment of Skills: novice and experienced programmers. During the first class session, students rated their level of knowledge on basic programming topics. Students were prompted, "Please rate on the scale 1 to 5 (where 1 – no knowledge at all and 5 – very good knowledge) your knowledge level for these topics taught in this course: types, variables, conditional statement, recursion, loops and arrays." Based on the self-assessment data, students were split into two groups: a "novice" group of students who reported their skill level as 1 and 2 (N=121; 59 experimental, 62 control) and an "experienced" group of students who self-reported a skill level of 3 and above (N=118; 54 experimental, 64 control). These values were selected to split students roughly evenly between the two groups. In the following sections we will refer to these groups as novice programmers and experienced programmers. Directly after the self-assessment, students were administered a pre-test to assess their knowledge level in a more objective manner. The test contained 8 multi-choice questions referring to the basic concepts taught during the course. The test questions were designed in a programming language-agnostic way, to take into account different programming languages that students may have previously learnt. The post-test was administered during the 10th class session, after all the relevant topics had previously been introduced during classes. The post-test contained 9 multi-choice questions.

Affective state while using the platform. Students self-reported their affective states while solving tasks on the platform through a dynamic HTML element. They were prompted to do so after they received submission results with the question: "Choose the option that best describes how you feel at the moment" with the following response options: *Focused*, *Anxious*, *Bored*, *Confused*, *Frustrated*, *Other* (in this order) and appropriate emoticons to visually highlight each of the available responses. This set of affective states was chosen based on their importance for learning and history of past research in AIED [12]. To avoid frustration that could arise from being required to fill out the survey too frequently, it was not presented after every submission, but randomly, with a probability of 1/3.

3 Results

Affective states. The most frequently reported state during regular programming tasks in both conditions was *focused* (more than half of collected responses in both conditions). A total of 6,051 survey responses were collected (experimental: 3,063, control: 2,988) from 128 students (experimental: 64, control: 64). Non-parametric Mann-Whitney U tests were calculated to compare differences in the frequencies of the affective states reported by each student (during regular programming tasks), due violation of normality assumptions. Due to multiple comparisons, the Benjamini-Hochberg alpha correction was used. Three affective states showed significant differences between conditions. For *boredom*, students in the experimental group reported marginally lower rates (*Mdn*=0) than those in the control group (*Mdn*=0.0139), (*W*=1566.5, adjusted α=0.01, p=0.0125, for a non-parametric Mann-Whitney U test). The same was true for *confusion*, where the experimental group (*Mdn*=0) was marginally significantly lower than for the control group (*Mdn*=0.007), (*W*=1641.5, adjusted α=0.02, p=0.029, for a non-parametric Mann-Whitney U test). *Frustration* followed the same pattern with the experimental group (*Mdn*=0) reporting marginally significantly lower rates than the control group (*Mdn*=0.03), (*W*=1674, adjusted α=0.03, p=0.053, for a non-parametric Mann-Whitney U test). Focus and anxiety, however, did not show significant differences between conditions. The experimental group reported a median that was not statistically significantly different from the control group (*Mdn*=0.72 vs *Mdn*=0.52), (*W*=2236, adjusted α=0.05, p=0.369, for a non-parametric Mann-Whitney U test). The same was true for *anxiety* (*Mdn*=0 vs. *Mdn*=0.02), (*W*=2236, adjusted α=0.04, p=0.149, for a non-parametric Mann-Whitney U test).

Learning outcomes. To evaluate the impact of the introduced micro-task module on the post-test outcomes, we used a rank-based regression [15] (a non-parametric alternative to traditional likelihood or least squares estimators) used to test whether the pre-test and group significantly predicted the post-test results. The pre-test was a statistically significant predictor of the post-test, $t(236)=14.53$, $p<0.001$. The group was a statistically significant predictor of the post-test as well, $t(236)=2.35$, $p=0.02$; on average, students in the experimental condition performed 5.08% better on the post-test after controlling for pre-test. For novice programmers, the pre-test was a statistically significant predictor of the post-test, $t(118)=3.50$, $p=0.001$. The group was also a

statistically significant predictor of the post-test, $t(118)=2.07$, $p=0.04$. In the group of novices, students in experimental condition performed on average 5.67% better on the post-test after controlling for pre-test. For experienced programmers, pre-test was again a statistically significant predictor of the post-test, $t(115)=6.07$, $p<0.001$, but group variable was not a statistically significant predictor of the post-test, $t(115)=0.89$, $p=0.373$.

Platform usage. Within the experimental group, the time spent by novice programmers on micro-tasks (*Mdn*=190.47 min.) was significantly higher than experienced programmers (*Mdn*=103.27 min.), for a non-parametric Mann-Whitney U test ($W=1406$, $p=0.003$). To evaluate differences between students in the control and the experimental group in terms of time spent to solve tasks on the platform, we analyzed 7391 successful student attempts on tasks during the study period and evaluated the time students in both conditions needed to solve tasks. In doing so, we omitted tasks which fewer than 3 students successfully completed in each condition. Some students ceased work on a task without logging out; due to the fact that the platform does not monitor the keystroke-level data, a cutoff has been defined for tasks that have not been solved within the time of 2000 seconds (~33 min.). The vast majority of student attempts on tasks ended with a successful submission within that time (but several successful attempts were still in the 30-33 minute range). The time needed to completely solve each task was significantly lower for the experimental group (*Mdn*=3.17 min.) than the time for the control group (*Mdn*=4.7 min.), $W=1765.5$, $p<0.001$, for a non-parametric Mann-Whitney U test. The time needed by novice programmers to completely solve each task was significantly lower for the experimental group (*Mdn*=3.45 min.) than for the control group (*Mdn*=4.98 min.), for a non-parametric Mann-Whitney U test ($W=468.5$, $p<0.001$). The time needed by experienced programmers to completely solve each task was marginally significantly lower for the experimental group (*Mdn*=3.3 min.) than for the control group (*Mdn*=3.8 min.), for a non-parametric Mann-Whitney U test ($W=404$, $p=0.053$).

4 Discussion

The results of the experiment show that students in the experimental group (micro-tasks enabled) were less likely to report frustration, confusion and boredom, needed less time than students in the control group to submit a correct solution to a programming task, and achieved better results on the post-test overall. However, the benefits of the micro-tasks for learning were clearer for students declaring no previous programming experience than for students with past experience. This finding indicates that not all students need the micro-tasks, suggesting that it may be best to provide them on the basis of self-report of expertise or, better yet, based on automatically inferring which students are likely to need them from initial within-system performance.

Limitation of this study is that we only studied impacts during the first ten classes of the semester; future work should study the impacts of micro-tasks over a longer time period. Finally, future work should study interventions of this nature in a broader selection of universities and introductory programming language, to establish generalizability. Overall, this initial evidence suggests that incorporating micro-tasks into

introductory computer science platforms may have benefits both for learning and affect, potentially increasing the number of students who succeed in introductory courses and continue further into CS programs.

References

1. Anderson, J. R., Reiser, B. J.: The LISP tutor. Byte, 10(4), 159–175 (1985).
2. Brusilovsky, P. L.: Intelligent tutor, environment and manual for introductory programming. Educational & Training Technology International, 29(1), 26–34 (1992).
3. Robins, A. V.: Novice Programmers and Introductory Programming. The Cambridge handbook of computing education research. In: The Cambridge Handbook of Computing Education Research, 327-376 (2019).
4. Prather, J., Pettit, R., McMurry, K., Peters, A., Homer, J., Cohen, M.: Metacognitive difficulties faced by novice programmers in automated assessment tools. In Proc. of the ACM Conf. on Int'l Computing Education Research, 41–50 (2018).
5. Falkner, K., Vivian, R., Falkner, N. J.: Identifying computer science self-regulated learning strategies. In Proc. of the Conf. on Innovation & technology in computer science education, 291–296 (2014).
6. Renumol, V. G., Janakiram, D., Jayaprakash, S.: Identification of cognitive processes of effective and ineffective students during computer programming. ACM Transactions on Computing Education, 10(3), 1–21 (2010).
7. Alaoutinen, S.: Evaluating the effect of learning style and student background on self-assessment accuracy. Computer Science Education, 22(2), 175–198 (2012).
8. Crow, T., Luxton-Reilly, A., Wuensche, B.: Intelligent tutoring systems for programming education: a systematic review. In Proc. of the Australasian Comp. Education Conf., 53–62 (2018).
9. Rivers, K., Koedinger, K. R.: Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. Int'l J. of Artificial Intelligence in Education, 27(1), 37–64 (2017).
10. Paiva, J. C., Leal, J. P., Figueira, Á.: Automated assessment in computer science education: A state-of-the-art review. ACM Trans. on Comp. Education, 22(3), 1–40 (2022).
11. Rodrigo, M.M.T., Baker, R.S., Jadud, M.C., Amarra, A.C.M., Dy, T., Espejo-Lahoz, M.B.V., Lim, S.A.L., Pascua, S.A.M.S., Sugay, J.O., Tabanao, E.S.: Affective and behavioral predictors of novice programmer achievement. In Proc. of ACM SIGCSE, 156–160, (2009).
12. Karumbaiah, S., Baker, R.S., Tao, Y., Liu, Z.: How does Students' Affect in Virtual Learning Relate to Their Outcomes? A Systematic Review Challenging the Positive-Negative Dichotomy. In Proc. of the Int'l Learning Analytics and Knowledge Conference, 24–33 (2022).
13. Bosch, N., D'Mello, S.: The affective experience of novice computer programmers. Int'l J. of Artificial Intelligence in Education, 27(1), 181–206 (2017).
14. Pankiewicz, M.: Move in the Right Direction: Impacting Students' Engagement With Gamification in a Programming Course. In EdMedia+ Innovate Learning. Association for the Advancement of Computing in Education (AACE), 1180-1185 (2020).
15. Kloke, J. D., McKean, J. W.: Rfit: rank-based estimation for linear models. The R Journal, 4(2), 57–64 (2012).