Automatic analysis of students’ solving process in programming exercises

Aivar Annamaa, aivar.annamaa@ut.ee
University of Tartu, Liivi 2 Tartu, 50409
Estonia

Annika Hansalu, anzza8i8@ut.ee
University of Tartu, Liivi 2 Tartu, 50409
Estonia

Eno Tonisson, eno.tonisson@ut.ee
University of Tartu, Liivi 2 Tartu, 50409
Estonia

Abstract

In most cases student programming exercises are assessed based solely on the final program and a teacher does not know how students actually arrived at the solution or what steps they completed during the process. In this paper we review different approaches to observe how students solve their programming exercises and see how integrated development environment Thonny can be used to observe exactly what novice programmers do while programming. This is followed by a discussion of the benefits of (automatic) analysis of students’ solving process.

Keywords

Computer science education, automatic analyses of solving process, novice programmers

INTRODUCTION

Solving of programming exercises has a title role in computer science education. Usually, the students’ work is evaluated on the grounds of a completed program. The program could be assessed by the system for automatic assessment, which is almost inevitable in case of a large number of students. Human evaluation of program code is used, especially in case of beginner courses, in order to give more useful and specific feedback, which could be impossible with automatic assessment. In addition to the final program, observation of the solving process is also very important to help rationalise solving, save time, increase motivation, etc. The teacher could, in principle, observe the solving process by standing next to the computer when solving takes place in a classroom. This would be impossible if solving is processed outside of classroom and complicated in case of a larger number of students.

This paper is focussed on automatic analysis of the students’ solving process in programming exercises. The aim is to outline approaches that have been used for analysing the solving process. The approaches differ by granularity of data, for example. It is remarkable that the solving process is also (or even especially) interesting and useful in software engineering, not only for educational reasons. After the section on data collection, a brief introduction on potential benefits of such analysis for educational purposes is presented. It is closely related to different types of learners (like stoppers, tinkerers and movers (Perkins et al., 1989)). The innovative part of the paper introduces the logs from a new educational integrated development environment (IDE) Thonny (Annamaa, 2015). The conclusive section describes some possible future works as well.

DATA ON THE SOLVING PROCESS

An analysis of the programming process must start with data collection. The most straightforward approach would be recording the screen and possibly also the subject's body language and voice for later analysis. This approach combined with subsequent manual video analysis, or letting an expert directly monitor and comment on the process, is very flexible but also very expensive.

It would be more scalable to collect process data in a structured form allowing automatic analysis. In software engineering research, the analysis is usually based on source code snapshots extracted from version control systems (VCS). This makes data collection very simple, as using version control is a standard practice in professional context and therefore it is not necessary to set up a separate data collection mechanism. However, as Negara et al have explained (Negara 2012), many interesting aspects of code evolution get lost when we analyse only code snapshots at commit points.

Relying on conventional use of version control systems is even more problematic in the context of programming education, because requiring that students use a VCS can create an excessive cognitive load. Furthermore, beginner programming exercises are usually too small to be analysed on a scale where code commits usually mark the completion of a feature or a bug fix. For these reasons, the analysis of educational programming process is usually supported by a submission system, which may allow several submissions for the same task, for example Web-CAT (Edwards, 2009). More granular and diverse data can be collected by specifically designed or instrumented programming environments, which gather code snapshots on each compile or run.

Vihavainen et al show that in many cases even snapshots collected on each compile or run may be insufficient for getting realistic picture of the programming process, and it makes sense to collect detailed info about all relevant user actions, including key strokes and mouse presses, together with their time stamps (Vihavainen, 2014). If necessary, code snapshots can be constructed from these low level events for arbitrary points in time. There exist several such programming environments or IDE add-ons, for example, Fluorite for Eclipse (Yoon, 2013). In this paper we are mostly interested in these kinds of systems.

POSSIBLE BENEFITS

One could collect data on the solving process but it is important that the data supports improvement of learning and teaching. It is possible to describe different types of novice programmers. Perkins et al (1989) have divided them into three different learner types based on their problem solving strategy: stoppers, tinkerers and movers. Stoppers are students who when faced with a problem tend to give up faster and ask for help rather than trying to solve the problem themselves first. Tinkerers are students who solve problems by experimenting and making small changes in the code while hoping to get the code working. Movers on the other hand are students who move towards the right solution. They have a certain idea for the solution and they are not afraid to try different approaches if the first one does not take them closer to solving the problem.

Cardell-Oliver (2011) has noticed on her students that students who are stoppers do not tolerate too much negative feedback on their work because, if they get too much negativity at once, they tend to turn into non-starters. On the other hand, because of their problem solving strategies, tinkerers and movers benefit the most from detailed feedback. When tinkerers see that they are getting fewer errors it probably means that they are on the right track.

Other authors (Housseini et al, 2014) have found that the classification of students into stoppers, movers and tinkerers by Perkins et al is not enough. They concluded that it would be more accurate to divide student problem solving strategies into four groups: builders, massagers, reducers, and strugglers. Builders are students who constantly add new concepts to their code and by doing that improve correctness of their code. Massagers are similar to builders but they have periods where they only do small code changes without adding or removing any new concepts. Reducers are students who in the beginning take a completed code from somewhere (i.e., from a previously solved exercise) and start removing unnecessary concepts. They remove things until they get the right solution. Strugglers are the students who struggle with their code and make all kinds of changes in their code but they tend to have not enough knowledge to get their code working or find the mistakes and fix them.

Analyses of logs could provide valuable information for more specific classification of novice programmers' solving style. For example, it is possible to observe a particular learner for longer time and get information about persistency of behaviours and impact of feedback. The ultimate task of teaching – provide as tailored feedback as possible – could also be accomplished better with the help of (hopefully automatic) analysis of logs. At first glance, even only awareness of his or her possible weaknesses in the solving process could help a student (and a teacher).

THONNY LOGS

Thonny is a new Python IDE developed in the University of Tartu, designed for learning and teaching programming. Besides program editing and execution capabilities, its most prominent feature is support for program animation – the user can easily step through the execution of the program and follow the changes in the program's runtime state (including global and local variables and call stack).

In order to gain better insight into the solving process, we made Thonny log all interesting events that happen in its window (although we might not need them all because we are still researching which data will give us the needed information). For each usage session Thonny creates a log file containing descriptions of the actions performed by the user. These actions include loading and saving files, modifications to the program text (paste can be distinguished from typed text, for example), program executions, writes into and reads from the program's standard streams, stepping commands, losing and gaining the focus of Thonny window, etc. Each action gets recorded together with its time stamp. The collected information can be used to replay the whole process of program construction and the activities in the shell. For this Thonny provides a separate window where one can choose a log file and see the events replayed at selected speed.

Thonny has been used for one semester in the Programming (Computer Science 1) course at the University of Tartu. The course had 280 participants; half of them were first-year computer science students while the other half consisted mostly of students from related fields (mathematics, statistics). Its program animation features were initially used only in the lectures for demonstrating Python's run time behaviour, but many students chose to use Thonny also on their own for solving exercises in labs and at home.

Before a midterm examination we offered our students extra credit if they solved the programming exercises in Thonny and sent us the log files describing their actions during the midterm. We got logs from 44 students. For proof of the concept we created a small summary of the logs and from this data we learned, for example, that

- a student who used deletion commands two times more often and undo command 10 times more often than students in average, produced one of the best solutions;

- one third of the program executions generated error messages, one third of the errors were syntax errors;
- one of the top students got error messages for 90% of his program executions, another top student for 20%;
- only 14 students had used Python shell for executing statements or evaluating expressions;
- 27 students used the program animation features, 8 of them invoked more than 1,000 animation steps during 100 minutes.

Besides analysis of numeric summaries, we created a visualization of all users' editing and program execution actions on a unified timeline. An extract from this is shown in Figure 1.

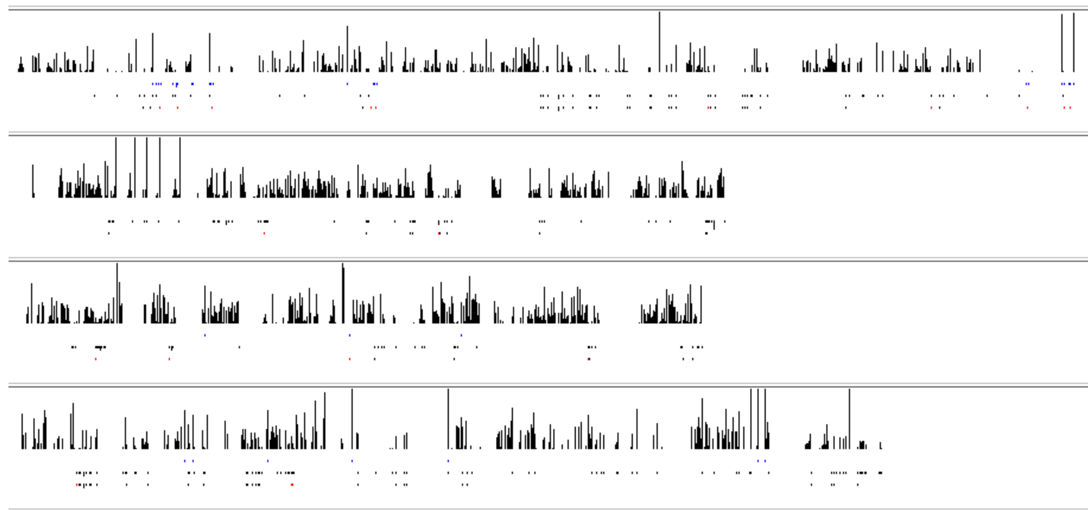

Figure 1: Visualization of solving processes

Different lanes depict the actions of different students (from top to bottom) on a linear time scale (left to right). Height of the black bars corresponds to the number of characters entered during a given time slice; small blue bars indicate the number of pastes (note that one student did not paste text at all). Two bottom rows on each lane show the number of program runs and the result of the run (a dot in the lowest row indicates a syntax or runtime error). We hoped to find some distinctive similarities in the action patterns of stronger or weaker students but instead we found that two very similar action patterns can result in a very high or a very low grade.

After finding some curious cases in our numeric summaries (for example, a well performing student was using the “Undo” command 10 times more often than other students), which were not explained by the action pattern visualization, we replayed the respective logs in Thonny, i.e., we observed how the actual text appeared in the editors and the shell. In most cases this helped us understand why data was different from average (e.g. the student mentioned previously was using undo to get rid of recently entered words with typos).

CONCLUSION

Former studies and our experiences both show that in the context of programming education it is worthwhile to collect fine grained data about learners’ actions during the programming process, and in principle this provides us with the same opportunities for giving feedback as commenting video logs or direct observation. At the same time, fine grained data probably create better opportunities for automatic analysis, for example, based on data mining. Some authors have proposed using data mining techniques to uncover the higher level meaning of a sequence of low level

user actions, but this remains a difficult problem. It would be of great help, if learners could easily mark the points in time, at which they completed one micro task (e.g., renaming a variable). It is not difficult to provide a keyboard shortcut for this, but most users likely need extra motivation for using this shortcut often enough to be useful. One possible reward for this key press could be saving the current snapshot of the code into a local history, which can be revised when necessary.

In addition to assessment of the learners' final program, (automatic) analysis of the solving process creates additional benefits for more adequate feedback and evaluation. Considering that in some cases students never submit their work (Vihavainen 2014), it would make sense to keep submission of logs separate from submission of the solutions for grading. It would not be the main purpose but analysis of the solving process could still help to identify illegal attempts in examinations, for example...

It is worth considering how to integrate analysis of programming logs more directly into the teaching and learning process. One approach would be integrating process analysis into automatic feedback systems, which usually analyse only the end result of a programming session. Such a system could, for example, warn the learner if it detects a possibly ineffective working pattern. Process data could be used in gamification – for example, the learner who writes a correct solution with very few deletes and corrections would receive a "Steady hand" badge. In labs, process data could be streamed into a visualization on the teacher's screen to make it easier to identify the students who need help.

We are convinced that automatic analyses of students' solving processes could provide various opportunities that have not been very thoroughly studied at this moment. Furthermore, Thonny seems to be a suitable environment for future experiments (and can be improved if necessary).Of course it needs to be specified which log data is useful to us. One way is to analyse what was the student doing before receiving an error message and how he/she responded to that. Also we have already experimented a little how to give students more specific instructions for solving their exercises and finding their mistakesbased on their programming logs.

This research was supported by the European Union through the European Regional Development Fund.

REFERENCES

Annamaa, A., (2015), Source code and installers of Thonny IDE, https://bitbucket.org/plas/thonny/

Cardell-Oliver, R. (2011), How can software metrics help novice programmers? In Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114 (pp. 55-62). Australian Computer Society, Inc.

Edwards S. H., Snyder J., Pérez-Quiñones M. A., Allevato A., Kim D., Tretola B., 2009, Comparing effective and ineffective behaviors of student programmers, ICER'09

Hosseini, R., Vihavainen, A., Brusilovsky, P. (2014), Exploring Problem Solving Paths in a Java Programming Course, University of Sussex

Negara, S., Vakilian, M., Chen, N., Johnson, R. E., Dig, D. (2012), Is it dangerous to use version control histories to study source code evolution? ECOOP'12

Negara, S., Codoban, M., Dig, D., Johnson, R. E. (2014), Mining fine-grained code changes to detect unknown change patterns, ICSE'14

Perkins, D., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989), Conditions of learning in novice programming, New Jersey

Vihavainen, A., Luukkainen, M., Ihantola, P. (2014), Analysis of source code snapshot granularity levels, SIGITE'14

Yoon, Y., Myers, B.A., Koo, S. 2013, Visualization of Fine-Grained Code Change History.

Biography

AivarAnnamaa was born in Tartu, Estonia in 1979. From 2001 till 2008 he worked as a programmer. At 2008 he started PhD in Computer Science in University of Tartu and is currently working as teaching assistant. His research interests are related to programming languages and teaching.

Annika Hansaluwas born in Kuressaare, Estonia in 1989. She is going to graduate as a mathematics and informatics teacher in 2015. Previous two years she has been teaching programming to beginners in University of Tartu and also one year in Tartu Tamme Gymnasium.

Eno Tonisson was born in Tartu, Estonia in 1969. He is a lecturer in the Institute of Computer Science of the University of Tartu. He graduated as a mathematics teacher in 1992 and received his master of science degree in mathematics in 1996 from the University of Tartu. His current research themes include didactics of programming; use of computer algebra systems in mathematics education; career choices of students of computer science and information technology.

He has worked as a mathematics teacher of secondary school for 8 years.

Copyright

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/3.0/