Automated Code Tracing Exercises for CS1

Seán Russell
University College Dublin
Dublin, Ireland
sean.russell@ucd.ie

ABSTRACT

The ability for students to read and comprehend code is a fundamental skill in computer programming. Relying on students to build this skill through typical programming assignments can lead to many persevering through trial and error rather than understanding. This paper describes Trace Generator, a work-in-progress application for generating automatically graded code tracing questions for Python and C programs. The fundamental principles behind this work are mastery through repetition and providing comprehensive and understandable feedback to enable students to learn from their mistakes. Feedback and reflections from the use of the generated questions with two introductory procedural programming classes (200 students) are also discussed. Analysis of student attempts suggests a willingness to complete quizzes multiple times until they achieved a satisfactory score (average final result of 91%).

CCS CONCEPTS

• **Social and professional topics** → **Computing education**; **CS1**;
• **Applied computing** → *Learning management systems*; *Computer-assisted instruction*.

KEYWORDS

code tracing, code comprehension, automatic grading, generated feedback

ACM Reference Format:

Seán Russell. 2022. Automated Code Tracing Exercises for CS1. In *Computing Education Practice 2022 (CEP 2022)*, January 6, 2022, Durham, United Kingdom. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3498343.3498347>

1 INTRODUCTION

The principal aim of an introductory programming course is typically to teach students to program. While educators will be aware that programming involves the combination of many interdependent skills, this may not be as evident to students.

When learning their first programming language, students must learn the sometimes unforgiving syntax of a new language, data types and their operations, the effects different statements have on variables and control flow. In parallel, students must develop strategies and techniques to solve problems using the language.

From an instructor perspective, a large well designed programming assessment will encourage the development and use of all

of the required skills. But from the perspective of some students the same assignment will be perceived as a requirement to produce code that gives the “right” answer. Students may also completely bypass the learning opportunity provided because these types of assessment are particularly vulnerable to plagiarism [10].

Research has shown that some students will resort to the *trial and error* approach when faced with completing these types of programming tasks [3]. Trial and error is unlikely to be successful in the long term, but can be quite effective for solving simpler CS1 and CS2 assignments. When the student reaches more advanced courses the use of such an approach could have negatively impacted the development of many of the key skills used in programming.

Assessing students on their code comprehension skills directly both provides an incentive for students to develop a key skill (increasing their grades) and emphasises the importance that we place on their abilities in this regard. Moreover, these assessments should provide an opportunity for the students to learn [8, 14].

In aiming to design code comprehension assessments, two features were considered essential. First, feedback should be immediate and comprehensive. Second, Students should have the opportunity to apply what they have learned from the feedback in repeated attempts (with different questions).

The nature of these features requires the construction of a question bank of sufficient size that students can repeat the assessments without encountering the same question twice. Manually building these questions would be prohibitively time consuming, as such an approach that generated questions was used.

The remaining sections of the paper detail related work, the design principles for questions, and preliminary evaluation.

2 RELATED WORK

The report of the McCracken group [10], highlighted the importance of program comprehension skills in program writing and prompted more investigation into the factors effecting performance in programming tasks. Lister et al. [9] investigated how students approach code tracing problems and categorised the “doodles” that the students produced in this process. These categories of doodle, such as Synchronized Trace, Trace, Number and Computation, were identified in student responses and the percentage of correct answers for each category were calculated. The more effective strategies that students use to solve program comprehension tasks were incorporated into the Trace Generator.

There has been research into the benefits of code tracing tutors on students abilities in program writing tasks [1, 7, 13]. While some studies have reported no benefit [1], others have found some or significant improvement in code writing abilities [7, 13].

A large number of tools and tutors have been developed to help students learn code comprehension skills. Some tools provide animated visualisation of the execution of programs [11, 13] with

This work is licensed under a Creative Commons Attribution International 4.0 License.

CEP 2022, January 6, 2022, Durham, United Kingdom
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9561-8/22/01.
<https://doi.org/10.1145/3498343.3498347>

internal state of the program displayed as the execution progresses under the control of the user. Some tools that provide this functionality are designed as an integrated development environment (IDE) which provides visualisation of program execution or a plugin that provides the same functionality to an existing IDE [2, 13]. More recently, tools of this nature are implemented for use on the Internet through a web browser [5, 15].

Beyond visualisation, many programs are designed as tutors to guide student learning through different aspects of program comprehension [4, 6, 12]. Helminen and Malmi [4] used the visualisations provided as a feedback mechanism to help students understand the automatically graded programming exercises they completed using the tool. Other tutors test students on other aspects using automatically generated questions [6, 12].

The Trace Generator is designed to reproduce the key features of these tutors (visualisations for feedback and testing on program comprehension) without requiring students to use an external tool/tutor. The generate questions can be imported into a question bank in Moodle without requiring any additional plugins or libraries. A key motivation was the simplicity of use by the students in the classes, where all other materials and assessments are hosted within the Moodle page of the course. That motivation was also heavily reinforced by the transition to online learning in the wake of the COVID-19 pandemic.

3 CODE TRACING QUESTIONS

3.1 Design of Questions

The Trace Generator is intended to produce questions in several formats. This is to complement the deepening knowledge and understanding of the students as they progress through a CS1 class.

The first format used in a course would be the single-line question. Students are required to trace the execution and consequences of a single line of code from a short program. Answering these questions correctly requires an understanding of the evaluation of expressions, focusing on the order that calculations are performed in, the result of individual calculations, the data type of that result and which line of code will be executed next.

Where the code contains an assignment or a function with side-effects (like `scanf`), the resulting changes to variables in memory must also be entered. This gives the student a structured way to practice the Computation and Number doodles described in [9]. An example of this type of question can be seen in Figure 1.

The order in which operations are executed is based first on operator precedence, and then from left to right. While this may not match the actual order of execution when optimisations are applied, it is consistent with the order that would be used if the variables or code literals were replaced by functions. Addresses used in these questions are abstract identifiers and are included for use with the `scanf` function.

The second format used in a course would be the multi-line question. Students are required to trace the execution and consequences of a sequence of statements or a short program. Answering these questions correctly requires an understanding of the control flow in the code and the changing values of variables as execution progresses. These questions do not require the detailed step-by-step break down of each line of code, but still require that the student

Variables Before Execution

Address	Name	Type	Value
0	a	int	12

Calculations

Order	Explanation	Code Executed	Result	Type of Result
1	Value loaded from variable	a	12	int
2	Literal constant in code	2	2	int
3	Multiplication	12 * 2	24	int
4	Assignment	b = 24	24	int

Variables After Execution

Address	Name	Type	Value
0	a	int	12
1	b	int	24

Figure 1: Questions displayed tracing the code `b = a * 2`

evaluates all of the expression in the code to choose the correct answer. This gives the student a structured way to practice the Synchronized Trace doodle described in [9].

Further formats are currently under development focused on two areas, (1) variable representation in the stack and (2) pointers/references and memory. A student that successfully completes code tracing exercises in all of the above formats would be required to have a good understanding of the underlying principles by which their code is executed.

3.2 Answering Questions

The questions generated by the tool require many responses from the students. In order to make the process of completing the questions less time consuming (and repeating assessments more palatable), many of the questions are multiple choice (dropdown boxes). Explanation, code executed, data type and line number questions are all multiple choice, while the result of expression and value of variable questions must be typed by the student. The multiple choice questions contain distractors generated based on the program code being traced in the question.

3.3 Feedback

In order to more easily establish the code tracing questions as learning events, sufficient feedback must be provided [14]. While no automated system is likely to provide feedback as directed and individual as an instructor or teaching assistant, thorough feedback is generated for each question. Each individual part of the question is highlighted as correct or incorrect and the correct answer is shown. In addition to this, general feedback is provided in the form of an animation accompanied by explanatory text.

Figure 2 shows a frame from the animated feedback of a single line question (the same question that is shown in Figure 1). A representation of the steps in executing the statement `b = a * 2` is shown. During the animation, each part highlighted in the order that they are evaluated and the results of each expression moving to the nodes in the tree where they are used. The animation is

Figure 2: Feedback for a single-line question variant

accompanied by text explaining the order that the expressions needed to be evaluated in and why.

Feedback for multi-line questions contains an animated flow-chart describing the control flow the accompanying code. The node in the flowchart and line of code are simultaneously highlighted as the program executes and a table containing the evaluated code being executed and a rudimentary symbol table is shown and updated at the same time.

3.4 Generating Questions

The Trace Generator is a command line application with files containing code, template parameters or input text passed as arguments. By default, a file containing a single multi-line question, a single-line question for each line of code, or both is produced ready to be imported in the Moodle XML file format. Other arguments allow naming and categorising questions, selecting the language, producing questions only for specific lines and choosing which type of question to generate.

Template values to generate different versions of the source code are supplied in a separate file. This was chosen over using a template engine for two reasons, first while basic variations are quite simple to implement using templating engines, more complex requirements can be quite difficult. Secondly, as the expected users are CS educators they should find it easier to write a short program to generate the required values than to learn a new language in order to be able to use the tool. Placeholders can be used to represent any tokens in the source code including variable names, operators, literal values, keywords and function names.

The generation of questions is done in two phases. First the source code is used to produce a JSON representation of the code and state as it is executed. In the second phase, this JSON representation is used to produce questions and feedback in the correct format for the Moodle virtual learning environment. This two part process is designed to enable easier extensibility in the form of alternate programming languages (currently only C and Python are supported) and alternate output formats for other learning environments and tutors (currently only Moodle is supported).

4 USAGE CONTEXT

Questions generated using the Trace Generator were used in two introductory procedural programming classes during the autumn semester of 2020. The first class contains 119 students majoring in

Figure 3: Average Scores Across Attempts

electronic engineering or Internet of Things (group 1) and was focused on the C programming language. The second class contained 91 students majoring in software engineering (group 2) and was focused on the Python programming language.

All students in both classes were non-native English speakers, attending their first semester at university and as a consequence of the pandemic were also studying remotely.

Previous instances of both classes used automated assessment of weekly programming tasks using a plugin in the Moodle virtual learning environment. In order to integrate the code tracing assessments seamlessly, the Trace Generator was designed to produce questions that could be used in the same environment rather than through an external tutor application.

The group 2 students were required to complete three separate quizzes containing a total of 16 single-line questions. In the first 6 weeks of class, while the group 1 students were required to complete four separate quizzes containing a total of 17 single-line questions. The multi-line questions were not sufficiently developed and tested and so were not used in the classes.

4.1 Student Engagement

The Trace Generator allows the creation of large banks of questions such that students could make multiple attempts at code comprehension assessments. Thus each attempt becomes a learning event where the student is able to correct any mistakes when facing a similar (but not the same) question. The average number of attempts across all quizzes in both classes was 2.1, while approximately 40% of students made only a single attempt for any quiz.

Figure 3 shows the average results of students in group 2 for each of the quizzes as the number of attempts increases. The solid lines represent the average quiz score for n^{th} and final attempts, while the dashed lines show the average quiz score for n^{th} but non-final attempts.

There are several interesting trends to note in this figure. Firstly, the average final submission for attempt x is always higher than the intermediary average for attempt $x - 1$, this shows that the students are improving with each attempt. Secondly, the rate at

which the intermediary averages climb steadily increases for each subsequent quiz, showing the students are becoming more familiar with the concept and achieving better scores more quickly. Thirdly, every student who made more than a single attempt achieved the highest grade possible (95% is the lower bound for an A+).

4.2 Student Results

The advent of the pandemic and remote learning necessitated large changes to the structure and assessment of the classes. These changes significantly impact the comparability of results from this cohort against previous cohorts. With those limitations in mind, the comparisons show promise.

The students in group 2 were required to complete a capstone assignment during the course that was roughly equivalent in difficulty to the previous years class. The average result in 2019 was 68%, while in 2020 the average result was 77%. Differences in the assignment, the mode of delivery, and a University mandated accommodation for student issues all likely contributed to this increase.

Exam results in both groups were slightly improved (1% for group 1 and 2% for group 2) but these were not statistically significant. Results in the code comprehension section of the exams were improved (4% for group 1 and 11% for group 2) but changes in the format of the questions mean the improvement may not be attributed only to the code tracing exercises.

4.3 Student Experience

While the students were not surveyed about the code tracing questions used in the course, several students did provide unprompted anonymous feedback about them. Two students commented on its usefulness stating “The code tracing helps me a lot to understand each steps of the process” and “The code tracing told me how a program worked in detail”. The students knew only that the questions were newly designed and that I wanted them to tell me when they encountered any problems or inconsistencies in the quizzes.

A third student commented on the format and feedback of the quizzes, this student was from group 2 which had less frequent and longer quizzes. The student stated “The code tracing unit is always too long, and something important is not highlighted”. At the least this suggests that shorter and more numerous quizzes should probably be preferred as they would require less time for students that have already achieved mastery and take less time to repeat for students yet to attain it. The second part of the comment was not accompanied with any more details and as the feedback was anonymous more could not be sought. At the least this suggests that more detailed responses should be sought about the feedback generated in the questions.

5 CONCLUSIONS, LIMITATIONS AND FUTURE WORK

This paper provides the authors experiences with the use of a new tool for generating program tracing questions. The fundamental principle is to encourage students to attain mastery through repetition of similar code tracing exercises. Evidence is provided that the students were willing to repeat the exercises, averaging 2.1 attempts for each quiz, and that their performance on these tasks improved (often to the point of mastery). However the conditions

under which the course was delivered and changes to assessment practices to compensate severely limit any conclusions that can be made about the effect on programming ability of the students.

The results are promising enough to suggest that a more thorough study should be completed with an aim to gather qualitative and quantitative data in order to more fully assess its use. The development of the Trace Generator is currently in its infancy, further development is planned to incorporate more programming languages and support the object-oriented programming paradigm, generate questions for different virtual learning environments and develop its use as a web based tutor.

REFERENCES

- [1] John R Anderson, Frederick G Conrad, and Albert T Corbett. 1989. Skill Acquisition and the LISP Tutor. *Cognitive Science* 13, 4 (1989), 467–505.
- [2] Aivar Annamaa. 2015. Thonny, A Python IDE for Learning Programming. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '15)*. ACM, New York, NY, USA, 343.
- [3] Stephen H Edwards. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '04)*. ACM, New York, NY, USA, 26–30.
- [4] Juha Helminen and Lauri Malmi. 2010. Jype - a Program Visualization and Programming Exercise Tool for Python. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 153–162.
- [5] Amruth Kumar. 2004. Web-Based Tutors for Learning Programming in C++/Java. In *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE '04)*. ACM, New York, NY, USA, 266.
- [6] Amruth N Kumar. 2005. Generation of Problems, Answers, Grade, and Feedback—Case Study of a Fully Automated Tutor. *J. Educ. Resour. Comput.* 5, 3 (sep 2005).
- [7] Amruth N Kumar. 2013. A Study of the Influence of Code-Tracing Problems on Code-Writing Skills. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '13)*. ACM, New York, NY, USA, 183–188.
- [8] Teemu Lehtinen, Aleksi Lukkarinen, and Lassi Haaranen. 2021. Students Struggle to Explain Their Own Program Code. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1 (ITiCSE '21)*. ACM, New York, NY, USA, 206–212.
- [9] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A Multi-National Study of Reading and Tracing Skills in Novice Programmers. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '04)*. ACM, New York, NY, USA, 119–150.
- [10] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education (ITiCSE-WGR '01)*. ACM, New York, NY, USA, 125–180.
- [11] Andrés Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. 2004. Visualizing Programs with Jeliot 3. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI '04)*. ACM, New York, NY, USA, 373–376.
- [12] Ruixiang Qi and Davide Fossati. 2020. Unlimited Trace Tutor: Learning Code Tracing With Automatically Generated Programs. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)*. ACM, New York, NY, USA, 427–433.
- [13] Juha Sorva and Teemu Sirkkiä. 2010. UUhistle: A Software Tool for Visual Program Simulation. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 49–54.
- [14] Leigh Ann Sudol-DeLyser, Mark Stehlik, and Sharon Carver. 2012. Code Comprehension Problems as Learning Events. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE '12)*. ACM, New York, NY, USA, 81–86.
- [15] Jun Zheng, Sohee Kang, and Brian Harrington. 2019. Immediate Feedback Collaborative Code Tracing. In *Proceedings of the Western Canadian Conference on Computing Education (WCCCE '19)*. ACM, New York, NY, USA.