Evaluation of an Automatic Generation System for Tracing Tasks Based on Textbook Programs

Tomohiro MOGI[a*], Yuichiro TATEIWA[b], Takahito TOMOTO[c] & Takako AKAKURA[d]

[a] *Graduate School of Information and Computer Science, Chiba Institute of Technology, Japan*
[b] *Graduate School of Engineering, Nagoya Institute of Technology, Japan*
[c] *Faculty of Information and Computer Science, Chiba Institute of Technology, Japan*
[d] *Faculty of Engineering, Tokyo University of Science, Japan*
* tomo1238hi@gmail.com

Abstract: In programming education, it is important not only to create programs but also to reflect on the created programs in order to understand them more deeply. To that end, the authors developed a system that automatically generates tracing tasks from input programs. In this study, we used programs from a textbook as input in order to confirm that the automatic problem and feedback generation system can generate tracing tasks automatically without any problems. The results of the experiment showed that the system was able to automatically generate tracing tasks from 53.85% of the input program elements, and it was confirmed that functions such as 'for' and 'if' statements as well as assignments to variables were supported without issue. However, orthodox syntax and functions such as 'while' and 'switch' statements, as well as batch assignments to structures and arrays, were not correctly implemented, so it will be necessary to correct these implementations in the future.

Keywords: Automatic Problem Generation, Program Trace Learning

1. Introduction

In recent years, the demand for programming education has increased. In general programming education, the teacher presents a task, and the learner creates a program to perform to the task. If a learner makes a mistake while creating the program, they can enhance their learning by identifying the mistake and correcting it. However, it is not always easy for learners to identify their mistakes or fix them.

Typical learning flows during a programming learning activity are shown in Figure 1. In normal programming learning, The learner receives the task (requirement) and creates a program satisfying the received task. The learners necessity to consider the expected results as they develop their programs. The learners (1) observe the execution results by running the program they have created, and then (2) detect the difference between the execution results and the expected results (correct answers). If the execution result differs from the expected result, the learner (3) looks back at the program code and corrects any mistakes, but it can be difficult to correct the program code simply by observing the execution results. Therefore, in advanced programming learning (i.e., trace learning: To consider the order in which programs are executed, the value of variables, output and conditional branching.), a debugger is used to trace the learner's code, making it easier to correct the mistake in the program code. In advanced programming learning, in addition to the normal programming learning flow, the learner (i) observes the execution process and the local results generated by the debugger, and (ii) detects the differences between the expected execution process

and the local results. In this way, trace learning makes it easier to modify the program compared with merely observing the execution results, thereby realizing more effective learning. However, because learners must learn how to use the debugger effectively, an environment in which feedback can be given is considered necessary.

We previously developed an automatic problem and feedback generation system (Mogi et al., 2023). By using this system to generate tracing tasks from programs created by learners, it is possible to quickly implement advanced programming learning using a debugger because (i)' Observe the process of execution and local results can be practiced and (ii)' Recognize differences from expected results can be visualized.

In this paper, we conduct experiments involving the automatic generation of tracing tasks, based on programs from a programming textbook (Shibata, 2021) in order to evaluate whether the developed system is able to generate appropriate tracing tasks.

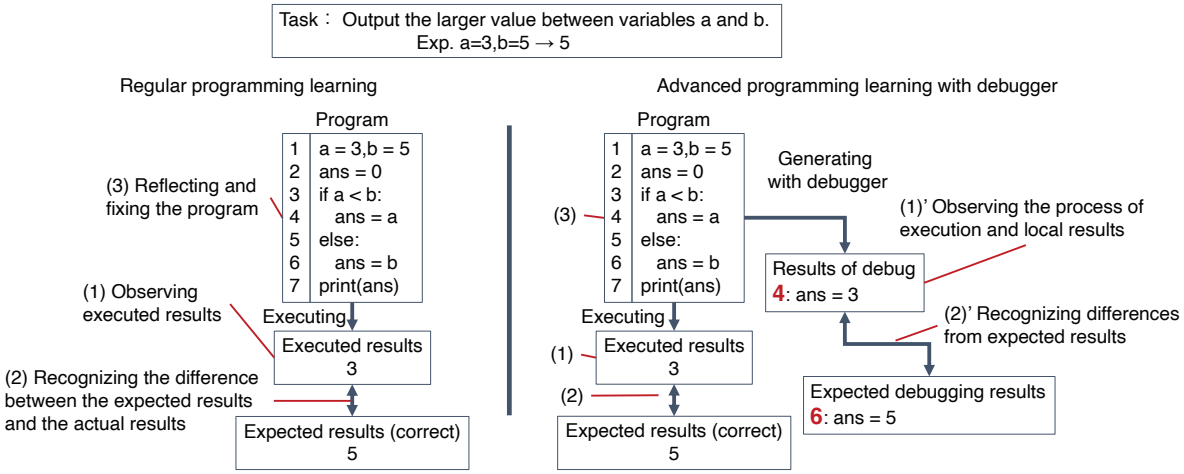

Figure 1. Learner's activities in programming learning

2. Related Work

In general programming education, emphasis is placed on the creation of programs, while activities involving reading programs are rarely conducted. However, in actual programming situations, the ability to read programs is also important. Also, when modifying (debugging) a program that you have created, it is necessary to understand the behavior of the program in detailed units. Several studies have proposed learning support aimed at developing the ability to read programs. These studies include support to estimate the functional requirements of programs (Kanamori et al., 2013; Arai et al., 2014), support to read the semantic coherence of programs (Watanabe et al., 2015), and support to make dedicated program tracing activities (Tomoto & Akakura, 2017). However, these studies dealt only with problems previously registered in the system and could not realize the advanced learning illustrated in Fig. 1.

3. Program Tracing Task

The program tracing task asks the learner about the order of the lines to be executed and the content of the instructions to be executed in each line, with the aim of making the learner consider the behavior of the program. In this task, the learner is first given the source code. Next, the learner is asked to select the first line in the source code to be executed. Then, to make the learner think about the instructions to be executed on that line, the learner is asked to describe the change in the value of a variable and the content of the output. In repetitive processing and conditional branching, the program processing flow changes depending on the success or failure of the condition, so the program is also required to describe the

success or failure of the condition. In the same way, the learner selects the next line and describes the behavior of the program, repeating this process until the end of the program. In the case of sequential structures, the order of the lines in the source code and the order of execution are the same, but in the case of iterations, conditional branches, or functions, line jumps can occur, and thus the task is to check whether the processing flow can be traced appropriately. Although experienced users might perform these tasks on a daily basis, novice users may be unaware of these tasks or might not be able to perform them properly and thus require guidance.

It is assumed that novice trainees may make mistakes in the order of execution and in the input of the contents of each line of processing. Although the program tracing task itself can be carried out on paper, it is possible to proceed without realizing where mistakes have been made when learning by oneself on paper or by consulting reference books. In such cases, the error may not be noticed until the end, or the error may be noticed by looking at the correct answer at the final stage. However, because the correct answer is known when the error is noticed, the opportunity to correct the error oneself is lost. Even if the teacher provides guidance on correcting the error in an educational setting, sufficient support cannot be expected, given the one-to-many nature of the classroom. Some ordinary debuggers are equipped with a trace function, which can be used for learning. Many of these functions allow tracing while stopping the program line by line, and may even allow the values of variables in each line and the screen output to be known. However, even when learning using a debugger, the correct answer is known immediately, which precludes the possibility of noticing errors. Therefore, in this study, we considered it important to have the learner perform the tracing task and then provide feedback.

4. Automatic Problem and Feedback Generation System

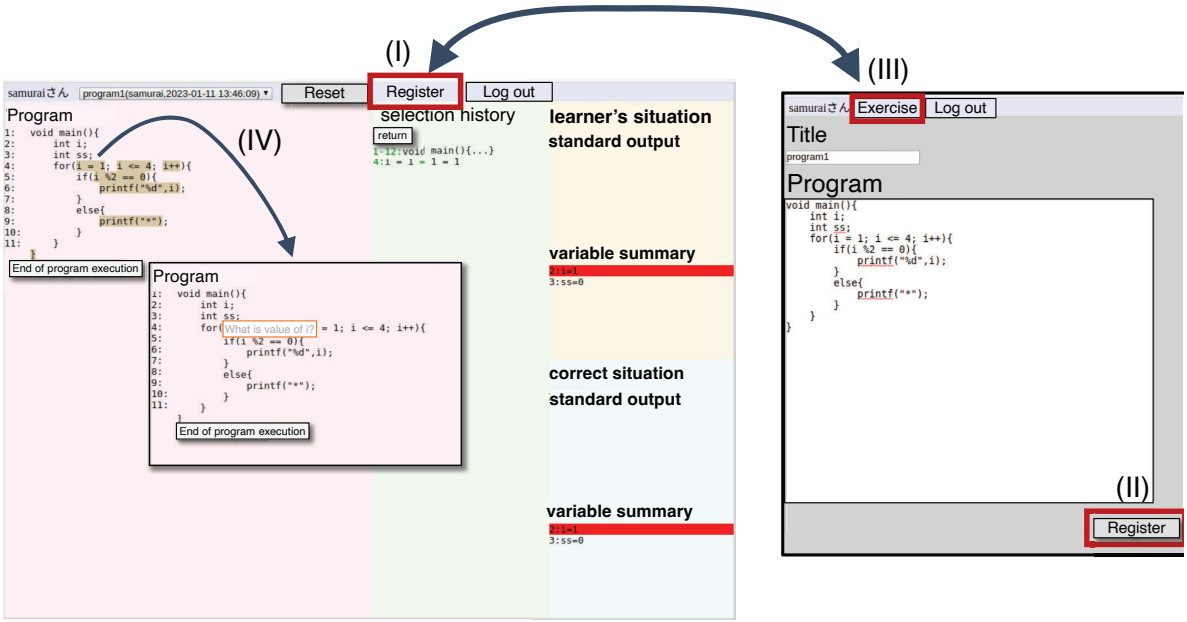

Figure 2. Interface of the Automatic Problem and Feedback Generation System

The system interface is shown in Figure 2. The system is largely divided into two phases: a phase for registering programs and automatically generating tracing problems, and a phase for working on the automatically generated tracing tasks. The learner first moves to the program registration screen by pressing the 'Register program' button shown in Fig. 2(I) and begins the first phase (registering a program and automatically generating tracing problems). On the program registration screen, the title and program are entered and the tracing problem is automatically generated by pressing the 'Register' button (Fig. 2(II)). After

registering the problem, the learner moves to the tracing screen by pressing the ‘Exercise’ button (Fig. 2(III)) at the top of the screen.

The learner then enters the second phase (working on the automatically generated tracing tasks). The program and the term-by-term options are presented on the left-hand side of the tracing screen. The learner considers which terms from the presented program are to be executed and in which order. If clicking on a selectable part changes a variable in that part, the learner will be asked to enter its value, using the keyboard (Fig. 2(IV)). For example, if the learner selects the term “*i = 1*”, they would be asked “What is the value for i?” Because the value of i in this case is 1, they must enter “1”. In the case of an ‘if’ statement, both the value of the conditional judgement and the judgement result (T or F) are required, while in the case of a ‘for’ statement, the value and judgement result are required for the initial value, the conditional judgement, and the increment.

The system has a function for displaying the values of variables as feedback to the learner. The value based on the learner’s answer is displayed at the top center of the screen, and the value in the case of a correct answer is displayed at the bottom center of the screen. The system also generates feedback automatically by analyzing the program, so that the value of the correct answer can be displayed without the teacher having to enter the correct answer beforehand. The learner aims to correct errors by observing the difference between the expected and executed results.

Here, we describe the flow when the learner’s answer is correct is explained, using the program shown in the Fig. 2 as a concrete example. The learner first considers which part of the program is executed first. The choices are “"i = 1"”, “"i <= 4"”, “"i++"”, “"i %2 == 0"”, “"print(“"\%d"”,i)"” and “"print(“"*"”)"”. The first part to be executed is considered to be “"i = 1"”, so the learner clicks on that part. Then, as shown in (IV), the system asks “"What is the value for i?” Therefore, the learner enters ‘‘1’’ on the keyboard and presses the Enter key. This operation corresponds to the answer “"1 is assigned to the value of i"” and is therefore indicated as “"i=1"” in the variable summary of your the situation. Because this answer is equal corresponds to the actual system flow (correct answer), the variable summary of the correct situation changes in the same way.

Next, we describe the flow when the learner’s answer is incorrect. For example, if the learner responds to the question “What value goes in i?” by entering ‘2’, then the variable overview of the situation will show “*i=2*”. The fact that this display differs from the variable summary of the correct situation, “*i=1*”, allows the learner to realize their error. If the learner answered “*print(“*”)*” for the part that is executed first, a “*” would appear in the standard output of the situation, drawing attention to error.

5. Evaluation

This paper describes an experiment conducted to investigate whether the developed automatic problem and feedback generation system is capable of generating appropriate tracing tasks. In the experiment, tracing tasks were automatically generated, using a program from a programming textbook (Shibata, 2021) as input.

5.1 Method

Because there are 13 chapters in the textbook (Shibata, 2021), a total of 39 programs, three from each chapter, were extracted for the experiments. In addition, characters and functions that are not supported by the system were modified. First, the extracted programs contained multibyte characters, but this system does not support 2-byte characters, so they were changed to 1-byte characters. Next, because the system does not support the *scanf* function, it was changed to an appropriate numeric or character assignment. We manually imported the program with these changes into the system and confirmed that the tracing task could be generated automatically.

5.2 Results

Table 1 summarizes the elements of the programs used and whether or not tracing tasks could be generated automatically for these elements.

Table 1. Elements of the programs used and whether tracing tasks could be generated automatically

Program elements	Number of programs included	Number of generated questions
printf function	39/39 (100%)	21/39 (53.85%)
puts function	13/39 (33.33%)	6/13 (46.15%)
sizeof function	1/39 (2.56%)	1/1 (100%)
atoi function	1/39 (2.56%)	1/1 (100%)
fclose function	3/39 (7.69%)	0/3 (0%)
user-defined function	12/39 (30.77%)	4/12 (33.33%)
four arithmetic operations	14/39 (35.9%)	11/14 (78.57%)
ternary operator	4/39 (10.26%)	2/4 (50%)
assignments to variables	23/39 (58.97%)	13/23 (56.52%)
Assignments to arrays	7/39 (17.95%)	1/7 (14.29%)
struct	3/39 (7.69%)	0/3 (0%)
if statement	10/39 (25.64%)	3/10 (30%)
for statement	14/39 (35.9%)	5/14 (35.71%)
while statement	9/39 (23.08%)	0/9 (0%)
switch statement	2/39 (5.13%)	0/2 (0%)
all	-	21/39 (53.85%)

Tracing tasks could be automatically generated for only 53.85% of the program elements. The reasons for this are thought to be due to several unsupported syntax and functions. For example, the developed system does not support ‘while’ and ‘switch’ statements as well as batch assignments to structures and arrays. Table 1 also shows that tasks for elements that include these statements cannot be generated automatically. Tracing tasks for functions such as for statements (35.71%) and if statements (30%), as well as assignments to variables (56.52%), had a low generation rate and at first glance appeared to be unsuccessful. This is because unsupported syntax and functions such as switch statements and array batch assignments tend to occur at the same time in these syntax and functions. Therefore, it was confirmed that the cause of failure was due to the currently unsupported syntax. And that, if did not contain unsupported syntax and functions was 100% generating of the programs successfully generated. It was also confirmed that tracing tasks for programs with inappropriate line breaks and indentation could also be generated automatically without problems. For these reasons and the fact that experiments in Mogi et al. (2023) confirmed the learning effect, we believe that the this system has a certain value. However, to realize debug-style learning using this system, it is necessary to be able to handle orthodox syntax and functions such as ‘while’ and ‘switch’ statements as well as batch assignment to structures and arrays. We do not consider the syntax and functions to be beyond the scope of the system, but rather the problem is that they are not implemented correctly. Therefore, we plan to correct their implementation in the future.

6. Conclusion

In programming learning, it is important to acquire the ability to understand how a program works (tracing ability). We previously developed a system (Mogi et al., 2023) that realizes this aim as a learning method for acquiring tracing ability.

In the present paper, we conducted experiments to confirm that the system can be used for learning with debuggers. The experiments used programs from textbooks as input and investigated the degree to which the system could automatically generate tracing tasks based on these programs. The results showed that tracing tasks could be generated for 53.85% of the program elements. However, it was confirmed that orthodox syntax and functions such as ‘while’ and ‘switch’ statements as well as batch assignment to structures and arrays were not correctly implemented.

This study aims to improve the programming skills of learners. In particular, we focus on the ability to trace a program by statement. However, we consider it necessary to start working on other areas in the future.

When expert programmers debug a program, they first recognize the program in large block units and roughly identify the cause of problems by tracing. Then, by tracing in more detail, they identify and fix the cause of the problem. Expert programmers perform hierarchical debugging in this way, but this system does not support hierarchical debugging. However, we consider that the skill to recognize a program as a large block and debug it is important and should be considered in the future.

Elsewhere, The current system only uses values embedded in the code by the learner for tracing. However, for the learner to understand the behaviour of the program, the test data(input) needs to be appropriate. For example, in the problem "output the larger value of variables a and b", it is not possible to judge that it is an appropriate program with only "data in which b is larger than a", but at least "data in which a is larger than b" or "data in which a and b are the same size" etc. are considered necessary. Therefore, it is necessary to consider test data in the future.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Numbers JP22K12322 and JP20H01730.

References

Arai, T., Kanamori, H., Tomoto, T., Kometani, Y., & Akakura, T. (2014). Development of a learning support system for source code reading comprehension. Lecture Notes in Computer Science, 12–19. https://doi.org/10.1007/978-3-319-07863-2_2

Kanamori, H., Tomoto, T., & Akakura, T. (2013). Development of a computer programming learning support system based on reading computer program. Human Interface and the Management of Information. Information and Interaction for Learning, Culture, Collaboration and Business, 63–69. https://doi.org/10.1007/978-3-642-39226-9_8

Mogi, T., Tateiwa, Y., Tomoto, T., & Akakura, T. (2023). Proposal for Automatic Problem and Feedback Generation for use in Trace Learning Support Systems. Lecture Notes in Computer Science, 310–321. https://doi.org/10.1007/978-3-031-35129-7_23

Shibata, B. (2021). Shin Meikai Cgengo Nyumonhen [New lucidity C Programming Language Introductory] (Vol. 2). SB Creative.

Tomoto, T., & Akakura, T. (2017). Report on practice of a learning support system for reading program code exercise. Human Interface and the Management of Information: Supporting Learning, Decision-Making and Collaboration, 85–98. https://doi.org/10.1007/978-3-319-58524-6_8

Watanabe, K., Tomoto, T., & Akakura, T. (2015). Development of a learning support system for reading source code by Stepwise Abstraction. Lecture Notes in Computer Science, 387–394. https://doi.org/10.1007/978-3-319-20618-9_39