Chen, W. et al. (Eds.) (2017). Proceedings of the 25th International Conference on Computers in Education.
New Zealand: Asia-Pacific Society for Computers in Education

Learning Environment for Recursive Functions by Visualization of Execution Process

Raiya YAMAMOTO[a*], Yasuhiro ANZAI[b], Satoru KOGURE[b], Yasuhiro NOGUCHI[b], Koichi YAMASHITA[c], Tatsuhiro KONISHI[b] & Yukihiro ITOH[d]
[a]*Graduate School of Science and Technology, Shizuoka University, Japan*
[b]*Faculty of Informatics, Shizuoka University, Japan*
[c]*Faculty of Business Administration, Tokoha University, Japan*
[d]*Shizuoka University, Japan*
*yamamoto.raiya.15@shizuoka.ac.jp

Abstract: In programming learning, it is difficult to imagine the behavior of a recursive function. This difficulty stems from the following: (1) learners have to manage different instances of functions with the same name, (2) they cannot understand the execution process of passive flow, and (3) novice learners cannot manage unnecessary portions as a black box. Furthermore, it is desirable for authors of teaching materials to customize the visualization of each instance individually. In this study, we have extended a program visualization tool (TEDViT) to be able to visualize recursive functions by satisfying the above issues. We conducted an evaluation of our method and obtained positive results.

Keywords: Visualization, programming education, recursive function

1. Introduction

In programming learning, it is thought to be difficult to imagine the behavior of a recursive function. We believe that this difficulty exists due to the following three reasons: (1) learners must manage different instances of a function, (2) they cannot understand the execution process of passive flow, and (3) novice learners are unable to manage unnecessary portions as a black box.

For (1), a recursive function calls the same function recursively; therefore, learners need to manage different instances of the same function. It is difficult to reproduce the behavior of each instance and to retain these behaviors in their brains, because these instances are produced from the same function when recursive calls are executed. To address this, we consider it is necessary to visualize the execution processes of instances with a bird's-eye view **(Common Requirement 1)**.

Regarding (2), Sholtz et al. revealed the source of the difficulty of imagining a recursive function's execution process: the control flow, known as "passive flow" (Scholtz, et al., 2010). In a recursive function, a new instance of the function is called, with control repeatedly passing forward to the newly created instance (active flow). Then, the flow reaches a termination condition of the recursive calling, after which the control and a value are repeatedly returned to the calling function. This flow is known as the passive flow. Scholtz et al. mention that the difficulty of the passive flow stems from novice learners misunderstanding that execution process of a recursive function ends immediately when termination conditions are satisfied; Shortly, they never imagine passive flow. To address this, we consider it is necessary to visualize behaviors by suggesting passive flow **(Common Requirement 2)**.

In terms of (3), a recursive function's execution process tends to be complex, due to increasing instances of the same function, and the need for learners to recognize passive flow. To prevent this, a method for hiding the called functions' behaviors and simplifying the entire process is required, which we refer to as "black boxing." To achieve black boxing, we consider it is necessary to visualize behaviors using a situation in which called functions' behaviors are concealed and they only return output **(Common Requirement 3)**.

Furthermore, it is desirable to customize the visualization of the behaviors of each instance of recursive functions individually, because data structures differ in program algorithms and the data

managed and received by called functions vary. For this reason, it is preferable for authors of teaching materials to be able to customize the ways of visualization **(Individual Requirement)**.

Various tools have been developed to visualize programs' behavior with animation. Jeliot3 (Moreno, et al., 2004) is a tool for visualizing Java program behaviors. However, this tool does not satisfy the Individual Requirement and Common Requirements 1 and 3. SRec (Velázquez-Iturbide, et al., 2008) is a visualization tool to assist understanding recursive functions. However, this tool does not satisfy Common Requirement 3 and the Individual Requirement. ANIMAL (Rößling and Freisleben, 2002) and TEDViT (Kogure, et al., 2014) are tools to visualize program behavior using animation. Teachers can setup the visualization with scripts or rules, so they satisfy the Individual Requirement. Moreover, teachers can write scripts and rules that satisfy the Common Requirements. However, these scripts and rules are not specialized for recursive functions, so a great deal of time is required to write scripts and rules that satisfy the Common Requirements. The Common Requirements are common features for managing recursive functions. Hence, these features should not be written with scripts and rules, but rather embedded in systems. In comparing TEDViT and ANIMAL, the cost of writing TEDViT rules is less than that of ANIMAL scripts, so we extend TEDViT.

Consequently, the purpose of our research is to extend TEDViT to be capable of managing the Common Requirements for recursive function visualization. In this paper, we report on our process of extending TEDViT, and the evaluation thereof.

2. Supplemental Features to Apply Learning Assistance for Recursive Functions

2.1 Basic Function of TEDViT

Teachers can customize the visualization of unique algorithm concepts and program behaviors with TEDViT (The interface is shown in Figure 3). The figures are reproduced in an area called the status of target world (STW). To use TEDViT, teachers should write rule sets (hereinafter, "drawing rules"). Drawing rules allow TEDViT to display data objects with effects. TEDViT supports objects: variables, arrays, arrows indicating relationships between data, and so on, as well as visual effects: colors representing emphasized parts and balloons showing explanatory texts, among others. Learners can trace program steps with these visualizations.

2.2 Visualization of Instances of the Same Function

As mentioned in the introduction, recursive functions call themselves recursively and create instances from the same function, so making it difficult for learners to manage their behaviors using only their brains. For this reason, TEDViT should show the functions comprehensively, and assist learners in focusing on a particular function. However, TEDViT currently can visualize only one function at a time. Therefore, TEDViT needs to be expanded to be able to generate a new drawing area on the STW and visualize objects corresponding to variables, for example, on the generated drawing area when recursive calling is executed. Moreover, to know which recursive function calls a function recursively, TEDViT needs to visualize drawing areas as a tree structure so that learners can see which area is generated recursively and can know their calling/called relations. We refer to the drawing area as a "function drawing area" (hereinafter, "F-Area"). An example of the F-Area is shown in Figure 3.

2.3 Visualization of Passive Flow

As mentioned in the introduction, it is difficult for learners to recognize passive flow, as they misunderstand recursive calling ending when the termination condition is reached. Furthermore, in most cases, passive flow exhibits a complex process. Therefore, it can be challenging to imagine the passive flow process of data returning to the calling function correctly. For this reason, TEDViT needs a new visualization method so that learners can understand the passive flow process.

As mentioned in 2.2, the extended TEDViT can visualize F-Areas by connecting them in a tree structure, and learners can thereby see the data flow among functions. Figure 1 demonstrates how the passive flow is observed. By observing behaviors from the structure, learners can check the values that

the calling function obtains as the output of the called functions, and understand the flow of returning output of the called functions. We refer to these functions visualized as a tree structure the "recursive function tree." However, learners may not realize the existence of passive flow using the recursive function tree alone. The solution to managing this problem is described in 2.5.

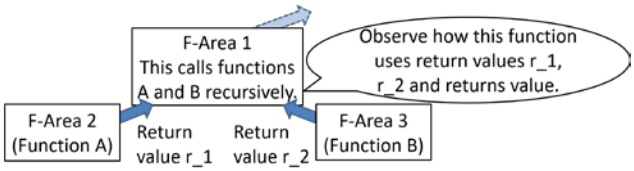

Figure 1. Observation of Passive Flow

2.4 Visualizing Recursive Calling with Black Boxing

When writing recursive functions, we don't imagine the called function behaviors, but code the process considering the return values obtained from the called functions. By black boxing calls to subordinate functions, we can hide and simplify complex function calls, and observe an overview of the recursive function execution process. However, it should be recognized that black boxed called functions behave in the same manner as calling functions. If function calls are black boxed and we do not consider concrete behavior, this means hiding recursive features and termination conditions. For this reason, in order to visualize recursive calling effectively, the following requirements should be satisfied; The ability to observe recursive calls by means of their being black boxed **(Requirement for black boxing 1)**. The ability to observe recursive calls by disabling black boxing from the top to bottom function **(Requirement for black boxing 2)**. The ability to enable learners to consider the termination condition **(Requirement for black boxing 3)**. To achieve requirement for black boxing 1, we add the process that behaviors of called functions is hidden and the behaviors are skipped until the called function returns a value (hereinafter, "1-step recursive calling", the left side of Figure 2). When subordinate functions are called, the behaviors of the called functions are black boxed and learners observe only results from the called functions. Learners also can disable black boxing by clicking the black boxed F-Area (the right side of Figure 2), and check black boxed behaviors recursively. Learners observe the behavior of a black boxed subordinate function, and check that these functions behave in the same way, which is one characteristic of recursive functions (this satisfies requirement for black boxing 2). When leaners have noticed that the recursively called functions behave in the same way by repeating the disabling of black boxing, and want to stop the disabling, an instance of the calling function satisfying the termination condition is opened. Learners can then understand that there is a point in the execution process where recursive calling is terminated (this satisfies requirement for black boxing 3).

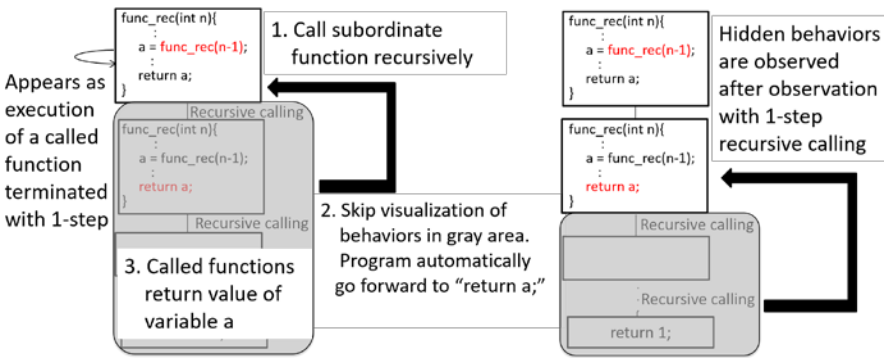

Figure 2. 1-step Recursive Calling (left side) and Disabling Black Boxing (right side)

2.5 Learning Scenario with TEDViT

We define an ideal learning scenario as steps I to IV described below. Steps I to III are for the learning in 2.4, while step IV is for the learning in 2.3; however, the system does not force learners to follow the scenario. It is preferable that the system is designed to let learners refer to messages to learn actively.

All recursively called functions are black boxed, and only first-executed functions are visualized during first step. Learners check an overview of the entire process without considering subordinate functions (**Step I**). Learners observe the recursive function characteristic that functions behave in the same way (**Step II**). Learners observe the termination of the recursive calling, in order to understand when the termination condition is satisfied (**Step III**). Learners check the passive flow by observing the processes of the functions that are black boxed in step I return outputs (**Step IV**).

3. Learning Flow Using the System

Learners learn recursive functions with C language. The system analyzes a source code and visualizes a program's behavior. We extended TEDViT with the features in section 2. The interface of the system consists of four areas: the source code, message, control button, and STW areas (Figure 3). Learners can see the source code in the source code area. In the message area, learners can read messages suggesting what to do next or stating what type of processing was executed in the recursive calling. Learners can select which point in the execution process to be visualized by buttons in the control button area, and a process tree of recursive functions is produced in the STW area. Figure 4 (A) displays a situation where the first-called function calls functions recursively. The called function's behaviors are black boxed with 1-step recursive calling (learning scenario **Step I**). Learners can observe black boxed recursive functions by clicking on their F-Areas. Figure 4 (B) shows a situation in which the recursive calling reaches the termination condition by repeatedly disabling black boxing (**Step II**). Then, a message suggests terminate condition and passive flow (**Step III**). Finally, learners check passive flow by observing the passing of control back from terminated functions (**Step IV**) (Figure 4 (C)).

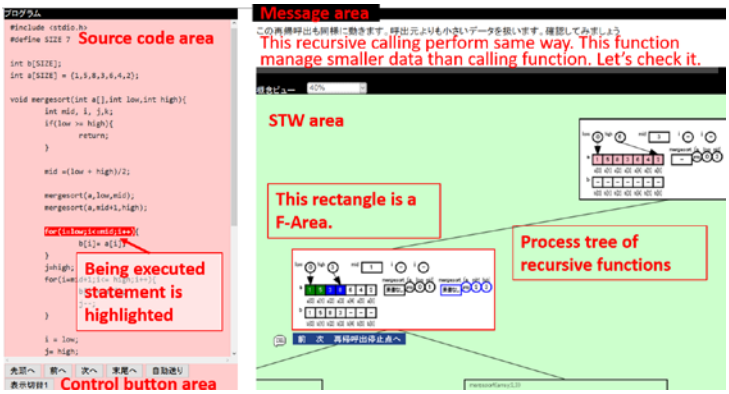

Figure 3. System Interface

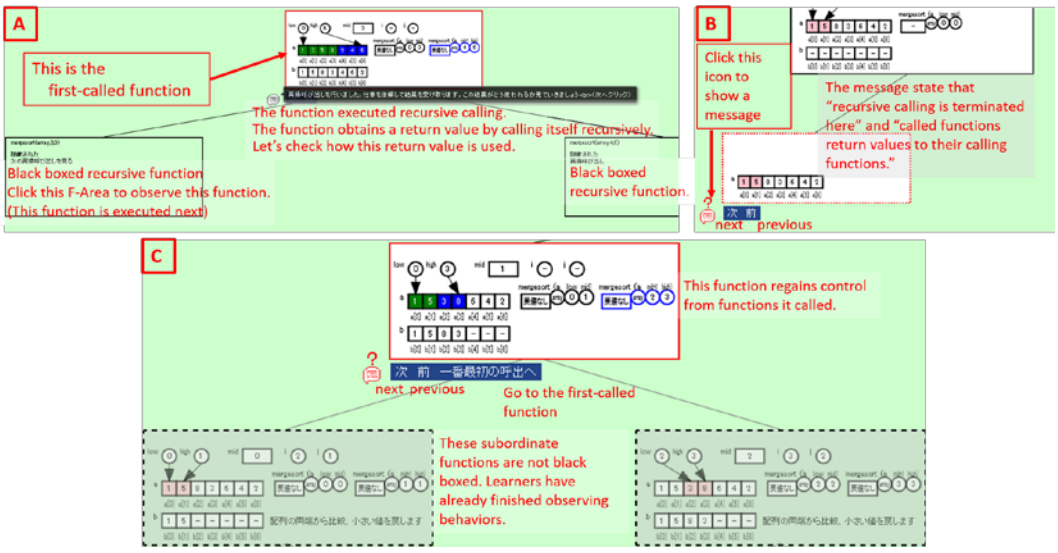

Figure 4. (A) 1-step Recursive Calling with Black Boxing,
(B) Recursive Calling Reaching Termination Condition, (C) Observation of Passive Flow

4. Evaluation

4.1 Method of the Experiment

In order to evaluate whether the extended TEDViT is effective for learning recursive functions, we tested the three hypotheses below; The new TEDViT is effective for learning the execution process of a recursive function (**Hypothesis 1**). It is effective to visualize recursive calling and dataflow as a tree structure when TEDViT visualizes recursive calling behaviors (**Hypothesis 2**). Visualizing simplified behaviors of recursive calling by means of black boxing is effective for learning recursive calling (**Hypothesis 3**). We furthermore compared TEDViT with ANIMAL, because their methods are similar.

Subjects were eight students who have learned the C language and recursive functions. Subjects were divided into the control group and the experimental group based on the pretest result. Each group has four subjects. The control group learned recursive functions with ANIMAL, and the experimental group learned with the extended TEDViT. Subjects attended a brief review lecture on recursive functions at the beginning of the experiment, then took a pretest. Subjects in both groups learned with a learning support system, and afterwards they took a posttest. They learned recursive functions with the twice: Fibonacci sequence and merge sort. Finally, the subjects answered a questionnaire. The control group used ANIMAL, which includes visualization samples of both the Fibonacci sequence and merge sort. To confirm hypotheses 2 and 3, visualization with the tree structure and black box was included only when subjects learned the Fibonacci sequence.

We set the same questions for the pretest and posttest. The test consisted of three parts, each of which contained two questions, where the first part was about the Fibonacci sequence and the second about merge sort. The final part was to observe whether learners could apply the knowledge they gained to the system they used. The questions in the first and second parts were about programs with different parameters that the students observed with the systems, and we checked whether they successfully learned by using the systems according to the test results. In each part, the first question was to evaluate whether the students could explain recursive function behaviors with the concept of the black box, and the second was to evaluate whether they answered the return value and order of the return value being output (in short, it checked whether they understood passive flow or not).

Since the pretest and posttest contained the same questions, we could expect that all subjects would obtain higher scores in the posttest. We verified the learning effectiveness by checking the improvement ratio of the scores. The evaluation criteria for each question are as follows. Criteria for the first question: (1) Subjects can see recursive calling as a black box, (2) They can answer which value is returned from the black box, (3) They can explain how to calculate using the value from the black box and which value the function in question returns to the function that called it. Criteria for the second question is that whether they can illustrate the passive flow with returned values of each called function.

4.2 Results

Figure 5 illustrates the average pretest and posttest scores (maximum score is 100) and increase ratio. Experimental group got higher score at the posttest than the control group. The experimental group also marked higher increase ratio.

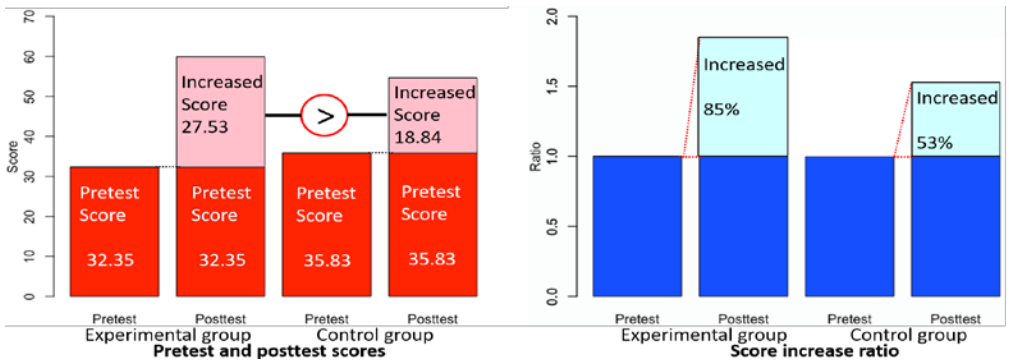

Figure 5. Average Pretest and Posttest Scores (left side) and Increase Ratio (right side)

Table 1 displays the questionnaire results. The control group's scores for Q2 and Q3 are lower than those of the experimental group, which we believe resulted in the control group obtaining lower scores on the posttest. The control group's score for Q4 is higher than that of the experimental group, which we consider being due to not only TEDViT, used by the experimental group, but also ANIMAL, used by the control group, including the feature of visualization with a tree structure.

Table 1: Questionnaire results

Contents of questionnaire (best score is 5.00pt)	Experimental group	Control group
Q1: It was easier to understand recursive functions than by learning only with a classroom lecture.	**4.25**	3.75
Q2: You could understand the idea of the black box.	**3.88**	2.88
Q3: You could understand passive flow.	**4.50**	3.13
Q4: You thought visualization of the recursive function execution process was understandable.	4.00	**4.50**

The results indicate that the extended TEDViT can provide effectiveness for learning the recursive function execution process, because the experimental group obtained higher scores and a more increased score rate than the control group. Furthermore, it is suggested that visualizing the recursive function execution process with a tree structure is effective, and it is necessary to grasp the idea of the black box and passive flow for understanding recursive functions.

5. Conclusion

We expanded TEDViT in order to achieve learning of recursive functions with the flexible visualization TEDViT originally applied. The results of our evaluation suggest that the proposed method is effective; however, potential points of improvement for the learning scenario were also revealed. Furthermore, more subjects are required to obtain more convincing results. Moreover, the current TEDViT is still not capable of managing structures. Since recursive programs include aspects such as tree structures, we are planning to expand TEDViT further to be able to manage structure.

Acknowledgements

This research was supported by the Japanese Grant-in-Aid for Scientific Research (B) Grant Number JP24300282, (C) Grant Number JP16K01084 and Grant-in-Aid for Young Scientists (B) Grant Number JP15K16104.

References

Moreno, A., Myller, N., Sutinen, E. (2004). Visualizing programs with Jeliot 3, *Proceedings of the Working Conference on Advanced Visual Interfaces*, 373-376.

Rößling, G., Freisleben, B. (2002). ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation, *Journal of Visual Languages & Computing,* 13(3), 341-354.

Kogure, S., Fujioka, R., Noguchi, Y., Yamashita, K., Konishi, T., Itoh, Y. (2014). Code Reading Environment by Visualizing both Variable's Memory Image and Target World's Status, *Proceedings of International Conference on Computers in Education*, 343-348.

Scholts, T. L., Sandrs, I. (2010). Mental Models of Recursion: Investigating Students' Understanding of Recursion. *Proceedings of the 15thAnnual Conference on Innovation and Technology in Computer Science Education,* 103-107.

Velázquez-Iturbide, J. Á., Pérez-Carrasco, A., Urquiza-Fuentes, J. (2008). SRec: an animation system of recursion for algorithm courses. *Proceedings of the 13thAnnual Conference on Innovation and Technology in Computer Science Education,* 225-229.