Creative Education, 2023, 14, 1945-1964
https://www.scirp.org/journal/ce
ISSN Online: 2151-4771
ISSN Print: 2151-4755

Evaluating Prospects in Programming with Features in Graphic Function Use

Tomoharu Kobayashi[1], Hiromitsu Shimakawa[1], Fumiko Harada[2]

[1]Graduate School of Information Science and Engineering, Ritsumeikan University, Shiga, Japan
[2]Research Organization of Science and Technology, Ritsumeikan University, Shiga, Japan
Email: tomoharu@de.is.ritsumei.ac.jp, simakawa@cs.ritsumei.ac.jp, harada@de.is.ritsumei.ac.jp

How to cite this paper: Kobayashi, T., Shimakawa, H., & Harada, F. (2023). Evaluating Prospects in Programming with Features in Graphic Function Use. *Creative Education, 14,* 1945-1964.
https://doi.org/10.4236/ce.2023.1410124

Received: July 17, 2023
Accepted: October 14, 2023
Published: October 17, 2023

Abstract

In this paper, we propose a quantitative evaluation method of students' thinking in group learning. Thinking evaluation will become increasingly important in programming education in Japan. However, it is impossible for instructors to single-handedly evaluate their students' thinking at the same time. It is necessary to provide a quantitative evaluation method that can be applied to a variety of educational situations in order to help instructors. We define coding vectors based on students' source code that will serve as an indicator of evaluation. Moreover, we judge students' prospects through a 3-step analysis with their coding vectors. We analyzed coding vectors for 22 participants obtained through a task experiment. We evaluated students' thinking from three perspectives: visualization, distance, and direction. As a result, all three ways had the ability to grasp students' thinking content. Coding vectors allow us to comprehensively judge students' coding steps and their prospects. In this paper, we discuss the expressive power of coding vectors for coding content, and task settings appropriate for them.

Keywords

Programming, Computational Thinking, Source Code, Figure Drawing, Feature Vector, PCA

1. Introduction

From the 2020 academic year onward, education in Japan has been based on the new government course guidelines. As a result, programming education has become compulsory in primary and secondary education and has gained importance. In the programming education guide published by the Ministry of Education, Culture, Sports, Science, and Technology, the term "programming thinking" is used as a keyword. This programming thinking is a mindset and ability that can be developed through programming education. In a word, it is the ability to construct procedures. The guide says that one major goal is to develop programming thinking. A method to grasp students' thinking about coding is important.

However, there are two educational problems with it. First, it is not clear how to evaluate the thinking process. This is due to the lack of experience in the new educational system. Second, it is difficult to evaluate thinking ability in group learning. To evaluate thinking skills, for example, simple calculations and correct memorization of theorems are not enough. As in a proof problem, the process of how to use them to derive the necessary matter is important. However, it is physically impossible for a teacher to observe the thinking processes of more than ten students at a time. It is also tough to evaluate tasks with different characteristics using a common indicator, as there is currently no standard. The subjective judgments of instructors and the impressions of students are different standards for each individual, and they may change with one's mood. A new evaluation method in programming education is required to solve the two problems in Japan.

Programming thinking mentioned above, which is an educational goal in Japan, is referred to as computational thinking in other countries which have started programming education earlier than Japan (Wing, 2006). In countries with advanced programming education, there are many studies that relate programming to computational thinking. Tedre et al. provide some notes on future research based on the history of computational thinking (Tedre & Denning, 2016). Sun et al. showed positive effects of programming on computational thinking skills through a meta-analysis (Sun et al., 2021). They also provided a discussion of good design factors of programming education. Studies from overseas such as these studies indicate a high interest in thinking ability about programming in primary education.

Tikva et al. and Fagerlund et al. present a review of the literature related to computational thinking (Tikva & Tambouris, 2021; Fagerlund et al., 2021). These studies point out a lack of indicative research examples in the evaluation of computational thinking and the causes of the lack. One of the causes is the difficulty of grasping the reality of the thinking process itself. Another cause is that the results of each study may be dependent on regional differences and curriculums. Since programming thinking is based on the idea of computational thinking, problems related to computational thinking are common. Therefore, solving these two causes is necessary to tackle the educational problems in Japan.

This paper presents a method to solve these educational problems. As the first solution to these problems, this paper deals with a thought-readable task suitable for primary education. In this way, we clearly define how we grasp the thinking in programming. Second, the method is independent of the environment and the individual so that it can be applied to a variety of situations. To realize this solution, this paper focuses on the development of a quantitative evaluation indicator as its policy.

Specifically, the policy consists of 2 issues: dealing with drawing tasks and converting source code into features. Introducing these policies, the paper solves the problems related to programming thinking.

2. Policies for Issue Resolution

2.1. Use of Figure Drawing Tasks

There are different styles of programming, depending on the content of the task being handled. It is not realistic to present an evaluation method covering all of them. We should consider characteristics of tasks that are suitable for beginning students. In programming education in Japan, the majority of elementary schools use visual languages such as Scratch and Viscuit. These tools are used because they are suited for primary education.

Methods of operation are sensible and user-friendly. It is easy to recognize mistakes because the execution results can be visually confirmed.

In addition, the programming guide in Japan introduces the task of drawing polygons. In figure drawing tasks, students directly recognize their own coding content. Molina et al. confirmed that displaying text and images together enhances students' understanding of elementary education about geometry (Molina et al., 2018). Procedures that draw figures are easier to interpret for beginning students than procedures without a clear image. Programming tasks that have visual results such as figure drawing is easy to tackle for beginners. We believe that a figure drawing task is one of the tasks in which students' thinking is most easily reflected.

To search for an evaluation method for thinking activities during programming, we suggest limiting the target tasks in this paper. There are two important points. The first is to target intuitive tasks applicable to elementary education. The reason is obvious from programming tasks adopted in elementary education in Japan.

The second is that the structure of the solution is unique. Tanigawa et al. have found that there are common patterns among students in figure drawing tasks (Tanigawa et al., 2011). In programming, students have many points where they must select the right options to reach the correct codes. The work examines the order of functions to draw a figure. From the function call logs, the work has succeeded in identifying points where many students may take options leading to wrong answers. In the tasks they use, the drawing order is specified. If the order of drawing is not specified, there will be an unlimited number of correct procedures, which makes it difficult to distinguish good thinking from wrong one. Similarly, if a correct procedure can be described by a series of different functions, it becomes difficult to determine the correctness of the procedure.

It is necessary that the flow of function calls should be unique to evaluate the thinking during the task. If it is the case, it is possible to evaluate students' thinking patterns from their answers, as in the study. Tanigawa et al. only focus on the final answers. A method is required to evaluate students' thinking during solving a task.

2.2. Conversion of Source Code to Features

In evaluating students' thinking activities during programming, time-series data representing students' activities is necessary. Bosch et al., Grafsgaard et al. and Jaques et al. have estimated students' emotions in learning including programming from sensing data of body movements and eye gaze (Bosch et al., 2014; Grafsgaard et al., 2014; Jaques et al., 2014). Many studies estimate tendencies during learning from sensing data that seem to be useful. However, in actual education, it costs too high to provide a sensor for each student.

Let us see what kind of data is being dealt with in studies that focus on computational thinking. Guggemos has shown a relationship between students' computational thinking ability and surrounding factors (Guggemos, 2021). Wei et al. have presented the direction of an effective approach for improving computational thinking by conducting a controlled experiment in elementary school subjects (Wei et al., 2021). Different from sensing data, these studies connect the quality of computational thinking to indicators such as questionnaires for students and the scores that students earned on tasks.

However, data obtained after finishing a task cannot be used to estimate thinking about the contents of the task in the process of answering it. If the data are not time-series data that occur during the solving process, we will miss the timing of interventions that are closely related to the contents of tasks. Moreover, personal metrics such as questionnaires are subjective labeling, and the results are dependent on environments and individuals.

Source code changes while a student solves a task. Unless it is a large programming task, there is only one source code file to edit. By observing this file sequentially, source code can be regarded as time-series data, such as sensor data. The source code is the student's solution process itself. Intervention by students' thinking is reflected in it every time they change their minds. Therefore, it is quite possible to read the student's thinking toward a task from time-series source code.

Unfortunately, source code is not utilized in the area of thinking evaluation. Cosma et al. showed the result of research on plagiarism of source code (Cosma & Joy, 2008). Djuric et al. designed a source code similarity detection system that is more promising than existing systems (Đurić & Gašević, 2013). A major theme in source code research is plagiarism. This is probably because the source code serves as a submission for the instructors.

Lazar et al. proposed a method of automatically presenting necessary fixes to a program through text-based program synthesis (Lazar & Bratko, 2014). This is an example of providing students with guidelines during the solution, based on their editing of source code. However, it is difficult to express task contents in detail with text-based ideas. It is possible to recognize the existence of a description as text. It is difficult to determine the role of the description from the text information. There is probably no previous method for estimating students' thinking on task contents from time-series data of source code.

3. Quantitative Evaluation by Coding Vectors

3.1. Creation of Coding Vectors

In this section, we describe the procedure for creating coding vectors. We call the vector data created by this method a coding vector. We named it a coding vector because it is a vector representation of the data during coding. In addition, we discuss its characteristics along with the procedure for its creation. Figure 1 shows the procedure for creating feature vectors. We describe the procedure in Figure 1 in three main parts.

The first is to obtain the source code. While students solve a programming task, their solution, source code, changes. We save their source code sequentially. The bundle of source code is the time series data of one's solution process. As can be seen from Figure 1, source code at a point in time t is a single sample of time-series data. We believe that source code can potentially show the change in students' thinking. It is also necessary to transform source code into a form that facilitates thinking evaluation. Therefore, it is important not to miss the results of programs expressed in source code by transformation.

The second is to obtain function logs. Functions logs refer to records of functions that are called when a program is executed. One of the logs is a log of a function name and an argument value when the function is called. We use function logs to transform source code into a form that is easier to analyze. The

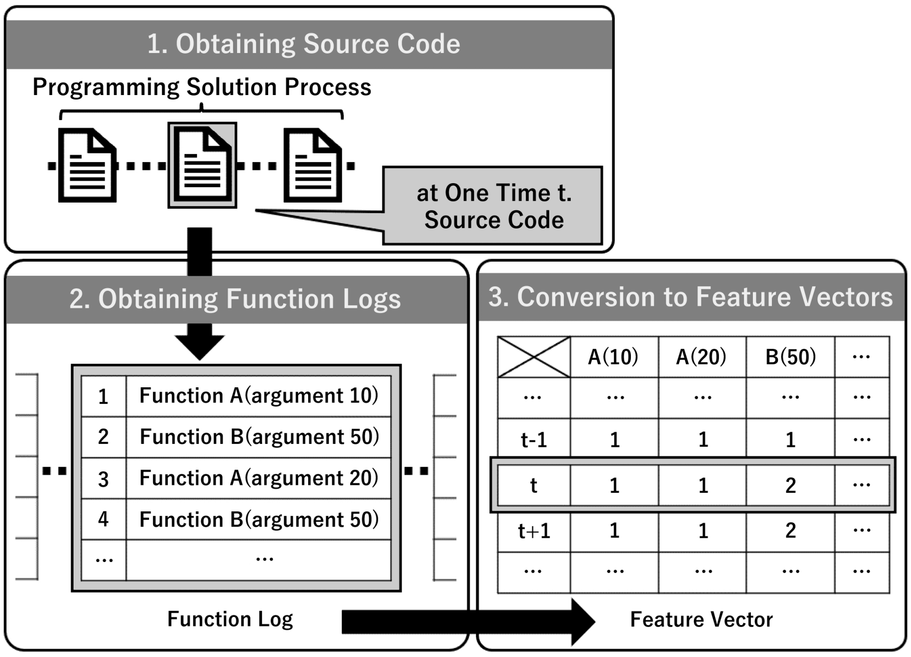

1. Obtaining Source Code
Programming Solution Process
at One Time t. Source Code
2. Obtaining Function Logs
1 Function A(argument 10)
2 Function B(argument 50)
3 Function A(argument 20)
4 Function B(argument 50)
Function Log
3. Conversion to Feature Vectors
A(10) A(20) B(50)
t-1 1 1 1
t 1 1 2
t+1 1 1 2
Feature Vector

Figure 1. Procedure of coding vectors creation.

number of functions and actions marked as text is different from the number of functions and actions called in execution. This is because of the use of repetition and conditional branches, and changes of variable values as arguments. Function logs include the number of times the function is called and the value of an argument. Therefore, the content that students are trying to realize can be properly recognized by function logs. We obtain function logs from the source code at each point in time obtained in the previous step. Time-series data of function logs are created with the same number of samples as the source code.

The third is to obtain coding vectors. Function logs are a good representation of what students are trying to express. However, this is not easy to analyze mechanically. The vertical logs are not suitable for calculating values as indicators. Therefore, we transform the data from function logs to vector type. The function log is a combination of functions and arguments. We count how many times each combination has been called from a single function log. If the number of combination types is n, the result of this measurement is an n-dimensional vector. By converting the time-series data of function logs into vectors, we obtain data in tabular form, with a vertical length of t and a horizontal length of n. This tabular data represents the student's solution process. The vertical length of the table is the number of samples in a time series. The horizontal length of the table is the number of combinations of functions and arguments required for a task. In other words, columns of coding vectors are features to represent the coding details. The blank source code has 0 values for all features and can be regarded as the origin. In other words, the values of coding vectors represent the total amount and direction of coding.

Through these steps, coding vectors can be obtained. The coding vector, which is the measure of evaluation, should well represent the student's thinking during the solution. We then describe our thoughts on the characteristics of vectors. The contents of source code and function logs would not change depending on the method of obtaining them. However, coding vectors are different. The columns as features should contain the necessary combinations for a task. Some functions would not require arguments. It is better to record the number of function calls regardless of the arguments. The problem is the wrong use of functions. In terms of covering students' thinking, it is necessary to take from the indicators even when they make mistakes. To do this, it is impossible to simply count combinations for correct content. Coding vector features must also include information about wrong combinations. However, if the argument is a continuous value, the number of combinations required for the expression is countless. It is difficult to cover all wrong combinations. In this paper, if a value of an argument is a mistake, we count it as the number of mistakes in the function. This is the basic characteristic of coding vectors to represent students' thinking.

3.2. Assumptions about Solution Process

In Section 3.1, we defined coding vectors for quantitative evaluation, based on the idea that source code is a potential expression of thinking about tasks. Next, it is important to grasp changes in coding vectors caused by the progression and regression of students' thinking states. In other words, we should consider how the activity of thinking develops during programming. In this section, we discuss our thoughts about students' programming solution process before considering how to utilize coding vectors.

When students solve programming tasks, their answers are expected to approach the correct answer in a stepwise manner.

This is the first assumption. In general programming, it is difficult to build a perfect flowchart-like structure from the beginning. A novice programmer unfamiliar with programming would build with the goal of reaching the middle of the structure. They would also take other ways, such as putting off the conditional branches. It is also expected that students will naturally proceed with programming from the front of the flow. This is because the correctness of the previous function's implementation determines the conditions for creating the next function. If the previous function incorrectly deals with a common variable, the next function cannot begin in the correct condition in the first place. Implementing from the front and building up each step so that the functionality works correctly is a natural way to solve programming tasks. This assumption will be especially true for figure-drawing tasks. If a unique procedure is established, like a drawing song, students will naturally implement it according to that procedure. In the case of drawing tasks, this is because the coordinates of an object, such as a paintbrush, are affected by the previous step. The position and direction of the object after the previous procedure are directly the conditions before starting the next procedure.

Second, we believe that programming activities follow a transition similar to a tree structure. As in the first assumption, the student's thinking is considered to approach the correct answer in stages. In other words, the contents required for a correct answer can be divided into stepwise elements. Stepwise elements are necessary for coding, such as setting up components consisting of functions and variables, repetitive expressions, and conditional branches. If a unique procedure for a task is established, the order in which each element should be implemented will naturally be established. Students may be able to implement stepwise elements in one fell swoop if they are highly skilled. However, even in that case, it follows an absolute forward direction to approach the correct answer. There is an absolute flow of solutions common to students in coding programming tasks.

Of course, it is possible to make a mistake during the solution. A mistake is to code with a misrepresentation of the element one is trying to realize. If the stepwise flow approaching the correct answer is viewed as linear, then mistakes are flows branching off from it. Students who make a mistake will make revisions on the same element until the result of that run is correct. Fixing a mistake can be regarded as finding the correct flow for the element they were thinking about. Students then seek the correct flow for the next step in a similar manner. In this way, students' activities solving a task spread out like a tree structure. The stepwise flow of correct answers is the central trunk, and patterns of mistakes exist as branches. In the next section, based on these assumptions, we propose a method of thinking evaluation using coding vectors.

3.3. Distance Evaluation of Coding Vectors

Coding vectors are created based on source code that reflects students' thinking activities. Therefore, the value of the coding vector is a coordinate that represents the stage of the student's thinking during a task. The number of columns of coding vectors, or of features, is the number of dimensions. It means that vector data at each point in time has a coordinate in a multidimensional space. Therefore, we try to grasp the changes in students' thinking activities by observing the movement of coordinates in time series. With reference to the assumptions in section 3.2, there are vectors that belong to a correct flow in coding vectors. These vectors can be viewed as a single connected constellation in a multidimensional space. By connecting several source codes that satisfy the stepwise elements in a sequence, the coding flow that should be followed in the task can be represented in space. Based on this idea, we propose a three-step quantitative thinking evaluation by coding vectors. Before explaining our method, we show an image of evaluations in Figure 2.

The first is the visualization of coding vectors through dimensional compression. We assumed that the flow toward a correct answer existed in stages. In order to recognize this spatially, we need to reduce the dimensions of features to at least three. Therefore, we use PCA, a method of dimensional compression. We check coding vectors compressed into two dimensions by PCA on a two-dimensional graph. This method allows visualization of the correct flow that

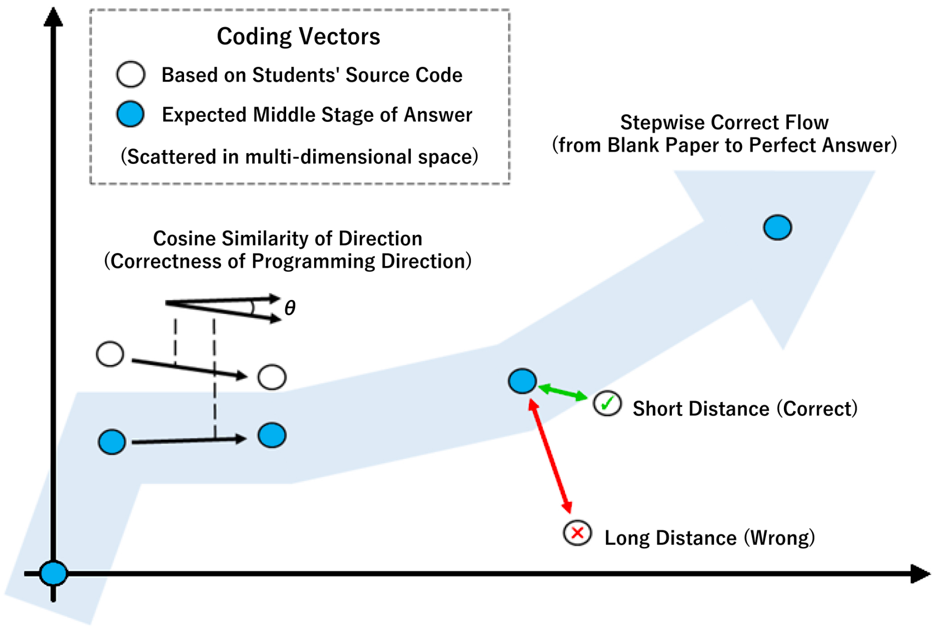

Coding Vectors
Based on Students' Source Code
Expected Middle Stage of Answer
(Scattered in multi-dimensional space)
Stepwise Correct Flow
(from Blank Paper to Perfect Answer)
Cosine Similarity of Direction
(Correctness of Programming Direction)
θ
Short Distance (Correct)
Long Distance (Wrong)

Figure 2. Evaluation based on coding vectors.

should be followed in a task. It would also provide a visual check on how far the source code in the student's solution is from the correct flow. This method is an easy way for anyone to check the path from a start to a goal.

The second is the determination of thinking separation degree based on distance. When a student is solving a task, the correctness of source code at a given point in time may be judged by whether it follows the correct flow. The correct flow in the feature space is formed from points with multiple coordinates, like a constellation. If the source code in a solution matches the stepwise correct answer, then these two coordinates in the feature space will also match. In the case of source code with a mistake, the coordinate exists at a distance from the coordinates of correct answers. In addition, we consider the case of a mistake on a certain phasing factor. In that case, the coordinate of a mistake would be located around a coordinate that correctly satisfies that element. This idea is based on the assumptions in section 3.2. The distance between coordinates can be calculated based on a definition of Euclidean distance. Correct coding would be close to zero distance from correct answers. Incorrect coding would be so far away from them that its content would be out of degree. By observing a distance, we would be able to determine what stage students are at, including any incorrect codes.

The third is an evaluation of prospects focused on time-series changes. We believe that students' thinking, together with their behavior, can be divided into three main categories. There are three types: thinking that has a right prospect and correctly realizes an element, thinking that has no right prospect and incorrect coding, and thinking that has no right prospect and stops coding. In order to judge these from coding vectors, we should focus on time-series changes. As coding occurs, the coding vector also changes, regardless of the correctness of the coding. If the student's hand stops moving, the coding vector will not change. Changes in coding vectors are closely related to coding according to the student's thinking. Therefore, the existence of students' prospects can be determined from the change in the coding vector. In particular, we focus on the change between vectors, which is a vector of difference. The difference between two vectors can be regarded as the direction of movement in space. The difference vector means the direction of the coding. Therefore, we estimate the correctness of students' prospects by the direction of difference vectors. In particular, we calculate cosine similarity between difference vectors in the correct answers flow and difference vectors generated by coding. If the direction of the student's stepwise coding is correct, it should have a high similarity to one of the directions in a correct flow.

4. Task Experiment for Data Collection

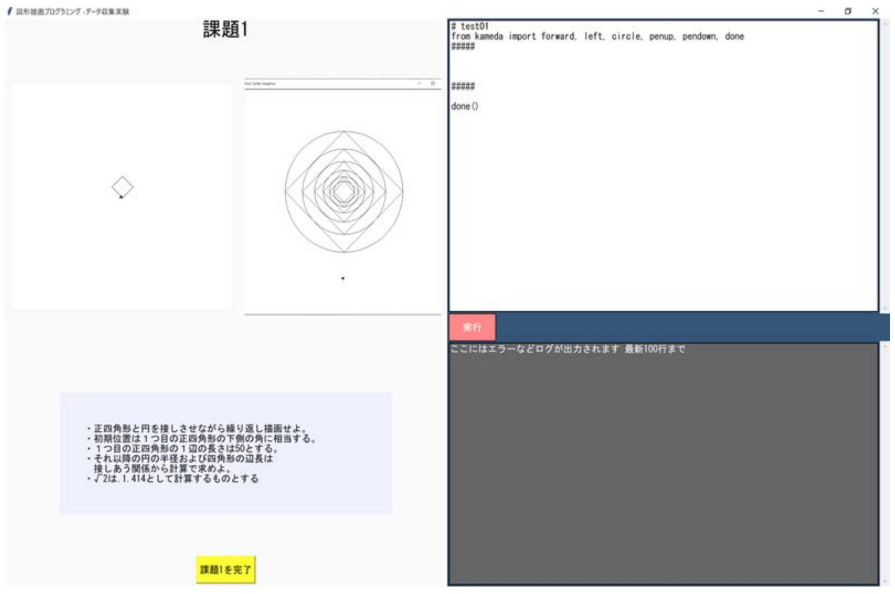
4.1. Outline of Task Experiment

We conducted a task experiment to gather source code. The programming task provided is a graphic drawing task. The number of tasks is two, and the standard solution time is 30 minutes for both tasks. The programming language used is Python. Turtle Graphics, the standard Python library, is used for drawing graphics. Drawing contents consist of five functions in Turtle Graphics functions. Those functions are forward(), left(), circle(), penup(), and pendown(). These were employed as operations for moving forward, rotating, drawing circles, and orbiting or not.

Figure 3 shows the actual task page displayed on a PC screen. The left side of the page is a drawing content. A Gif animation is placed on the left, and a completed drawing is placed on the right, side by side. Below them, there is a sentence of additional explanation about numerical values. The right side of the page is an editor. There is a source code editor at the top, and a box with debug log output at the bottom. There is an execution button in the center, which can be pressed to check the drawing result with the current source code.

26 university and graduate students participated in the task experiment. All participants had learned how to use Python. They understood how to handle drawing functions with 10 minutes tutorial. We told participants to ask their supervisors if they had any questions about descriptions. This is because we want to eliminate the time of stagnation caused by syntax errors. In addition, because the participants used their own laptops, they were able to code as they normally code.

We gathered source code data for 22 of the 26 participants through this experiment. We also recorded logs of keyboard input at the same time. Based on the response of three participants at the beginning, the level of difficulty and notations were refined. Therefore, the data of these three participants were excluded. One of the other participants lost information on keyboard inputs due to improper actions during the task. It was necessary to refer to keyboard inputs for data processing, which will be explained in the next section. Data from this one



```
# test01
from kameda import forward, left, circle, penup, pendown, done
#####


#####

done()
```

・正四角形と円を接しさせながら繰り返し描画せよ。
・初期位置は１つ目の正四角形の下側の角に相当する。
・１つ目の正四角形の１辺の長さは50とする。
・それ以降の円の半径および四角形の辺長は
　接しあう関係から計算で求めよ。
・√2は.1.414として計算するものとする

```
ここにはエラーなどログが出力されます 最新100行まで
```

Figure 3. Task page on PC screen.

participant was also excluded. As a result, 16 (73%) were able to solve the first task, and 18 (82%) were able to solve the second task. In both tasks, the number of participants who solved within a standard 30 minutes was less than half of all participants accomplishing the tasks. Other participants were divided into two groups: those who solved in more than 30 minutes or those who did not finish in more than 30 minutes and gave up. Therefore, the difficulty level of both tasks was suitable for them.

We gathered source code data for 22 of the 26 participants through this experiment. We also recorded logs of keyboard input at the same time. Based on the response of three participants at the beginning, the level of difficulty and notations were refined. Therefore, the data of these three participants were excluded. One of the other participants lost information on keyboard inputs due to improper actions during the task. It was necessary to refer to keyboard inputs for data processing, which will be explained in the next section. Data from this one participant was also excluded.

As a result, 16 (73%) were able to solve the first task, and 18 (82%) were able to solve the second task. In both tasks, the number of participants who solved within a standard 30 minutes was less than half of all participants who were able to solve the task. Other participants were divided into two groups: those who solved in more than 30 minutes or those who did not finish in more than 30 minutes and gave up. Therefore, the difficulty level of both tasks was suitable for them.

4.2. Attention for Conversion to Coding Vectors

In the task experiment, we recorded source code files every second. The one-second interval was set to grasp well changes caused by coding. The number of recorded files would be 1800 for 30 minutes of solution time. After counting files for 22 participants, the average solution time for the first task was about 34 minutes, and for the second one, about 36 minutes. The number of features required for the two tasks was 30 and 23, respectively. These features consist of the total number of each function, the correct combination of function and arguments, and the number of each function call with incorrect arguments.

Some of the recorded source code causes syntax errors when executed. This is because they contain code that was saved while the student was typing. Function logs cannot be obtained from them. In this case, we took over the feature values obtained from the latest source code that could be executed successfully. This method enables assigning coding vectors even to source code that cannot be executed. In other words, we should wait to change coding vectors until one coding session has been completed.

Even if no errors occur, source code that is different from the student's ideas may be recorded. For example, when a student writes the value of an argument as 1000, the value might be recorded as 1, 10, or 100, depending on the timing of the recording. In another example, when a student is indenting multiple lines sequentially, a state may be recorded in which not all indentations are complete. After observing the experiment data, we found some source code with these wrong conditions. In particular, the indentation operation was a relatively long coding process. Coding vectors at that time were expected to represent wrong expressions, regardless of the student's thinking. Therefore, we considered it necessary to make an exception at the step of obtaining the function logs. We observed logs of participants' keystrokes and found that there were at least five seconds between each coding session. Based on this finding, we processed these the same as the source code in error as long as a keystroke was occurring within 5 seconds. We believe that this processing makes students' thinking and coding vectors more relevant.

5. Coding Vectors Analysis

5.1. Visualization with PCA

For coding vectors collected in the task experiment, we show analysis results according to the method described in section 3.3. First, we check the spread of the solution process by visualization with PCA. Before applying PCA, data from 22 participants were normalized. The minimum value for each feature is 0 and the maximum value is the number of calls required for the final answer.

We specially processed numerical values of the number of features for calls with wrong arguments. The total number of mistakes can increase dramatically compared to the number of calls with a particular argument. Therefore, the values should be appropriately suppressed toward PCA. We took the natural logarithm for values greater than or equal to 0 for these features and added 1. If the original value is 1, it remains 1. The closer it is to 1, the more it retains the face of its size. We can considerably reduce the size of outliers, even those that exceed 100. This process allowed us to equalize the influence of each feature on the PCA.

Figure 4 shows a scatterplot of the flow of correct answers in the two tasks. Its two axes are the first and second principal components of the PCA. The contribution ratio up to the second principal component was 0.644. The flow of correct answers was divided into about 10 stages, such as drawing a polygon, drawing a circle, moving to the next starting point, repeating the process n times, and so on.

In Figure 4, we can see the route from a blank paper to a completed drawing by coloring. This route extends in a relatively coherent direction. In particular, at the end of the process, for the repetition of graphic elements, these points are placed in a straight line. From both scatter plots, it can be seen that the path to the correct answer consists of three major linear paths, at the beginning, middle, and end of the path. In summary, figure drawing programming has stepwise elements, and they are also coded according to a certain direction. We can see this trend in Figure 4.

Figure 5 shows a scatterplot of the solution process for some of the participants.

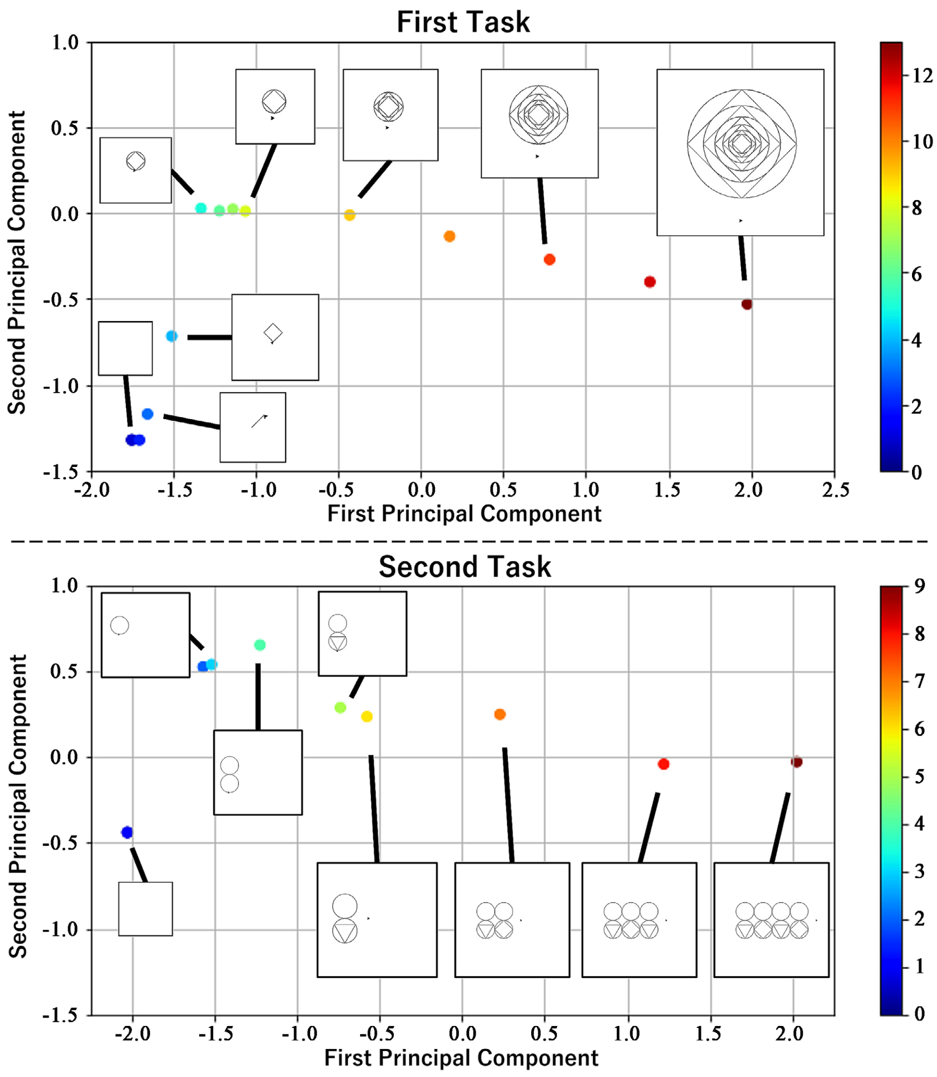


Figure 4. Coding vectors flow of correct answers.

There are several examples separated into completed and uncompleted solution processes. Each point is a coding vector obtained from per-second source code. Time series data are colored from blue to red from the start of a task to its end. It should be noted that while coding vectors do not change, their coordinates are the same, so dots in Figure 5 will overlap during those periods. Moreover, the flow of correct answers shown in Figure 4 is marked by gray triangles.

On the completed side, there are examples that follow this flow very well. It shows that these participants followed the expected steps in their thinking. It is clear that these participants understood the next stepwise element to be implemented and coded it correctly. However, on the completed side, there are examples where their final answer does not overlap with the correct answer dot. The reason is that the timing to stop drawing was different. Participants with understanding expressed the process of moving to the next starting point after drawing a shape. Other participants are missing the process of moving to the next starting point at the end of the drawing. The process does not involve an orbit,

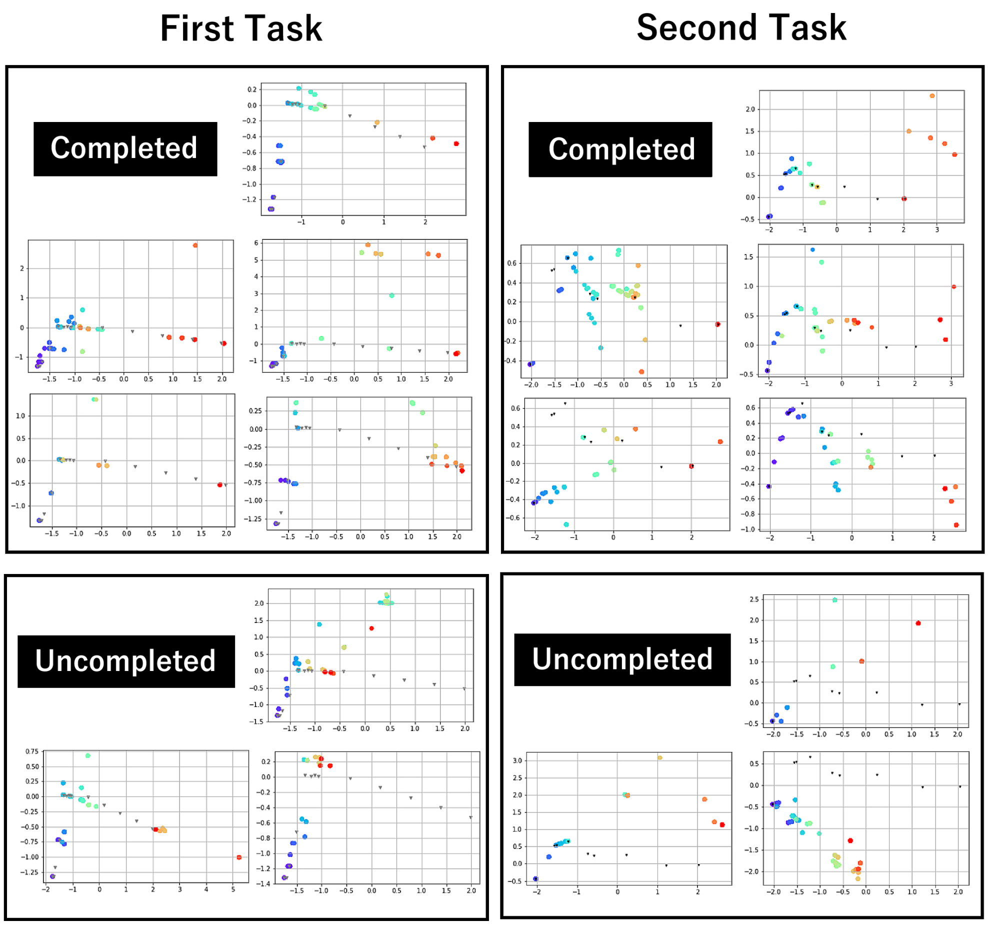


Figure 5. Solution process of participants.

so it is not reflected in the finished image as a drawing. We grouped such cases as the completed side for this analysis.

Next, there are examples with dots placed in space away from the flow of correct answers. It shows that participants made the wrong use of functions in their source code. It also shows the stage before mistakes were made and the stage after fixes were made, based on the color information. There are some durations when one or two dots pop up in space away from correct answers. It means that the mistake was fixed with a small number of touches. In contrast, there are some periods, especially in the uncompleted, where many dots are scattered in outer space. It means that participants make repeated fixes but are not able to fix them well. In other words, it is possible to visually check for three patterns: which stages they were able to think successfully, when mistakes were made, and whether they are struggling to fix those mistakes.

Finally, the uncompleted side had the characteristic of stagnation. Many on the uncompleted side stagnate in the early stages. We find that they stay in a state of mistake for a long time, because the color of the dots does not change continually. The reason why they cannot solve a task until the end is that they are unable to move forward into the next stage because of a lack of prospects.

Therefore, we can judge students' lack of prospects, which is the cause of the stagnation of the task, based on the long stay of coding vectors. Instructors can easily recognize students' thinking simply by observing the movement of coding vectors. Therefore, even in a group study setting, it would be possible to judge whether each student's coding is going well on the spot. Specifically, we can judge in detail the stage of thinking students reached, the occurrence of mistakes, and the lack of prospects due to stagnation.

5.2. Distance Evaluation of Correctness

We confirmed the visualization analysis with up to a second principal component. Next, we analyze the distance calculated from the base value of coding vectors as a more accurate indicator. Columns of wrong arguments were processed as we have done in section 5.1. This is to keep the recognition in line with PCA's results. We calculated Euclidean distances from the correct answers for coding vectors of 22 participants. The distance from correct answers is a distance from the nearest dot in the flow of correct answers shown in Figure 4.

Figure 6 shows the distribution of distances from correct answers. The total number of coding vectors for the first task is 44,708 and the number for the second task is 48,037. In Figure 6, outliers are excluded from the first task. This is because some of the distances from one's coding vectors were over 600 values. The histogram of the first task does not include these 145 (0.3\% of total) outliers.

The number of zeros is by far the largest number for both tasks in Figure 6. This is because the coding vectors of participants match the vectors of the expected correct answer flow. It also has a relatively large number of cases on the left side. These cases are source code with a small number of wrong function calls. In particular, students' solutions in the early steps would fit this case, since the number of functions required is small in itself.

In contrast, it would be far from a correct answer if the distance is greater than 10. The more advanced the steps of a task, the greater increases the number of wrong calls that occur from a single writing error. Therefore, the distance

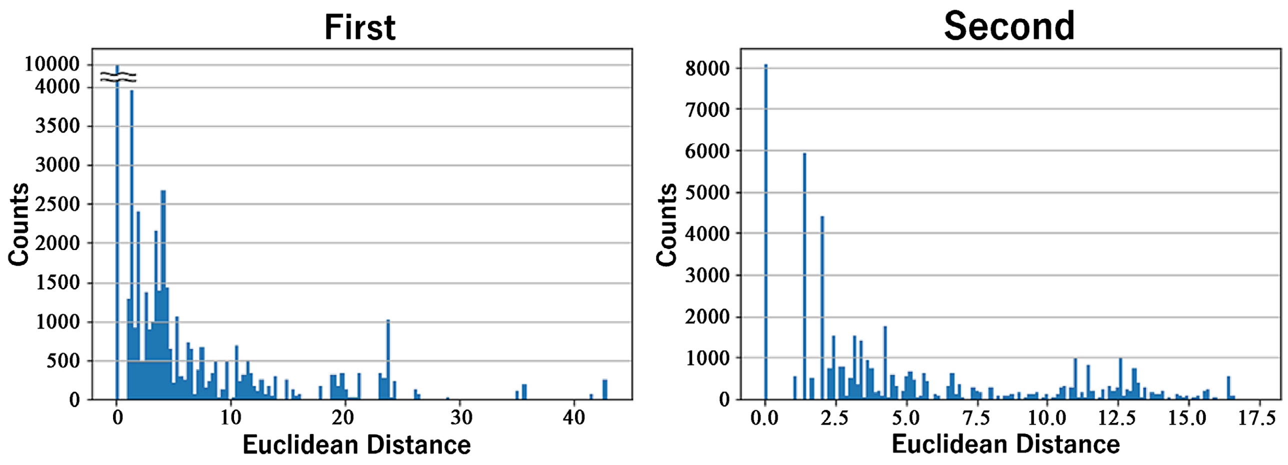


Figure 6. Histograms of Euclidean distance in two tasks.

from correct answers would have more effect on thinking evaluation if judged based on the steps in the coding process. In this way, we can evaluate quantitatively the steps of students' thinking as well as visualization by using the distance from correct answers as an indicator. In particular, it is said that the student has a good understanding of the steps if this distance is close to zero.

We describe a point that should be kept in mind in using distance as an indicator. The distribution is different for the first and second tasks in Figure 6. The distribution is different for the two tasks in Figure 6. A common threshold for these tasks does not exist in Figure 6. However, we might get the threshold by generalizing. As mentioned earlier, it is expected that the number of mistakes is proportional to the step students reached.

Figure 7 shows the result of dividing values used in Figure 6 by the number of all function calls. In other words, the distance was scaled by the progress of the task. Both shapes are closer to mountains than in Figure 6. The center of the mountain distribution is about 0.2 in both cases. In this way, the distribution of distances approaches the form of a probability distribution by scaling with the number of function calls.

Based on the idea of probability distributions, we can judge that these cases on the right side of Figure 7 are different, without the influence of progress. The result in Figure 7 shows the possibility of finding a common threshold for tasks.

5.3. Directions Evaluation Based on Coding Changes

We have shown a method to quantitatively evaluate the degree of separation by focusing on the current source code in Section 5.2. Next, we try a more detailed analysis by focusing on changes in time-series source code. Each coding vector represents the current coding content. These differences can be regarded as vectors with the amount and direction of movement from the content before the change to the content after the change. We assumed that the student's solution process would follow the flow of correct answers. The arrows connecting each dot in Figure 4 represent the direction they should follow. Therefore, we evaluate

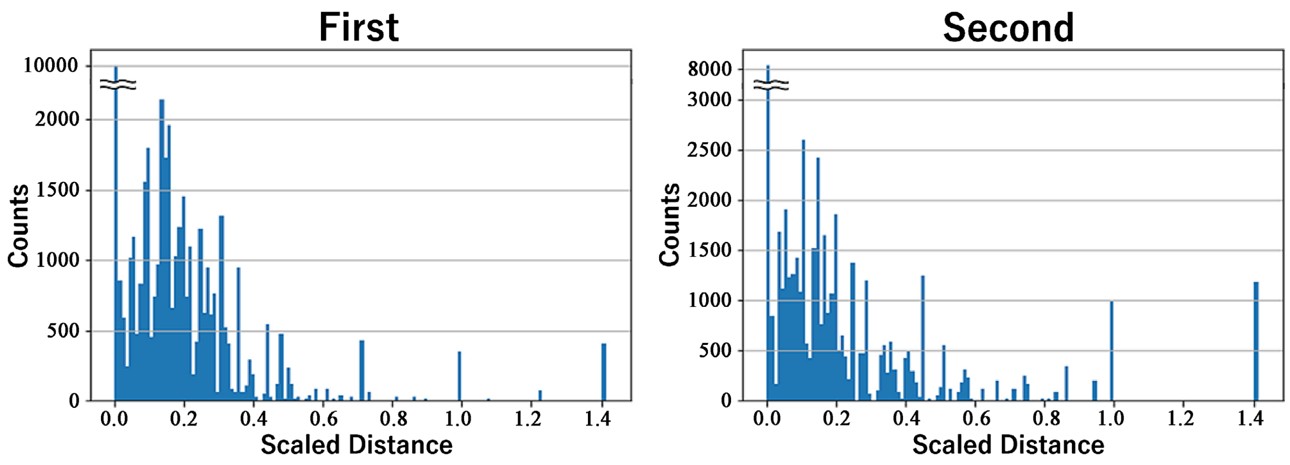


Figure 7. Histograms of scaled distance in two tasks.

the correctness of coding by cosine similarity with these difference vectors.

Figure 8 is a box plot of the number of coding changes from 22 participants. As described in section 4.2, the coding vector changes when there are no execution errors and no key input for 5 seconds. Figure 8 shows that both tasks averaged about 25 changes in students' coding vectors. Difference vectors were obtained for two coding vectors before and after these changes occurred. The difference vector of correct answers flow was prepared by the permutation method. This is because skipping or going back is expected for stepwise elements. For example, the first task has 13 vectors of correct answers flow, so there are 156 difference vectors.

Distributions of cosine similarity are shown in Figure 9. This cosine similarity value is the highest value of the combination of the student's difference vector and difference vector of correct answer flow. If cosine similarity is 1, the student's coding is in perfect sync with the ideal stepwise coding. Cases where the cosine similarity exceeds 0.8 would also be the result of equivalent ideal coding. Figure 9 shows that there is an ideal coding in students' difference vectors. Of course, students' difference vectors include changes related to mistakes. The case of mistakes does not have a high similarity to ideal difference vectors in the first

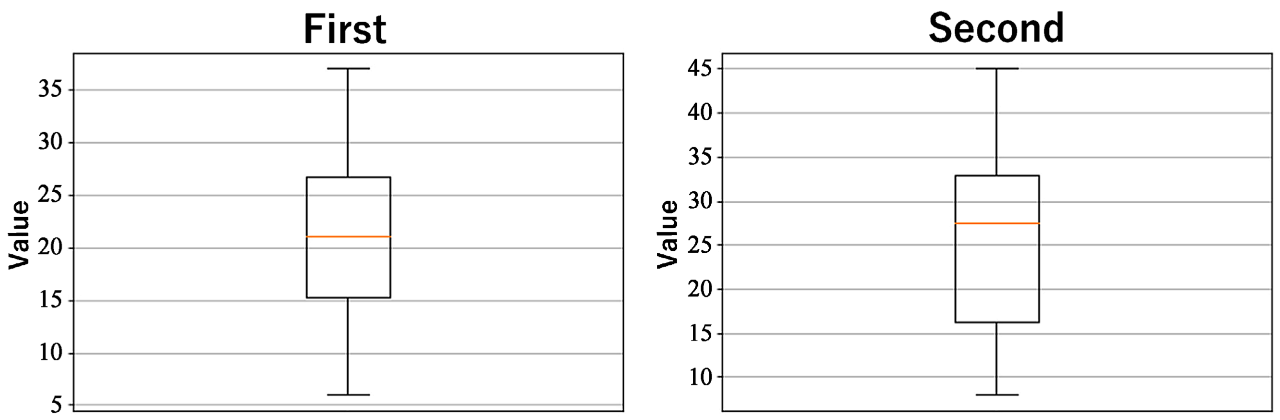


Figure 8. Box plot of cording changes in solution.

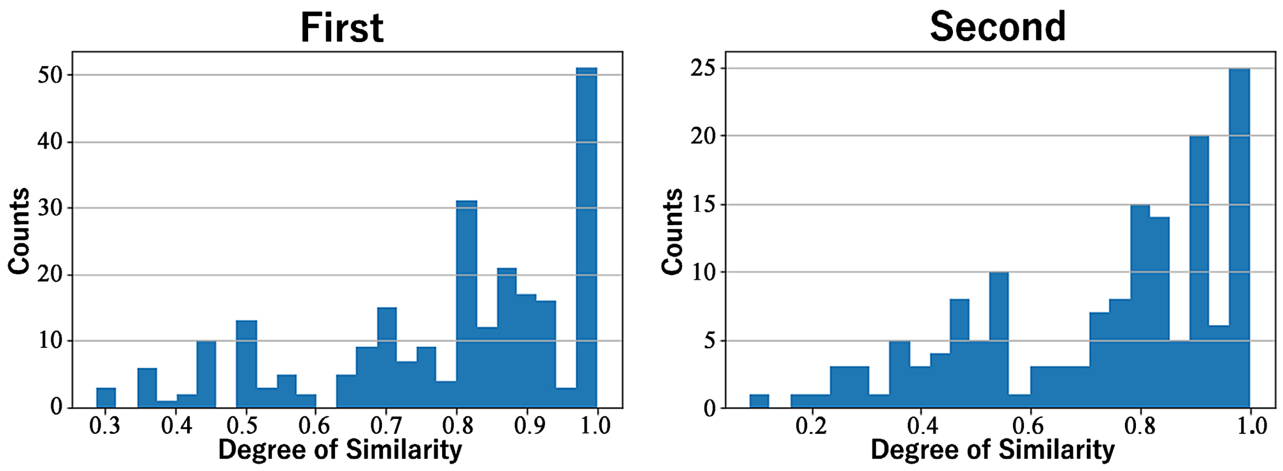


Figure 9. Histogram of cosine similarity with direction.

place. The cases with not high cosine similarity in **Figure 9** would belong to this category. In this manner, the higher similarity is, the more strongly they can be judged to have a similar directional identity. However, low similarity shows no relationship to expected directions, and it is difficult to discuss the content of the coding from this exclusive information.

In addition, we compare the starting point and ending point with the highest similarity combination in **Figure 9**.

We labeled the difference vectors with their starting and ending points.

The starting and ending points on the correct answer flow side were labeled with their base stage. The starting and ending points on the student side were labeled with the closest steps in distance to the based student's vector. The percentages of the starting and ending labels matched were 33.9% for the first task and 23.2% for the second task. Of course, in the case of wrong coding, there is no validity to the starting and ending labels. It would be better to say about this result that about 30% of the student's difference vector was about appropriate coding. In addition, including cases where only one of the labels matched, the percentages of labels that matched were 44.5% and 58.5%, respectively. Difference vectors include those toward a mistake and those fixed from a mistake. The increase from the percentage of both matched is considered these vectors.

We checked the similarity with the direction of the correct answer flow by cosine similarity. As a result, we can say that a student's appropriate coding can be identified in these coding stages by the similarity. The distance evaluation is an evaluation against source code at a point in time. In contrast, the directions evaluation is an evaluation of the differences in source code changes. It does not affect the difference even if any content is wrong except for the part that the student has changed. Therefore, directions evaluation can help measure the validity of thinking that may be missed in distance evaluation.

However, if a student's coding is wrong, evaluation by cosine similarity is not easy. This is because it is necessary to prepare a difference vector for the pattern we want to identify. It is difficult to prepare for a variety of error patterns in advance. To make it possible, it is preferable to gather data on students' answers in advance and discover typical mistake patterns that occur in the answers. It would allow us to evaluate the student's coding in three categories: correct, typically mistaken, and other (singularly mistaken).

6. Conclusion

In this paper, we proposed a method to evaluate students' thinking steps based on their coding vectors. We visually checked the paths of the solution process and the students' thinking steps in scatter plots with PCA. This method allows instructors to easily observe students' coding contents in a group study. We confirmed the validity of using distance from correct answer flow as a quantitative evaluation indicator. Moreover, scaled distances can result in common thresholds for multiple tasks. Directions evaluation by cosine similarity gives us a more certain judgment of the thinking step. However, this method requires labels for comparison. Our future task is to expand the generality and range of applications for similarity evaluation.

We should continue validation in order to show more clearly that this theory can be applied to educational programming. In conjunction, it is necessary to determine the range of programming for which this evaluation is valid. It is important for future development to confirm the possibility of applying this method not only to figure drawing, but also to various types of tasks. We would like to extend our evaluation theory based on the findings of this paper. One possible approach is the introduction of probability theory to distance evaluation.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

Bosch, N., Chen, Y., & D'Mello, S. (2014). It's Written on Your Face: Detecting Affective States from Facial Expressions While Learning Computer Programming. In *Intelligent Tutoring Systems: 12th International Conference, ITS 2014* (pp. 39-44). Springer International Publishing. https://doi.org/10.1007/978-3-319-07221-0_5

Cosma, G., & Joy, M. (2008). Towards a Definition of Source-Code Plagiarism. *IEEE Transactions on Education, 51,* 195-200. https://doi.org/10.1109/TE.2007.906776

Đurić, Z., & Gašević, D. (2013). A Source Code Similarity System for Plagiarism Detection. *The Computer Journal, 56,* 70-86. https://doi.org/10.1093/comjnl/bxs018

Fagerlund, J., Häkkinen, P., Vesisenaho, M., & Viiri, J. (2021). Computational Thinking in Programming with Scratch in Primary Schools: A Systematic Review. *Computer Applications in Engineering Education, 29,* 12-28. https://doi.org/10.1002/cae.22255

Grafsgaard, J., Wiggins, J., Boyer, K. E., Wiebe, E., & Lester, J. (2014). Predicting Learning and Affect from Multimodal Data Streams in Task-Oriented Tutorial Dialogue. In *Proceedings of the 7th International Conference on Educational Data Mining (EDM)* (pp. 122-129).

Guggemos, J. (2021). On the Predictors of Computational Thinking and Its Growth at the High-School Level. *Computers & Education, 161,* Article 104060. https://doi.org/10.1016/j.compedu.2020.104060

Jaques, N., Conati, C., Harley, J. M., & Azevedo, R. (2014). Predicting Affect from Gaze Data during Interaction with an Intelligent Tutoring System. In *Intelligent Tutoring Systems: 12th International Conference, ITS 2014* (pp. 29-38). Springer International Publishing. https://doi.org/10.1007/978-3-319-07221-0_4

Lazar, T., & Bratko, I. (2014). Data-Driven Program Synthesis for Hint Generation in Programming Tutors. In *Intelligent Tutoring Systems: 12th International Conference, ITS 2014* (pp. 306-311). Springer International Publishing. https://doi.org/10.1007/978-3-319-07221-0_38

Molina, A. I., Navarro, Ó., Ortega, M., & Lacruz, M. (2018). Evaluating Multimedia Learning Materials in Primary Education Using Eye Tracking. *Computer Standards & Interfaces, 59,* 45-60. https://doi.org/10.1016/j.csi.2018.02.004

Sun, L., Hu, L., & Zhou, D. (2021). Which Way of Design Programming Activities Is More

Effective to Promote K-12 Students' Computational Thinking Skills? A Meta-Analysis. *Journal of Computer Assisted Learning, 37,* 1048-1062. https://doi.org/10.1111/jcal.12545

Tanigawa, K., Harada, F., & Shimakawa, H. (2011). Detecting Learning Patterns during Exercise from Function Call Logs. *International Journal of Advanced Computer Science, 1,* 30-35.

Tedre, M., & Denning, P. J. (2016). The Long Quest for Computational Thinking. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (pp. 120-129). Association for Computing Machinery. https://doi.org/10.1145/2999541.2999542

Tikva, C., & Tambouris, E. (2021). Mapping Computational Thinking through Programming in K-12 Education: A Conceptual Model Based on a Systematic Literature Review. *Computers & Education, 162,* Article 104083. https://doi.org/10.1016/j.compedu.2020.104083

Wei, X., Lin, L., Meng, N., Tan, W., & Kong, S. C. (2021). The Effectiveness of Partial Pair Programming on Elementary School Students' Computational Thinking Skills and Self-Efficacy. *Computers & Education, 160,* Article 104023. https://doi.org/10.1016/j.compedu.2020.104023

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM, 49,* 33-35. https://doi.org/10.1145/1118178.1118215