A Graph-based Blank Element Selection Algorithm for Fill-in-Blank Problems in Java Programming Learning Assistant System

Nobuo Funabiki, Tana, Khin Khin Zaw, Nobuya Ishihara, and Wen-Chung Kao

***Abstract*—As a reliable and portable object-oriented programming language, *Java* has been extensively used in industries and taught in schools. To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS provides a *fill-in-blank problem* for novice students who have started learning Java programming including grammar and basic programming through *code reading*. In this problem, students are asked to fill in the blank elements in a high-quality Java code. In this paper, we propose a graph-based *blank element selection algorithm* to select as many blanks as possible that have grammatically correct and unique answers from a given code. First, the algorithm generates a graph by selecting each candidate element in the code as a vertex, and connecting any pair of vertices by an edge if they can be blanked together. Then, it extracts a maximal clique of the graph for a solution. For evaluations, the correctness of the algorithm is verified manually by applying it to 100 Java codes. Eventually, the educational effects in Java programming learning are confirmed by assigning generated fill-in-blank problems to students in our Java programming course.**

***Index Terms*—Java programming education, JPLAS, fill-in-blank problem, blank element selection, graph, clique, algorithm.**

I. Introduction

JAVA has been extensively used in industries as a reliable and portable object-oriented programming language, which involves mission critical systems for large enterprises and small-sized embedded systems. Thus, the cultivation of Java programming engineers has been in high demand amongst industries. A great number of universities and professional schools are offering Java programming courses to meet these needs.

A Java programming course consists of grammar instructions with classroom lectures from a teacher and program exercises through computer operations by students. A programming exercise will proceed under the following cycle:

1) The teacher gives an assignment to the students in the course.
2) A student writes a Java code for the assignment, and submits it by a paper or an electronic file to the teacher.
3) The teacher evaluates each code manually, and responds with comments to the student, if necessary.
4) The student modifies the code by following the teacher's comments and resubmits it, if necessary.

Manuscript received Mar. 20, 2017.

N. Funabiki, K. K. Zaw, Tana, and N. Ishihara are with the Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan, e-mail: funabiki@okayama-u.ac.jp.

W.-C. Kao is with the Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan, e-mail: jungkao@ntnu.edu.tw.

Generally, a teacher will teach several dozen students in a course. During the class, it could be a challenge for the teacher to give sufficient instructions to every student due to limited resources. Additionally, the evaluation of the codes and the returning of the comments to the students on time may become a burden to the teacher. If the response from the teacher is delayed or missed, the student may lose the chance to understand and fix the problems in the code. Eventually, it is potential that a number of students may lose motivations to study the Java programming seriously. Furthermore, nowadays, plenty of institutes have increased the number of students in the course due to budget constraints. It may be difficult even to manage plenty of codes, scores, and instructions/comments of students in the course, which could lead to human-induced mistakes.

To solve the mentioned problems in a Java programming course, we have proposed and implemented a Web-based *Java Programming Learning Assistant System (JPLAS)* [1]-[3]. JPLAS mainly provides two types of problems, namely the *code writing problem* and the *fill-in-blank problem*, to support students' self-studies at various learning levels. JPLAS performs excellently in reducing the load of evaluating the codes and in improving a student's motivation with immediate responses to his/her answers. This system is expected to promote Java programming educations in all kinds of institutes around the world.

In JPLAS, the *code writing problem* is designed for a student to learn writing a Java source code from scratch. This function is implemented based on the *test-driven development (TDD) method* [4], using an open source framework *JUnit* [5]. With *JUnit*, the answer code is automatically tested on the server to verify its correctness when submitted by a student. Thus, a student is allowed to repeat the cycle of writing, testing, modifying, and resubmitting a code by him/herself. However, we have detected one problem for this function that a student needs to write a code that can be correctly tested with *JUnit* via *test code* prepared by the teacher. Hence, this function may not be suitable for a novice student who has just started Java programming study.

On the other hand, the *fill-in-blank problem* in JPLAS intends for a student to learn the Java grammar and basic programming skills through *code reading*. In a fill-in-blank problem, a Java code with several blank elements is shown to a student, where he/she needs to fill in the blanks. This *problem code* should be of high-quality, most worth for code reading. An *element* is defined as the least unit of a code, such as a reserved word, an identifier, and a control symbol. A *reserved word* signifies a fixed sequence of characters that has been defined in Java grammar to represent a specific

(Advance online publication: 24 May 2017)

function, which should be mastered first by the students. An *identifier* is a sequence of characters defined in the code by the author to represent a variable, a class, or a method. A *control symbol* in this paper indicates other grammar elements such as "." (dot), ":" (colon), ";" (semicolon) , "(,)"(bracket), "{, }" (curly bracket).

To solve the fill-in-blank problem, students are required to carefully read the code to understand the structure, the algorithm/logic, and the semantics. Subsequently, by exercising knowledge of grammar and applying syntax rules to each statement, they fill in the blanks correctly. Hence, the fill-in-blank problem and code reading are intimately connected with each other. The importance of *code reading* in learning programming has been studied in a considerable amount of literature as introduced in Section VII.

The function for this *fill-in-blank problem* in JPLAS involves *teacher functions* and *student functions*. With teacher functions, a teacher can generate a new problem by selecting a Java code from the database and selecting the blank elements from the code. This generated problem can be stored in the database. Afterwards, the teacher can register a new assignment by selecting a problem from the database. With student functions, a student may repeat the cycle of accessing an assignment, solving the problem, submitting the answer, checking the test result, and correcting and resubmitting the answer to improve their learning, without a help from a teacher.

In our implementation of the *fill-in-blank problem*, the correctness of each answer is checked through string matching with the corresponding correct answer in the server. Here, the original element for each blank in the code is used as the unique correct answer, because it is not only simple but also encourages students to study code reading. Thus, when an element is selected for a blank, the original element in the code must be the unique and grammatically correct answer to avoid confusions of novice students. Unfortunately, this proper blank element selection is not that easy for a teacher, particularly if he/she wants to blank a good deal of elements for a harder problem. For example, if all the elements representing the identifier for one variable are blanked, a student cannot answer it at all. At least one element must remain in this case.

In this paper, we propose a graph-based *blank element selection algorithm* to automatically select proper blank elements from a given Java code. First, this algorithm generates a *compatibility graph* by selecting every candidate element in the code for a blank as a *vertex*, and connecting any pair of vertices by an *edge* if they can be blanked together. To fulfill this purpose, we define the conditions that a pair of elements cannot be blanked simultaneously. Then, a *maximal clique* [6] of the compatibility graph is extracted, to find a maximal set of proper blank elements. As discussed in Section V-C, a fill-in-blank problem becomes more difficult when a larger number of elements are blanked. Therefore, by blanking a subset of the elements selected by the algorithm, a variety of fill-in-blank problems can be generated with different levels of difficulties. In addition, we believe that the proposed algorithm can be applied to other programming languages, although detailed investigations will be required.

The proposed algorithm selects various types of blank elements for practical programming educations to novice students. To master programming, students must understand the grammar and code syntax that consists of various types of elements, where only the correct combination of elements will result in a correct code. Even if a semicolon " ; " or a period " . " is missing in a code, it will not be able to operate correctly. Unfortunately, it seems that novice students are inclined to make such mistakes. It is noticed that in [33][35][36], typo is a major source of frustration for most novice programmers. If the teacher focuses on specific types of elements, he/she can select the corresponding ones using the JPLAS interface. Besides, if the teacher needs to select a specific topic for the problem code, he/she may select the code related to the topic from the JPLAS database. If there is not, the preferred code can be uploaded.

For evaluations, we first verify the correctness of this algorithm through applications of up to 100 Java codes, where the uniqueness of the correct answer for any blank was manually confirmed. Besides, we find that the number of blank elements reflects a proportional tendency to the number of statements in the code. Afterwards, we generate fill-in-blank problems using this algorithm, and assign them to students in the Java programming course in our department. These students have started learning Java programming through this course and do not have abundant experiences before. Based upon the results in two-year applications, we confirm the correlation between the number of correctly solved blanks and the final assignment score or the course grade.

The rest of this paper is organized as follows: Section II shows the functions for fill-in-blank problems in JPLAS. Sections III and IV present the blank element selection algorithm. Sections V examines the correctness of the algorithm. Section VI analyzes educational effects in Java programming course. Section VII introduces various related works. Section VIII concludes this paper with future works.

II. Fill-in-blank Problem Functions in JPLAS

In this section, we show the currently implemented functions for fill-in-blank problems in JPLAS.

A. Software Platform for JPLAS

In the JPLAS server, we adopt *Linux* for the operating system, *Tomcat* for the Web application server, *JSP/Servlet* for application programs, and *MySQL* for the database. The user can access JPLAS through a Web browser.

In the implementation of fill-in-blank problem functions, we adopt *JFlex* [7] and *jay* [8] that are both open source software. *JFlex* is a lexical analyzer for a Java code, which is also coded by Java. It transforms a code into a sequence of lexical units that represent the least elements to compose the code. It can classify each element in a code into either a reserved word, an identifier, a symbol, or an immediate data. For example, a statement `int value = 123 + 456;` is divided into `int`, `value`, `=`, `123`, `+`, `456`, and `;`. However, since *JFlex* cannot identify an identifier among a class, a method, or a variable, *jay* is used as well, as a syntactic parsing program based on the LALR method, which can identify an identifier.

B. Definitions of Terms for Fill-in-blank Problem

Here, we define several terms for the fill-in-blank problem. A *problem code* represents a Java code that has some blanks. A *blank* means an element to be filled in by a student. An *assignment* consists of a problem code with some blanks and their correct answers, a title, and a comment on the assignment. On the whole, several assignments are given to students in each course, where JPLAS can support multiple courses at the same time. Any registered teacher in JPLAS can generate new problems and assignments using the shared database.

C. System Utilization Procedure

The JPLAS functions for the fill-in-blank problem comprise *teacher functions* and *student functions*. The former functions cover the *code registration*, the *code and type selection*, the *blank element selection*, the *problem preview*, the *assignment generation*, and the *score reference*. The latter functions are in charge of the *assignment solution* and the *score reference*. These functions are used through the following steps:

1) A teacher uploads Java codes to the database.
2) A teacher selects one code from the database.
3) A teacher selects the *blanks* from the code to generate a *problem code* using the proposed algorithm.
4) A teacher registers an *assignment* for a course by selecting one problem code and describing its title and comment.
5) A student selects one assignment.
6) A student submits the answers to the blanks in the problem code.
7) The server verifies the answers and returns the results.
8) A student modifies the incorrect answers and resubmits them to the server, if necessary.
9) A teacher and a student refer to the score of the assignment.

The details of the functions will be briefly introduced in the following subsections.

D. Teacher Functions

1) Code Registration: Firstly, a teacher collects suitable Java source codes for fill-in-blank problems, and uploads them to the database of JPLAS. These codes should contain the elements, particularly reserved words, which are worth studying in the corresponding classes for code reading.

2) Code and Element Type Selection: Secondly, the teacher selects one code from the database, and the element types of blanks among reserved words, identifiers, or control symbols using the interface in Figure 1.

3) Blank Element Selection: Thirdly, the proposed blank element selection algorithm is applied to the code on the server to select the blank elements.

4) Problem Code Preview: Fourthly, the teacher previews the generated problem code using the interface in Figure 2, and adds a comment on it. Then, the problem code is stored in the database.

5) Assignment Registration: Lastly, the teacher registers a new assignment for the course by selecting one problem code in the database, and describing the title and the comment. Then, the assignment is stored in the database.

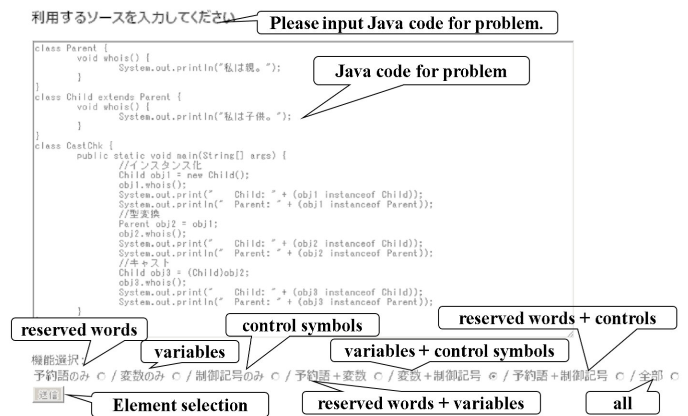

Fig. 1. Interface for code and element type selection.

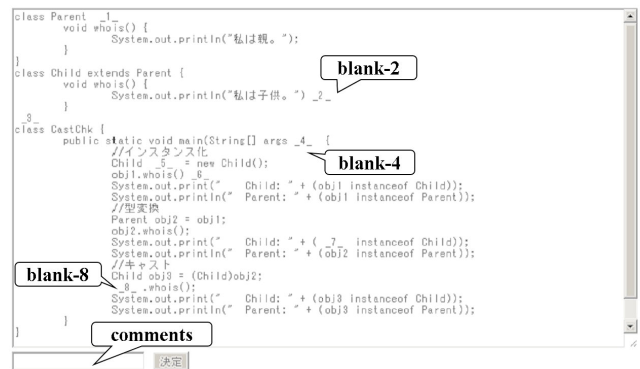

Fig. 2. Interface for problem code preview.

6) Score Reference: A teacher can check the answers from the students for the assignments in the course to evaluate their learning performance. For instant evaluations of assignment solutions, the teacher can overview the number of students that have solved each assignment and the average score among them. For detailed evaluations, the teacher can further examine the correctness of the answers and the number of answer submissions by every student using the interface in Figure 3.

Student index | Question index | Number of submissions

学籍番号	設問1	設問2	設問3	設問4	設問5	設問6	設問7	設問8	設問9	設問10	繰り返し回数
50000003	○	○	○	○	○	×	×	×	×	×	6
50000004	○	○	○	○	○	○	○	○	○	○	16
50000002	×	○	○	○	○	○	○	○	○	○	7
50000002	○	○	○	○	○	○	○	○	○	○	2
50000000	○	○	○	○	○	○	○	○	○	○	6
50000001	○	○	○	○	○	○	○	○	○	×	2
50000006	×	×	×	×	×	×	×	×	×	×	2
20000011	○	○	○	○	○	×	×	○	○	×	2
20000007	○	○	○	○	○	○	○	○	○	○	4
20000002	○	○	○	○	○	○	○	○	○	○	7
20000011	○	○	○	○	○	○	○	○	○	○	8
09421922	○	○	○	○	○	○	○	○	○	×	3
09421944	○	○	○	○	○	○	○	○	○	○	14

この課題の平均提出回数:6

Question result

Fig. 3. Interface for assignment answer results by students.

7) Database Management: The JPLAS database will retain the records of the user names and IDs of the teachers and the students, the course titles, the problem codes with

the correct answers, and the assignments with the titles and comments. When a new teacher starts using JPLAS, the system manager for JPLAS needs to register this teacher in the database to authorize him/her to register new courses and generate new problem codes and assignments.

When a teacher employs JPLAS in a course, he/she needs to register the course information such as the title and the student list in the database. Then, he/she can generate and register assignments for this course, give them to the students, and view their scores. Also, any student in the student list can access to the assignments, submit the answers, and check their own scores.

Any problem code generated by one teacher can be shared among the teachers using the database in JPLAS. Thus, it is expected that after this database is enriched with a variety of problem codes and assignments, the assignment preparation loads of a teacher can be drastically reduced by simply selecting existing problem codes or assignments from the database.

E. Student Functions

1) Assignment Selection: Firstly, a student can view the list of the registered courses and select one course following the access to the JPLAS server using a Web browser. Subsequently, the student can browse through the list of the assignments in the course that need to be solved from the answer submission status in the interface in Figure 4. Eventually, the student can select one assignment to answer by clicking the answer button.

Fig. 4. Interface for assignment list and submission status.

2) Answering Assignment: Secondly, the student can view the direction, the assignment index, the comment, the problem code with blanks, and the answer forms in the assignment using the interface in Figure 5, where the student can input the answers into the corresponding forms. In this case, an open source editor called *CodePress* is adopted to improve the readability of the problem code by using the highlighting function of this editor [9].

3) Automatic Rating: Thirdly, the student can submit the answers to the server for check by clicking either the "rating" button or the "finalizing" button. When the former button is clicked, the JPLAS server compares the answer of each blank with the correct one. It returns "OK" if they are matched, or otherwise "NG". When the latter button is clicked, the answers by the student are finalized and stored in the database with the number of submissions via clicking the rating button as well as the date/time. To complete the

Fig. 5. Interface for assignment answering.

assignment, clicking the "finalizing" button is a must. This number of submissions in the database can be used to evaluate the difficulty of the assignment and the performance of the student.

4) Score Reference: A student can check the performance for each assignment using the interface in Figure 6.

Fig. 6. Interface for score reference table.

F. Score Ranking Graph

A student can also realize his/her ranking among the students in the same course, in terms of the total number of correct answers for all the assignments using the *score ranking graph* in Figure 7. This graph is expected to encourage students to solve more problems aggressively by comparing the progress in solving assignments among the students.

III. Three Categories for Blank Element Selection Algorithm

In this section, we present the three categories to represent the constraints in selecting blank elements with unique answers for the blank element selection algorithm.

A. Group Selection Category

In the *group selection category*, all the elements related to each other in the code are grouped together so that at least one element from each group is not selected for blank-use. Five conditions are presented for this category in this paper. To elaborate, we use the following simple code.

```
1: class Sample1{
2:   public static void main(String args[]){
3:     int var1 = 10;
```

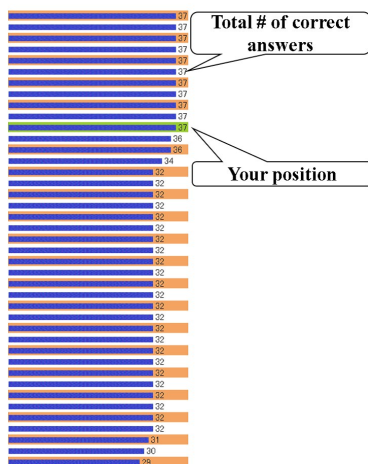

Fig. 7. Interface for score ranking graph.

```
4:      float var2 = sampleMethod(var1);
5:      System.out.println("indata="+var1+"
        outdata=" +var2);
6:    }
7:    static float sampleMethod(int p1){
8:      float tax = 1.08f;
9:      return p1*tax;
10:   }
11: }
```

(1) Identifier appearing two or more times in code

The multiple elements representing the same identifier with the same scope in the code are grouped together. A *scope* indicates the range in the code where a variable, a class, or a method is referred using the same name or identifier [10]. If all of such elements are blanked, a student cannot answer the original identifier. For example, in `Sample1`, `var1` appears three times with the same scope at lines 3, 4, and 5, which belong to this condition.

(2) Pairing reserved words composed of three or more elements

The three or more elements representing the reserved words in pairs are grouped together. If all of them are blanked, the unique correct answers may become too difficult or impossible to answer. Besides, one of those elements could be a desired hint to derive the other element for novice students such as the following two cases:

- switch - case - default
- try - catch - finally

(3) Data type for variables in equation

The elements representing the data types for variables in one equation are grouped together. For example, in `sum = a + b`, the data types of the three variables, `sum`, `a`, and `b`, must be the same. If a variable is casted like `sum = (int)a + b`, the casted data type `int` is also included in the group. In addition, if a method is included in an equation, like line 4 in `Sample1`, the data type of this method is also grouped together. Actually, `float` at lines 4 and 7 belong to this condition.

(4) Data type for method and its returning variable

The elements representing the data type of a method and its returning variable are grouped together. For example, in `Sample1`, `float` at lines 7 and 8 are grouped.

(5) Data type for arguments in method

The elements representing the data type of an argument in a method and its substituting variable are grouped together. For example, in `Sample1`, `int` at lines 3 and 7 belong to this condition through line 4.

The data type in (3)-(5) must be the same if at least one element in these groups is overlapped. Thus, after every group is found, the groups from (3)-(5) that contain an overlapped element are merged into one group.

B. Pair Selection Category

In the *pair selection category*, the elements appearing in the code in pairs are grouped together so that at least one element from each pair is not selected for blank-use. Four conditions of this category are illustrated as follows.

(1) Elements appearing continuously in statement

The two elements appearing continuously in the same statement in the code are paired. If both of them are blanked, their unique correct answers may not be guaranteed, or may become a significant challenge for novice students. Due to the same reason, the two elements connected with a dot (".") are also paired. For example, in `Sample1`, `static` and `float` at line 7 are paired. If it is removed, the following problem can be generated from line 7 in `Sample1`, which will provide the novice students with an exceedingly difficult demanding job:

```
7:    _1_  _2_  _3_  _4_  _5_  _6_ ) _7_
```

(2) Variables in equation

The elements representing any pair of the variables in an equation are also paired. If both are blanked, the unique correct answers become impossible because the reversed order is also grammatically correct. For example, for `sum = a + b`, `sum = b + a` is also feasible. If three or more variables are included in an equation, any pair of combinations can be found here.

(3) Pairing reserved words

The two elements are paired to represent the paring reserved words. If both are blanked, the unique correct answers may not be guaranteed, or may put a heavy burden on novice students. Besides, one of those elements can be a hint to derive the other one, including the following five paring reserved words:

- if - else

- do - while
- class - extends
- interface - extends
- interface - implements

(4) Pairing control symbols

The two elements representing a pair of control symbols, “(,)” (bracket) and “{, }” (curly bracket), are paired. Even if both are blanked at the same time, the code can be grammatically correct. Furthermore, novice students are expected to carefully check them in their codes to avoid making mistakes. For example, in `Sample1`, { at line 1 and } at line 11 are paired.

C. Prohibition Category

In the *prohibition category*, an element is prohibited from the blank selection because it does not satisfy the uniqueness with the high probability. This category involves four conditions. However, an element in a fixed sequence of elements indicating a specific meaning in a Java code, such as `public static void main` and `public void paint(Graphics g)`, is excluded from this category, for they should be mastered by students in advance.

(1) Identifier appearing only once in code

The selected element representing the identifier in this category appears only once in the code. If it is blanked, a student cannot answer the original identifier. For example, in `Sample1`, `Sample1` at lines 1 is prohibited.

(2) Operator

The element representing an operator such as the arithmetic operator: `=`, `+`, `-`, `*`, `/`, the comparative operator: `<`, `>`, `<=`, `>=`, `==`, `!=`, and the logical operator: `&`, `|`, `^`, `!` is selected for this category. If an operator is blanked, a student cannot answer the original one unless the proper explanation of the specification related to the operator is given. For example, in `Sample1`, `*` at line 9 is prohibited.

(3) Access modifier

The element representing an access modifier for an identifier is selected to this category. If it is blanked, either `public`, `protected`, or `private` can often be grammatically correct.

(4) Constant

The element representing a constant is selected for this category. If it is blanked, a student cannot answer the original constant. For example, in `Sample1`, `10` at lines 3 is prohibited.

IV. Blank Element Selection Algorithm

In this section, we propose a blank element selection algorithm using a graph generated from the category selections of the elements in the previous section.

A. Algorithm Overview

In this algorithm, the *constraint graph* is first generated from the given code. In this graph, a *vertex* represents a candidate blank element, and an *edge* does the constraint such that their incident elements cannot be blanked simultaneously for unique correct answers. Then, the *compatibility graph* is derived by taking the complement of the constraint graph. Finally, a maximal clique of the compatibility graph is sought to obtain a maximal set of blank elements with unique answers. This algorithm consists of the following four steps:

(1) Vertex generation for constraint graph
(2) Edge generation for constraint graph
(3) Compatibility graph generation
(4) Maximal clique extraction of compatibility graph

B. Vertex Generation for Constraint Graph

In the constraint graph, each vertex represents a candidate element for being blank. The candidate elements or vertices are extracted from the code through the lexical analysis using *JFlex* and *jay*. Each vertex contains the associated information in Table I, which is necessary for the category selection.

TABLE I
VERTEX INFORMATION.

item	content
symbol	symbol of element
line	row index of element
column	column index of element
count	number of element appearances
order	appearing order of element in code
group	statement group index partitioned by { and }
depth	number of { from top

Then, the vertices corresponding to the elements classified into the prohibition category in Section III-C are removed from the constraint graph.

C. Edge Generation for Constraint Graph

For the constraint graph, an edge is generated between any pair of the two vertices or elements that should not be blanked at the same time. These pairs are selected from the elements in the group selection category or the pair selection category in the previous section.

For each pair of elements in the *pair selection category*, an edge is simply generated between the two corresponding vertices. For each group of elements in the *group selection category*, one vertex among the elements is randomly selected first, and then, an edge is generated between this selected vertex and other individual vertices in the same group. Thus, at least this selected vertex is not selected for blank.

D. Example of Constraint Graph

Figure 8 illustrates the constraint graph for `sampleMethod` in `Sample1`. A *broken line* signifies that the two incident elements are grouped together by the group selection category, where the associated number represents the selecting condition in the category. For example, two `p1` are connected by condition (1) and

two `float` are by (4). A *straight line* signifies that they are grouped together by the pair selection category. For example, `static` and `float` are connected by (1), `p1` and `tax` are by (2), `(` and `)` are by (4). Moreover, a *broken circle* represents an element in the prohibition category that must not be selected as a blank element, where `=` and `*` are prohibited by (2), and `1.08f` is by (4).

Consequently, it is observed that `static` is included in this graph. Because `sampleMethod` is called without generating an object of the class for this method in `main`, it must be there. Being equipped with this grammar conception is critical and beneficial for the learning of students.

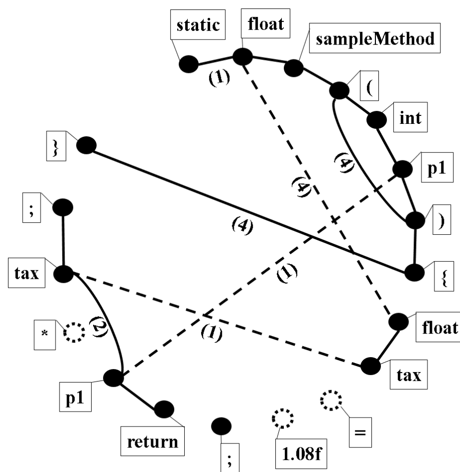

Fig. 8. Constraint graph for `sampleMethod` in `Sample1`.

E. Compatibility Graph Generation

By taking the complement of the constraint graph, the *compatibility graph* is generated to represent the pairs of elements that can be blanked simultaneously.

F. Maximal Clique Extraction of Compatibility Graph

Finally, a maximal clique of the compatibility graph is extracted by a simple greedy algorithm to find the maximal number of blank elements with unique answers. A *clique* of a graph represents its subgraph where any pair of two vertices is connected by an edge. The procedure of the algorithm is described as follows:

(1) Calculate the degree (= number of incident edges) of every vertex in the compatibility graph.
(2) Select one vertex among the vertices whose degree is maximum. If two or more vertices have the same maximum degree, select one randomly.
(3) If the selected vertex is a *control symbol* and the number of selected *control symbols* exceeds $1/3$ of the total number of selected vertices, remove this vertex from the compatibility graph and go to (5).
(4) Add the selected vertex as blank, and remove it as well as its non-adjacent vertices from the compatibility graph.
(5) If the compatibility graph becomes null, terminate the procedure.
(6) Go to (2).

G. Ratio of Control Symbols among Blanks

In the maximal clique procedure, (3) is introduced to sustain the total number of blank *control symbols*, because a code generally consists of a great quantity of control symbols. To investigate the proper threshold, we selected blank elements with different ratios for the set of 100 Java codes used in Section V, and examined average numbers of blank control symbols and other elements using the algorithm. As shown in Table II, the number of control symbols decreases and the number of other elements increases as this ratio becomes smaller, which is according to the condition (1) in the pair selection category. Then, we empirically selected $1/3$ as an appropriate ratio to generate fill-in-blank problems for novice students that have on average about three blanks for control symbols and eight blanks for other important elements. In future studies, we will investigate the effect of this ratio in Java programming learning performance.

TABLE II
NUMBER OF BLANK ELEMENTS BY DIFFERENT CONTROL SYMBOL RATIOS.

ratio	# of control symbols	# of others	total #
1/2	5.84	6.44	12.28
1/3	3.60	8.20	11.80
1/5	2.08	10.04	12.12
1/10	1.00	11.20	12.20

V. CORRECTNESS OF ALGORITHM

In this section, we verify the correctness of the blank element selection algorithm manually by applying it to 100 Java codes.

A. Uniqueness of Correct Answer

Firstly, we verify the uniqueness of grammatically correct answers for blank elements in the problem code that are selected by the algorithm. For this verification, we collected 100 Java codes from books and Web sites [11]-[16], where the number of statements in each code is distributed from 6 to 85, and 24 codes have multiple classes or methods. Then, we generate fill-in-blank problems by applying the algorithm, and asked four students in our group to solve them. These students are currently using Java in their researches and possess abundant experiences in Java programming.

The results show that for 97 codes, all the blanks selected by the algorithm have unique answers. For the remaining three codes, two variables can be exchanged between two blanks. The following code `Sample2` shows one such example. Grammatically, `outData2` and `outData1` can be filled at _4_ and _5_ respectively, although the reversed ones are in the original code. To avoid it, we additionally demonstrate the output result of this code as revealed in the last three lines.

```
1: public _1_ Sample2{
2:   public _2_ void main(String[] args){
3:     String _3_ = "abcdefgh";
4:     String _4_ = inData.substring(0, 5);
5:     String _5_ = inData.substring(3, 5);
6:     System._6_.println("out1= "+outData1);
7:      _7_.out.println("out2= "+outData2);
8:   }
```

```
9: _8_
//output result
//out1= abcde
//out2= de
```

B. Number of Statements and Number of Blank Elements

Secondly, we examine the relationship between the number of statements in a code and the number of blank elements selected by the algorithm. Figure 9 exhibits these numbers in 100 codes. This graph reflects that they are nearly proportional to one another, except for the first 10 codes. Based upon Table III, these codes have more statements composed of only curly brackets due to multiple classes/methods and/or multiple branches for conditions. Since we limit the number of control symbols, including curly brackets for blanks, the number of selected blanks becomes smaller if compared with the number of statements.

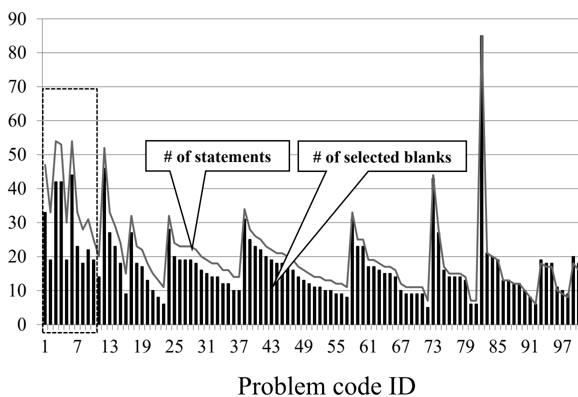

Fig. 9. Number of statements and number of blanks in 100 codes.

TABLE III
FEATURES OF 10 CODES WITH HUGE DIFFERENCE BETWEEN TWO NUMBERS.

code ID	# of blanks	# of statements	# of classes	# of methods	# of if, switch
1	33	47	1	3	8
2	19	33	1	3	3
3	42	54	3	9	0
4	42	53	1	4	3
5	19	30	1	1	2
6	44	54	3	8	0
7	23	30	3	3	0
8	18	28	1	1	2
9	22	31	2	5	0
10	19	25	1	1	7

C. Number of Blank Elements and Solution Difficulty

Thirdly, we evaluate the relationship between the number of blank elements in a fill-in-blank problem and the solution difficulty. For this purpose, we generated eight fill-in-blank problems from the same Java code where the number of blank elements changes from three to 10, and applied them to 41 sophomore students in our department. We asked them to solve the problems in descending number of blank elements within the limited time, so that they will not see the correct answers beforehand. Figure 10 offers the number of students who have correctly solved each number of blank elements and where the correlation is -0.73, which indicates the strong negative one.

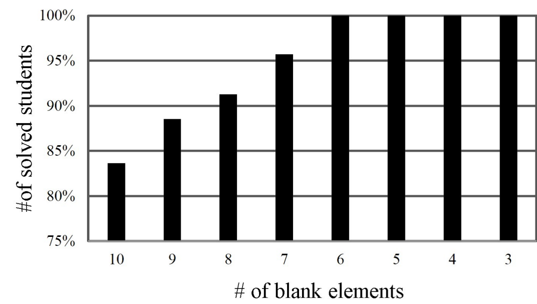

Fig. 10. Relationship between blank elements and solving students.

VI. EDUCATIONAL EFFECTS IN JAVA PROGRAMMING COURSE

In this section, we estimate educational effects in solving fill-in-blank problems by students who take the Java programming course in our department to evaluate the effectiveness. After implementing JPLAS and the proposed algorithm, we generated and assigned fill-in-blank problems to sophomore students taking the course in 2014 and 2015 who have started learning Java programming. Following this, the application result and the analysis in each year will be discussed below.

A. Application in 2014

First, we describe the application of fill-in-blank problems with 46 students in the 2014 course and the analysis results.

1) Overview of Fill-in-blank Problems and Solution Results: For this application, we collected 121 Java codes from the textbooks in [11]-[13] and the Web-sides in [14]-[16]. Then, we applied the algorithm and generated the fill-in-blank problems with the total of $1,552$ blanks.

Table IV displays the weekly summary of the assigned fill-in-blank problems and solving results with students. For example, for “variable/operator” in the first week, we used codes that consist of one class with the main method, standard outputs, variables with data types, and/or operators, in addition to control symbols. Then, our algorithm selects elements related to them for blanks. If the algorithm does not select any variable or operator, we will execute the algorithm with different random numbers, or replace this code with another one. For design pattern, data structure, and GUI, we collected codes implementing them so that students can study the structure of Java codes for important applications.

Table IV indicates that during the first three weeks, basic grammar related problems were assigned, and most students could solve them with a lesser submission. During the fourth week, the solving results became worse because the codes developed into harder and longer, and more blanks were selected. After the sixth week, rather advanced topics were explored where the codes have multiple classes and methods. Especially, the ninth week had the largest number of submissions, where data structure and algorithms were discussed. Most students tended to spend a long time to comprehend them.

TABLE IV
PROBLEMS IN EACH WEEK AND SOLVING RESULTS BY STUDENTS IN 2014.

week	# of codes	code topic	ave. # of blanks per code	# of solving students	ave. # of trials	ave. correct rate
1	24	variable/operator	8.41	46	2.59	99.88
2	21	conditional statement	7.00	46	1.96	99.36
3	28	class	9.75	46	6.35	98.84
4	10	object	15.80	46	13.57	97.52
5	3	quiz	20.00	46	12.03	90.71
6	5	file input/output	12.75	46	10.59	99.26
7	12	design pattern	15.50	30	10.95	98.67
8	5	interface	11.60	30	6.7	99.58
9	5	data structure/algorithms	29.20	20	21.26	99.16
10	8	GUI	20.12	16	10.29	99.24

2) Final Programming Assignment: As the final programming assignment for this course, each student is requested to freely select one application by himself/herself and to implement the Java code to realize it. For the implementation, any student is allowed to use existing codes on Web sites or others to complete it before the deadline. Then, many students selected applications related to games, painting tools, and face recognition tools and completed the codes.

To evaluate the final programming assignment, in the last class of the course, each student first explained the specification of his/her application to the teacher and the students, and demonstrated the implemented Java code using a PC to show its correct running. Then, the teacher and each student gave a score to the presenter on the usability or the uniqueness of the application with five points and on the implementation difficulty with five points. After that, these scores are tallied to calculate the *final point* with a 100-point scale, where the score from the teacher is weighted to 20% of the final point and the scores from the students are to 80%. Accordingly, the final point becomes the summary of the evaluations by a teacher and 45 students in terms of the usability or the uniqueness and the implementation difficulty of the code by the student, and is suitable to evaluate the programming ability in a short time.

3) Two Groups by Solved Blanks: After the course was completed, we analyzed the relationship between the numbers of correctly solved blanks and the final points among the students in Figure 11. A tendency is discovered that they are more correlated for the students solving more blanks (Group A) than for the students solving less blanks (Group B). Thus, we divided the 46 students into two groups by the equal number, and analyzed the solving performance in each group.

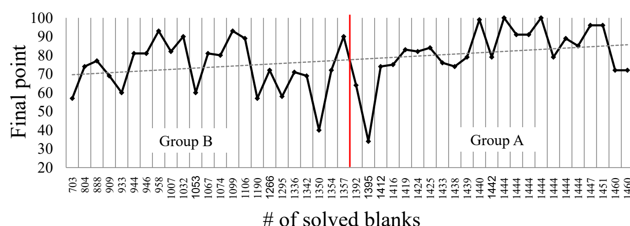

Fig. 11. Number of solved blanks and final points by students in 2014.

Table V shows the number of solved blanks, the average final point, and the average number of answer submissions for each code in the two groups. This table indicates that students in *Group A* solved more blanks with slightly increasing submissions, and achieved higher final points than students in *Group B* on average.

TABLE V
SOLVING PERFORMANCE OF TWO STUDENT GROUPS IN 2014.

group	A	B
# of students	23	23
# of solved blanks	1392-1460	703-1357
ave. final point	81.48	73.74
ave. # of submissions for one code	7.80	7.68

4) Correlation between Solved Blanks and Final Point: Then, we analyzed the correlation between the number of solved blanks and the final point in *Group A* and in *Groups B* individually. Figures 12 and 13 show the results for them respectively. The positive correlation ($r = 0.60$) exists for *Group A*, whereas no correlation ($r = -0.15$) does for *Groups B*. These results suggest that students need to solve a sufficient number of fill-in-blank problems in JPLAS to improve programming skills. If they stop solving them in the middle, their improvements may be extremely limited.

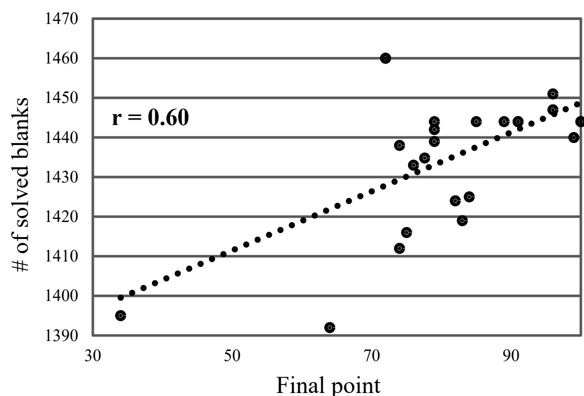

Fig. 12. Correlation between solved blanks and final points for *Group A* in 2014.

5) Correlation between Submission Times and Final Point: As mentioned above, JPLAS enables students to submit their answers and verify the correctness on the server without limitations, since it was designed to encourage students to study Java programming by themselves. Nevertheless, we believe that several students submit answers

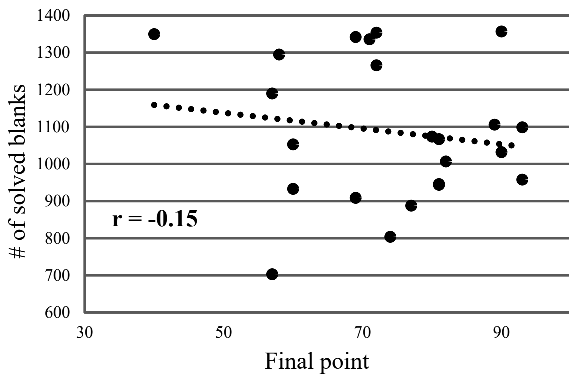

Fig. 13. Correlation between solved blanks and final points for *Group B* in 2014.

without seriously reading problem codes or considering the answers. Therefore, they cannot improve Java programming skills, even after they have solved a substantial of problems. Thus, we analyzed the correlation between the number of answer submissions and the final point in *Group A* and in *Group B* in Figures 14 and 15, respectively. The negative correlation ($r = -0.72$) exists for *Groups A*, whereas no correlation does for *Groups B*. They support our concern on lazy behaviors of low performing students. In the next Java programming course, we will inform students this fact to encourage them to solve the problems in JPLAS more carefully.

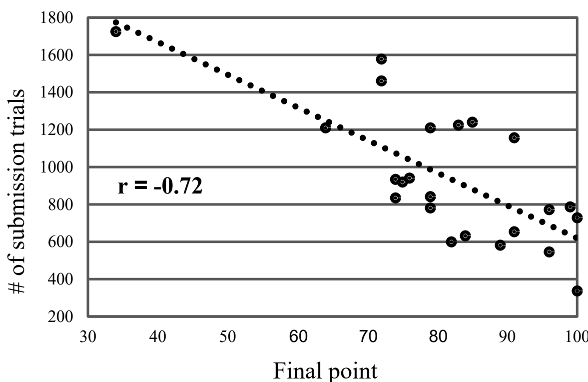

Fig. 14. Correlation between answer submissions and final points for *Group A* in 2014.

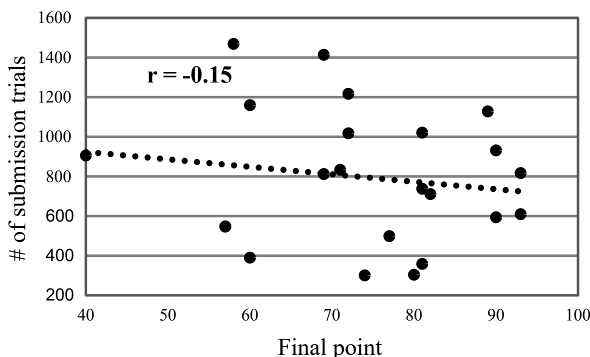

Fig. 15. Correlation between answer submissions and final points for *Group B* in 2014.

B. Application in 2015

Next, we describe the application to the 33 students in the 2015 Java programming course and the analysis results.

1) Fill-in-blank Problems in Course: In this year, we assigned a part of the problems generated in 2014 to the students, because in the course evaluation questionnaire, we found that the number of assigned problems was too much for specific students who have lost motivations of solving them completely. That is, we selected 16 problems that can be suitable for learning Java grammar.

Table VI manifests the overview of the assigned problems and solving results by the students. It is noticed that as the number of blanks increases, the correct answer rate decreases, where the correlation coefficient is $r = -0.57$. It confirms the claim in Section 5.3. For any code topic, a majority of the students correctly solved the assigned fill-in-blank problems in JPLAS that were generated by the proposed algorithm.

2) Course Grade: In the 2015 application, we adopted the *course grade* with a 100-point scale for each student to consider a different way of evaluating the programming abilities of students. The course grade was adopted in [29]. It was given by combining the scores of a quiz with 30%, a paper test with 30%, and a final programming assignment with 40%. Since the final programming assignment can evaluate the programming ability efficiently, it is designed to account for a higher percentage in the course grade.

3) Two Groups by Solved Blanks: As the 2014 application, we divided the 33 students into two groups and analyzed the solving performance in each group individually. Figure 16 offers the distribution of course grades and numbers of solving blanks among the students. Once more, it is observed that they are more correlated for the students solving more blanks (Group A) than for the remaining students (Group B).

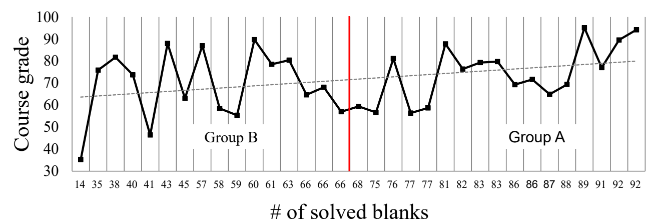

Fig. 16. Number of solved blanks and course grades by students in 2015.

Table VII shows the number of solved blanks, the average course grade, and the average number of answer submissions for each problem code in the two groups. The results convey that the students in *Group A* solved more blanks and achieved higher course grades.

4) Correlation between Solved Blanks and Course Grades: Then, we analyzed the correlation between the number of solved blanks and the course grade in each group, which is reflected by Figures 17 and 18 respectively. It is detected that the positive correlation ($r = 0.62$) exists for *Group A*, whereas no correlation ($r = 0.32$) does for *Groups B*. In other words, the same tendency is discovered as the previous year.

5) Correlation between Submission Times and Course Grades: Ultimately, we analyzed the correlation between the number of answer submissions and the course grade in each

TABLE VI
PROBLEMS AND SOLVING RESULTS BY STUDENTS IN 2015.

ID	code topic	# of blanks	# of solving students	ave. # of submissions	ave. correct rate
1	data type	8	30	9.17	83.75
2	data type	3	22	5.45	93.85
3	data type	5	19	2.42	95.79
4	data type	2	26	2.00	98.76
5	conditional statement	6	28	2.68	88.27
6	conditional statement	7	22	7.18	99.68
7	array	5	25	1.90	91.67
8	array	5	26	6.73	93.94
9	class	11	24	5.67	86.32
10	class	7	24	7.61	98.08
11	exception	3	19	4.63	96.49
12	file input/output	8	24	4.46	83.75
13	file input/output	6	31	6.03	98.39
14	quiz (data type)	4	24	6.50	100.00
15	quiz (conditional statement)	6	32	3.50	94.08
16	quiz (class)	6	27	4.15	100.00

TABLE VII
PERFORMANCE OF TWO STUDENT GROUPS BY PROBLEM SOLUTIONS IN 2015.

group	A	B
# of students	17	16
# of solved blanks	68-92	14-66
ave. course grade	74.51	68.97
ave. # of submissions for one code	6.25	3.87

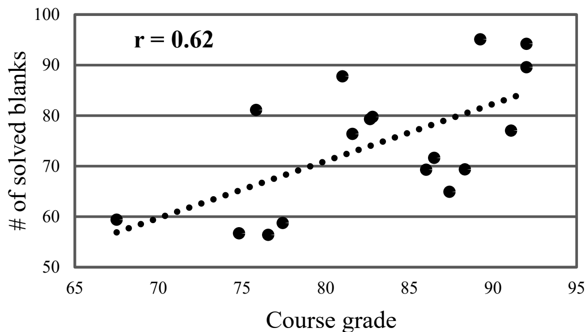

Fig. 17. Correlation between solved blanks and course grades for *Group A* in 2015.

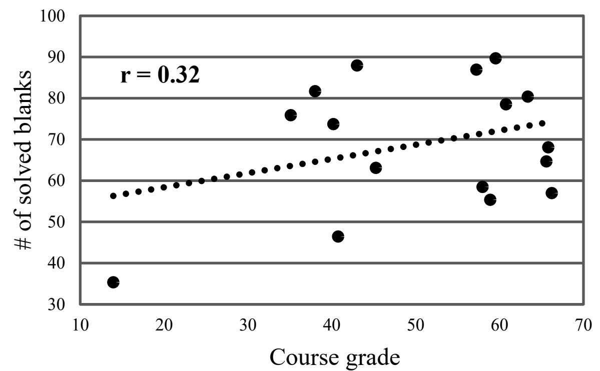

Fig. 18. Correlation between solved blanks and course grades for *Group B* in 2015.

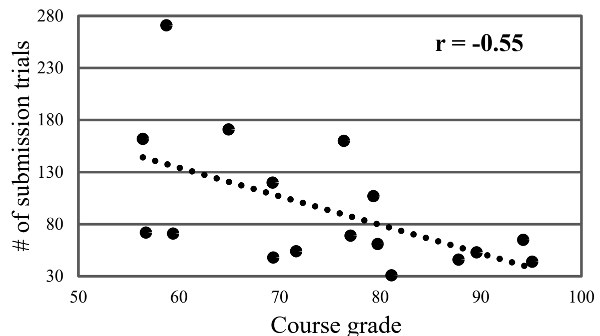

Fig. 19. Correlation between answer submissions and course grades for *Group A* in 2015.

group in Figures 19 and 20. Not surprisingly, the negative correlation ($r = -0.55$) exists for *Group A* and no correlation ($r = -0.11$) does for *Group B*, which is the same tendency as the previous year.

VII. RELATED WORKS

In this section, various related works are introduced to the fill-in-blank selection algorithm in JPLAS.

In [17], Kashihara et al. proposed a method of blanking an important point of data or control flow in a C code to make instructive fill-in-blank problems using *Program Dependence Graph (PDG)* without considering semantic aspects of the algorithm. PDG can represent the relationship of data dependency and control flows between commands using a graph. In future studies, it is considered to execute the use of PDG to extract important elements in the code.

In [18], Chang et al. proposed a programming learning system for novice students through three learning operators including cloze, modification, and extension of giving templates. The answers from students are assessed through matching with correct ones. In this system, applicable codes for students are limited since it is evaluated by high school students.

In [19], Kakugawa et al. presented a Web-based algorithm education system that can display the outline and the details of an algorithm, as well as to provide one blank statement in the algorithm description filled by students.

In [20], Bieg et al. demonstrated a Web-based tutorial

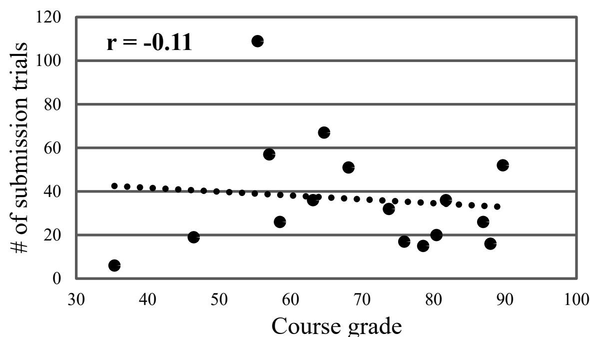

Fig. 20. Correlation between answer submissions and course grades for *Group B* in 2015.

system called *JOSH-online* to enable students to learn Java programming step by step and by interactive trial and error. It can relieve the programmer of initial difficulties in defining complete classes by executing program fragments directly. The design, the underlying interpreter, and its integration of the system are introduced.

In [21], Ala-Mutka surveyed programs of automatic assessment tools reported in literature that can be used to help teachers in grading assignments and students in studying programming. The dynamic versions include the functional assessment by running the program against several test data sets, the efficiency assessment by running time measurements or execution behavior analysis, and the testing skill of designing test cases. The static ones include the coding style in terms of readability, programming errors such as functional patterns and code redundancies, software metrics such as the number of statements and the cyclomatic number, and the structural requirement.

In [22], Taguchi et al. proposed a programming education assistant system to provide assignments suitable for individual students. The comprehension level and the motivation of a student is measured by using the collaboration filtering technique, which estimates the tendency and preference of a student from those of similar types using the same database. Since our algorithm can generate fill-in-blank problems with different levels for a variety of students, the exercise of this technique is expected to become exceedingly recommendable.

In [23], Shinkai et al. provided a C programming education assistant system on *Moodle* using fill-in-blank problems like in this paper. It extracts important elements in a code for questions using PDG.

In [24], Zacharis confirmed the effectiveness of virtual pair programming (VPP) using integrated desktop sharing and real-time communication on student performance and satisfaction in an introductory Java course.

In [25], Djenic et al. presented the development of a blended learning environment by the combination of classroom and Internet lessons for basic C and C++ programming courses. Contrary to JPLAS, this system provides online textbook contents with animations to enhance the motivation and efficiency of learning. Thus, it can reduce time for classroom lessons. In future works, we aim to develop the similar functions in JPLAS to promote self-studies of students.

In [26], Hauswirth et al. described the use of software clickers that allow for much richer problem types in a Java programming course, and introduced a pedagogical approach that allows students to learn from mistakes of their peers.

In [27], Zschaler et al. presented the details of *Salespoint* including functionality, architecture overview, example applications, and lessons. *Salespoint* is a Java-based framework for creating business applications to let students develop both technical skills and social skills such as collaborations with other developers.

In [28], Lopez et al. showed the correlation between the performance of code tracing tasks and that of code writing tasks in the introductory programming. In addition, the correlation between "explaining in plain English" tasks and code writing tasks is revealed. Their targets are novice programmers, as ours. A part of their code tracing task is similar to our fill-in-blank problem where some part of a code is removed to be filled in.

In [29], Hertz et al. presented *program memory traces* for tracing code by tracking what occurs in memory during a program execution, which is similar to the value trace problem in [34]. It is found that the trace-based teaching led to improvements in programming grades of students in introductory programming courses. The purpose is to encourage students to address code reading in a different way from the fill-in-blank problem. In [30], Kumar also found the effectiveness of this code tracing in improving code writing skills of students.

In [31], Busjahn et al. concluded that code reading, which can be a learning goal, is connected to comprehending programs and algorithms or algorithmic ideas as well as details such as semantics of constructs. That is, teaching code reading is regarded as a potential means to foster programming learning.

In [32], Allain, an engineer in Dropbox, introduced five ways to learn programming more effectively, where the first recommendation is "Look at the Example Code". He claimed "when you're first learning to program, you should make sure to look at, and try to understand, every example".

In [33], Mueller asserted "See the language used to perform specific tasks" among the seven key milestones for learning to code.

In [35], Brown et al reported a study to determine if programming educators form a consensus about which Java programming mistakes are the most common and found that educators formed a weak consensus about which mistakes are most frequent. Actually, students most often made mistakes in "mismatched parentheses", "calling method with wrong types", and "missing return statement", which were also reported in [36].

VIII. Conclusion

In this paper, a graph-based *blank element selection algorithm* for fill-in-blank problems in *Java Programming Learning Assistant System (JPLAS)* is presented. We verified the correctness of the algorithm manually by applying it to 100 Java codes, and confirmed the educational benefits in learning Java programming by assigning generated fill-in-blank problems to students in our Java programming course in two years. In future works, we will implement

hint functions to assist students who cannot solve fill-in-blank problems, consider methods to encourage students to aggressively solve more problems with the less number of submissions, and properly use JPLAS including fill-in-blank problems in Java programming courses.

REFERENCES

[1] N. Funabiki, Y. Matsushima, T. Nakanishi, K. Watanabe, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG International Journal of Computer Science, vol. 40, no.1, pp. 38-46, 2013.

[2] N. Funabiki, Y. Korenaga, Y. Matsushima, T. Nakanishi, and K. Watanabe, "An online fill-in-the-blank problem function for learning reserved words in Java programming education," Proceedings of The Eighth International Symposium on Frontiers of Information Systems and Network Applications, pp. 375-380, 2012.

[3] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W.-C. Kao, "Analysis of fill-in-blank problem solutions and extensions of blank element selection algorithm for Java programming learning assistant system," Lecture Notes in Engineering and Computer Science: Proceedings of The World Congress on Engineering and Computer Science 2016, WCECS 2016, 19-21 October, 2016, San Francisco, USA, pp. 237-242.

[4] K. Beck, Test-driven development: by example, Addison-Wesley, 2002.

[5] JUnit (online), http://www.junit.org/.

[6] M. R. Garey and D. S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, Freeman, New York, 1979.

[7] JFlex (online), http://jflex.de/.

[8] jay (online), http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html.

[9] CodePress (online), http://sourceforge.net/projects/codepress/.

[10] Scope (online), http://java.about.com/od/s/g/Scope.htm.

[11] M. Takahashi, Easy Java, Softbank Creative, 2013.

[12] Y. Kondo, Algorithm and data structure for Java programmers, Softbank Creative, 2011.

[13] H. Yuki, Introduction to design patterns using Java, Softbank Creative, 2006.

[14] ITSenka (online), http://www.itsenka.com/.

[15] tutorialspoint (online), http://www.tutorialspoint.com/java/index.htm.

[16] Java program samples (online), http://www7a.biglobe.ne.jp/~java-master/samples/.

[17] A. Kashihara, A. Terai, and J. Toyota, "Making fill-in-blank program problems for learning algorithm," Proceedings of International Conference on Computers in Education, pp. 776-783, 1999.

[18] K.-E. Chang, B.-C. Chiao, S.-W. Chen, and R.-S. Hsiao, "A programming learning system for beginners - a completion strategy approach," IEEE Transactions on Education, vol. 43, no. 2, pp. 211-220, 2000.

[19] H. Kakugawa and T. Mori, "Toward an algorithm education system on the Web," Proceedings of 13th International Conference on Systems Research, Informatics and Cybernetics, 2001.

[20] C. Bieg and S. Diehl, "Educational and technical design of a Web-based interactive tutorial on programming in Java," Science of Computer Programming, pp. 25-36, vol. 53, 2004.

[21] K. M. Ala-Mutka, "A survey of automated assessment approaches for programming assignments," Computer Science Education, vol. 15, no. 2, pp. 83-102, 2005.

[22] H. Taguchi, H. Itoga, K. Mouri, T. Yamamoto, and H. Shimakawa, "Programming training of students according to individual understanding and attitude," ISPJ Journal, vol. 48, no. 2, pp. 958-968, 2007.

[23] J. Shinkai, Y. Hayase, and I. Miyaji, "A study of generation and utilization of fill-in-the-blank questions for programming education on Moodle," IEICE Technical Report, ET, pp. 7-10, 2010.

[24] N. Z. Zacharis, "Measuring the effects of virtual pair programming in an introductory programming Java course," IEEE Transactions on Education, vol. 54, no. 1, pp. 168-170, 2011.

[25] S. Djenic, R. Krneta, and J. Mitic, "Blended learning of programming in the Internet age," IEEE Transactions on Education, vol. 54, no. 2, pp. 247-254, 2011.

[26] M. Hauswirth and A. Adamoli, "Teaching Java programming with the Informa clicker system," Science of Computer Programming, pp. 499-520, vol. 78, 2013.

[27] S. Zschaler, B. Demuthb, and L. Schmitz, "Salespoint: a Java framework for teaching object-oriented software development," Science of Computer Programming, pp. 189-203, vol. 79, 2014.

[28] M. Lopez, J. Whalley, P. Robbins, and R. Lister, "Relationships between reading, tracing and writing skills in introductory programming," Proceedings of The Fourth International Workshop on Computing Education Research, pp. 101-112, 2008.

[29] M. Hertz and M. Jump, "Trace-based teaching in early programming courses," Proceedings of The 44th ACM Technical Symposium on Computer Science Education, pp. 561-566, 2013.

[30] A. N. Kumar, "A study of the influence of code-tracing problems on code-writing skills," Proceedings of The 18th ACM Conference on Innovation and Technology in Computer Science Education, pp. 183-188, 2013.

[31] T. Busjahn and C. Schulte, "The use of code reading in teaching programming," Proceedings of The 13th Koli Calling International Conference on Computing Education Research, pp. 3-11, 2013.

[32] A. Allain (online), "5 ways you can learn programming faster," http://www.cprogramming.com/how_to_learn_to_program.html.

[33] J. P. Mueller (online), "Coding by the Book: 5 Tips for Learning How to Program From a Book," https://blog.newrelic.com/2015/03/18/learn-code-programming-book/.

[34] K. K. Zaw, N. Funabiki, and W.-C. Kao, "A proposal of value trace problem for algorithm code reading in Java programming learning assistant system," Information Engineering Express, vol. 1, no. 3, pp. 9-18, 2015.

[35] N. C. C. Brown and A. Altadmri, "Investigating novice programming mistakes: educator beliefs vs student data," Proceedings of The Tenth Annual Conference on International Computing Education Research, pp. 43-50, 2014.

[36] A. Altadmri and N. C. C. Brown, "37 million compilations: investigating novice programming mistakes in large-scale student data," Proceedings of The 46th ACM Technical Symposium on Computer Science Education, pp. 522-527, 2015.

Nobuo Funabiki received the B.S. and Ph.D. degrees in mathematical engineering and information physics from the University of Tokyo, Japan, in 1984 and 1993, respectively. He received the M.S. degree in electrical engineering from Case Western Reserve University, USA, in 1991. From 1984 to 1994, he was with Sumitomo Metal Industries, Ltd., Japan. In 1994, he joined the Department of Information and Computer Sciences at Osaka University, Japan, as an assistant professor, and became an associate professor in 1995. He stayed at University of Illinois, Urbana-Champaign, in 1998, and at University of California, Santa Barbara, in 2000-2001, as a visiting researcher. In 2001, he moved to the Department of Communication Network Engineering (currently, Department of Electrical and Communication Engineering) at Okayama University as a professor. His research interests include computer networks, optimization algorithms, educational technology, and Web technology. He is a member of IEEE, IEICE, and IPSJ.

Tana received the B.S. degree in computer science and technology from Inner Mongolia University, China, in 2006, and the M.S. degree in communication network engineering from Okayama University, Japan, in 2013, respectively. She is currently a Ph.D. candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and Web application systems. She is a member of IEICE.

Khin Khin Zaw received the B.E. degree in information technology from Technological University (HmawBi), Myanmar, in 2006, and the M.E. degree in information technology from Mandalay Technological University, Myanmar, in 2011, respectively. She is currently a Ph.D. candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and Web application systems. She is a member of IEICE.

Nobuya Ishihara received the B.S. degree in Mathematics from Okayama University, Japan, in 1989, and the M.S. degree in communication network engineering from Okayama University, Japan, in 2015, respectively. In 1990, he joined Okayama Information College as a lecturer. He stayed at Gyosei International School, UK, in 1991-1992, as an assistant research fellow. He is currently a Ph.D. candidate in Graduate School of Natural Science and Technology at Okayama University, Japan. His research interests include educational technology and Web application systems. He is a member of IEICE.

Wen-Chung Kao received the M.S. and Ph.D. degrees in electrical engineering from National Taiwan University, Taiwan, in 1992 and 1996, respectively. From 1996 to 2000, he was a Department Manager at SoC Technology Center, ERSO, ITRI, Taiwan. From 2000 to 2004, he was an Assistant Vice President at NuCam Corporation in Foxlink Group, Taiwan. Since 2004, he has been with National Taiwan Normal University, Taipei, Taiwan, where he is currently a Professor at Department of Electrical Engineering and the Dean of School of Continuing Education. His current research interests include system-on-a-chip (SoC), flexible electrophoretic display, machine vision system, digital camera system, and color imaging science. He is a senior member of IEEE.