Analysis of Fill-in-blank Problem Solutions and Extensions of Blank Element Selection Algorithm for Java Programming Learning Assistant System

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

***Abstract*—A Web-based *Java Programming Learning Assistant System (JPLAS)* has been developed in our group to advance Java programming educations. JPLAS provides *fill-in-blank problems* for Java novice students to study grammar and basic programming skills by filling in the blank elements in a high-quality code. In this paper, we first analyze solution results of students in the Java programming course and the correlation between the number of blanks in a problem and the correct answer rate of the students. Then, we extend the *blank element selection algorithm* to increase the number of blanks and control the problem difficulty by changing it. This algorithm has been proposed to generate a feasible fill-in-blank problem such that any blank has the grammatically correct and unique answer. To verify the effectiveness, we apply the extended algorithm to 58 Java codes for the fundamental data structure or algorithms, and confirm that the extensions can increase the number of blanks and control the problem difficulty.**

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

I. Introduction

JAVA, as a reliable and portable object-oriented programming language, has been extensively used in a variety of industries, including mission critical systems at large enterprises and small-sized embedded systems for real time controls. The cultivation of Java programming engineers has been highly demanded amongst industries. Hence, a number of universities and professional schools have designed Java programming courses to deal with these demands.

To advance Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [1]-[5]. As a function, JPLAS provides the *fill-in-blank problem* to support self-studies of students who have just started learning Java programming. The goal of this problem is to encourage students learning the grammar and basic programming skills through *code reading*.

In a fill-in-blank problem, a Java code with several blank elements is shown to each student, where he/she needs to fill in the blanks. This *problem code* should be of high-quality 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. To be more precisely, a *reserved word* is a fixed sequence of characters that has been defined in the grammar to represent a specified function, and must 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* intends other grammar elements such as grammar elements such as “ . ” (dot), “ : ” (colon), “ ; ” (semicolon) , “ (,) ” (bracket), “ {, } ” (curly bracket).

To help a teacher to prepare fill-in-blank problems in JPLAS, we have proposed the *blank element selection algorithm* to generate a fill-in-blank problem from a given code such that any blank has the grammatically correct and unique answer [2]. First, in this algorithm, we generate a *compatibility graph* by selecting any candidate element for a blank in the code as a *vertex*, and connecting any pair of vertices by an *edge* if they can be blanked together. For this purpose, we define the conditions that a pair of elements cannot be blanked simultaneously. Then, we extract a *maximal clique* [6] of the compatibility graph, which becomes a maximal set of proper blank elements. Empirically, we have observed that a fill-in-blank problem will become more difficult when a larger number of elements are blanked. Therefore, by blanking a subset of selected elements of the algorithm, we can generate a variety of fill-in-blank problems with different difficulty levels.

In this paper, firstly, we analyze solution results of students in the Java programming course in our department. Also, we observe the correlation between the number of blanks in a problem and the correct answer rate of the students. Secondly, we extend the blank element selection algorithm to increase the number of blanks and control the problem difficulty by changing the number of blanks. To verify the effectiveness, we apply our extended algorithm to 58 Java codes for the fundamental data structure or algorithms, and confirm that our extensions can increase the number of blanks and control the problem difficulty.

This paper is organized as follows: Section II and Section III introduce the fill-in-blank problem in JPLAS and reviews the blank element selection algorithm, respectively. Section IV analyzes solution results of fill-in-blank problems submitted by students. In Section V, three extensions of the blank element selection algorithm are presented. Lastly, Section VI concludes this paper with futures.

II. Fill-in-blank Problem in JPLAS

In this section, we review the fill-in-blank problem in JPLAS.

A. Software Platform for JPLAS

Based upon a Web application system, in the JPLAS server, we adopt *Linux* for the operating system, *Tomcat*

Funabiki, Tana, Zaw, and Ishihara are with the Department of Electrical and Communication Engineering, Okayama University, 3-1-1 Tsushimanaka, Okayama, 700-8530, Japan e-mail: funabiki@okayama-u.ac.jp. Kao is with the Department of Electrical Engineering, National Taiwan Normal University, Taipei, 106, Taiwan, e-mail: jungkao68@gmail.com.

TABLE I
VERTEX INFORMATION IN CONSTRAINT GRAPH.

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 the code
group	statement group index partitioned by { and }
depth	number of { from top

for the Web application server, *JSP/Servlet* for application programs, and *MySQL* for the database. For the fill-in-blank problem, we adopt open source software *JFlex* [7] and *jay* [8]. *JFlex* is a lexical analyzer generator for a Java code, which is also coded by Java. It transforms a code into a sequence of lexical units that represent the least meaningful elements to compose the code. It can classify each element in the 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 `;`. Unfortunately, *JFlex* cannot identify an identifier among a class, a method, or a variable. Thus, *jay* is applied as well. Since *jay* is a syntactic parsing program based on the LALR method, it can identify an identifier.

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

The definitions of terms for the fill-in-blank problem are listed as follows. A *problem code* represents a Java code involving some blanks. A *blank* indicates 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. Generally, several assignments will be given to students in each course, where JPLAS can support multiple courses at the same time. All registered teachers in JPLAS can generate and register new problems and assignments using the shared database.

III. REVIEW OF BLANK ELEMENT SELECTION ALGORITHM

In this section, we review the *blank element selection algorithm* [2] using the *constraint graph* that is generated to describe the constraints in the blank element selection.

A. Vertex Generation for Constraint graph

In the constraint graph, each vertex signifies a candidate element for being blank. The candidate elements or vertices are extracted from the Java code through the lexical analysis using *JFlex* and *jay*. Each vertex contains the associated information in Table I that is necessary for the following edge generation.

B. Edge Generation for Constraint graph

An edge is generated between any pair of two vertices or elements that should not be blanked at the same time. There are three categories to represent the constraints in selecting blank elements with unique answers:

1) Group Selection Category: In the *group selection category*, all the elements related to each other in the code are grouped together. First, in each group, one vertex is randomly selected. Then, edges are generated between this vertex and the other vertices to confirm that at least this selected element is not selected for blank. Five conditions are included in this category.

(1) Identifier appearing two or more times in the code

The multiple elements representing the same identifier of a variable, a class, and a method by using the same name, are grouped together. If all such elements are blanked, a student cannot answer the original identifier

(2) Pairing reserved words which are 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 answers may become too difficult 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.

(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.

(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.

2) Pair Selection Category: In the *pair selection category*, the elements appearing in the code in pairs are grouped together. For each pair, an edge is simply generated between the two corresponding vertices to assure that at least one element is not selected for blank.

(1) Elements appearing continuously in a statement

The two elements appearing continuously in the same statement are paired in the code. If both of them are blanked, their unique correct answers may not be guaranteed and the fill-in-blank problem may become too difficult for novice students. The two elements connected with a dot (" . ") are also paired.

(2) Variables in equation

The elements representing any pair of the variables in an equation are paired. If both are blanked, it will become impossible to access the unique answers. For example, for `sum = a + b`, `sum = b + a` is also feasible.

(3) Pairing reserved words

The two elements representing the paring reserved words are paired. If both are blanked, the unique correct answers may not be guaranteed. The following five paring reserved words are considered:

- if-else
- do-while
- class-extends
- interface-extends
- interface-implements

(4) Pairing control symbols

The two elements representing a pair of control symbols, namely " (,) " (bracket) and " {, } " (curly bracket), are

paired. The novice students should carefully check them in their codes to decrease the amount of mistakes.

3) 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. There are three conditions for this category. 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, because they should be mastered by students.

(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.

(2) Access modifier

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

(3) Constant

The element representing a constant is selected for this category. If it is blanked, a student cannot answer the original constant.

C. Compatibility Graph Generation

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

D. 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 from the given Java code. A clique of a graph represents its subgraph where any pair of two vertices is connected by an edge. The procedure for our algorithm is described as follows:

1) Calculate the degree (= number of incident edges) every vertex in the compatibility graph.
2) Select one vertex among the vertices whose degree is the 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 for blank, and remove it as well as its non-adjacent vertices of the compatibility graph.
5) If the compatibility graph becomes null, terminate the procedure.

E. Fill-in-blank Problem Generation

In the maximal clique procedure, 3) is used to sustain the total number of blank control symbols, because a code is generally composed of plenty of control symbols. Here, we examined the average number of blanks for control symbols and other symbols by the algorithm. Then, we empirically selected $1/3$ as an appropriate ratio to generate the feasible *fill-in-blank problems* for novice students. However, in these condition, the generated fill-in-blank problems can be solved without reading out the code if students are familiar with Java grammar.

TABLE II
ASSIGNED PROBLEMS AND SOLUTION RESULTS.

ID	key grammar in code	# of blanks	# of students	ave. # of trials	correct rate (%)
1	array (1)	8	30	9.17	83.75
2	array (2)	8	24	4.46	83.75
3	method (1)	11	24	5.67	86.32
4	method (2)	6	28	2.68	88.27
5	repeat (1)	5	25	1.9	91.67
6	variable	3	22	5.45	93.85
7	repeat (2)	5	26	6.73	93.94
8	method (3)	6	32	3.5	94.08
9	data type (1)	5	19	2.42	95.79
10	exception	3	19	4.63	96.49
11	class	7	24	7.61	98.08
12	method (4)	6	31	6.03	98.39
13	data type (2)	2	26	2	98.76
14	branch	7	22	7.18	99.68
15	method (4)	6	27	4.15	100
16	data type (3)	4	24	6.5	100
	total/ave.	92	25.19	5.00	93.93

TABLE III
TWO STUDENT GROUPS BY PROBLEM SOLUTIONS.

group	A	B
# of students	17	16
# of solved blanks	68 - 92	14 - 66
ave. course grade	74.51	68.97
ave. # of submission trials	6.25	3.87

IV. ANALYSIS OF FILL-IN-BLANK PROBLEM SOLUTIONS

In this section, we analyze solution results of fill-in-blank problems offered by students in our Java programming course.

A. Fill-in-blank Problem Solution Results

We collected Java codes from textbooks and Web sites [9]-[13], and generated 16 problems with the total of 92 blanks by applying the blank element selection algorithm. Afterwards, we asked students to solve them using JPLAS. Table II shows the assigned problems and solution results by students. In general, as the number of blanks increases, the correct answer rate decreases, where the correlation coefficient is $r = -0.57$.

B. Final Grades and Two Student Groups

After the course was finished, we evaluate the final grade of each student by one programming assignment (40%), several quizzes (30%), and the final paper test (30%). For the programming assignment, each student was requested to freely select a topic such as a game, a paint tool, and a face recognition, and write a Java code to implement it. In the last class, each code was evaluated on a 100-point scale by the teacher and the students in terms of complexity, completeness, and uniqueness. Next, we classified the students into two groups such that each group has the same number of students, considering the number of solved blanks in each group. Table III exhibits the statistics in each group.

C. Correlation between Solutions and Final Grades

First, we analyze the correlation between the number of solved blanks and the course grades among the students in group A and in group B as exhibited in Figures 1 and 2. The relatively strong correlation ($r = 0.62$) exists for group A,

ISBN: 978-988-14047-1-8
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

whereas the weak correlation ($r = -0.32$) does for group B. These results indicate that to improve Java programming skills, solving a sufficient number of fill-in-blank problems in JPLAS will be required. If the students stop solving them in the middle, their improvements will be suspended.

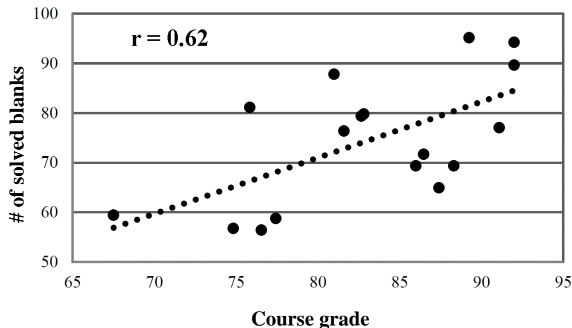

Fig. 1. Correlation between problem solutions and course grades for group A.

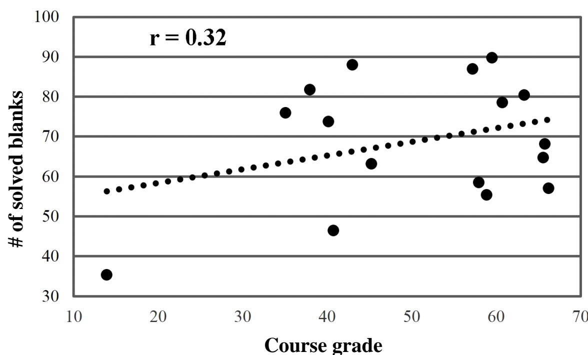

Fig. 2. Correlation between problem solutions and course grades for group B.

D. Correlation between Answer Submission Trials and Final Grades

In JPLAS, students are allowed to continuously repeat submissions of their answers to check the correctness at the server, because JPLAS has been designed to encourage students to study Java programming by self-learning. However, we suspect that unfortunately, students may submit answers without thinking them carefully. As a result, they may not be able to advance Java programming skills, despite of the number of problems they solved. Thus, we analyze the correlation between the number of submission trials and course grades in group A. Figure 3 reveals that the negative correlation ($r = -0.55$) exists between them, which supports our concern of the careless behaviors of students. We will notice this fact to students in the Java programming course to lead them to solve the problems with a less number of submissions.

V. Extensions of Blank Element Selection Algorithm

In this section, we propose three extensions of the blank element selection algorithm.

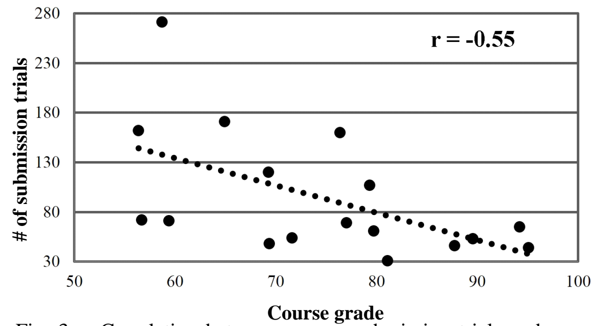

Fig. 3. Correlation between answer submission trials and course grades for group A.

TABLE IV
Operators for conditional expressions.

operator	example	operator	example
<	a<b	++	a++
<=	a<=b	--	a--
>	a>b	!	!a
>=	a>=b	+=	a+=b
==	a== b	-=	a-=b
!=	a!= b	*=	a*=b
&&	a&&b	/=	a/=b
||	a||b	%	a% b

A. Operators for Conditional Expressions

To assist students to understand the implementation of a logic or an algorithm in a Java code, we include elements representing *operators* in conditional expressions into blank elements. Table IV shows the corresponding 16 operators. To satisfy the uniqueness of the correct answers and avoid becoming too difficult for novice students, we classify all the operators in one conditional expression into the same group in the group selection category.

B. Introduction of Two Parameters

To adjust the difficulty of the generated fill-in-blank problem by controlling the ratio between the number of blank elements and non-blank ones in the problem code, we introduce the two parameters, namely BG (blank gap number) and CB (continuous blank number).

1) Blank Gap Number: The non-blanked elements in a problem code become hints to solve the fill-in-blank problem. As more non-blanked elements exist between blanked elements, it becomes easier. Thus, we try to control the difficulty of the problem by changing the number of non-blanked elements between blanked ones by introducing the *blank gap number* BG. To realize it, for the constraint graph, we generate an edge for each vertex with every vertex in the same statement in the code that exists within its BG neighbors, which is the modification of the condition (1) for the pair selection category. Here, we note that the previous algorithm actually adopts $BG = 1$ where at least one non-blanked element exists. For example, in the case of $BG = 2$, `bubbleSort` at line 1 has an edge with `static`, `void`, `(`, and `int` so that at least two non-blanked elements exist in the problem code.

2) Continuous Blank Number: On the contrary, as more blanked elements continue in a problem code, it becomes harder. Thus, we also try to control the difficulty by changing

ISBN: 978-988-14047-1-8
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

the number of continuously blanked elements by introducing the *continuous blank number* CB. In addition, we note that when CB is 2 or larger, BG must be set 0. The following procedure describes the extension to realize it:

(1) Find a solution by applying the blank element selection algorithm with $BG = 0$.
(2) Change the last blanked element into a non-blanked one if the number of continuously blanked elements exceeds CB.

C. Improvement of Edge Generation

The analysis in the previous section showed that as the number of blanks increases, the rates of students with correct answers decreases. Consequently, even for the same Java code, the number of blanks determines the difficulty of the fill-in-blank problem. To select a larger number of blanks, we improve the edge generation method for the group selection category in the constraint graph. Instead of randomly selecting one vertex in the same group, we select the vertex that has the largest number of incident edges. Then, if this vertex is not selected for the blank, other vertices can be selected for blanks.

1) Improved Edge Generation Procedure: The following procedure explains the details:

1) Generate the edge between two vertices for each vertex pair in the pair selection category.
2) Sort the vertex groups for the group selection category in the descending order of the group size.
3) Select one vertex for each group in 2) from the top by the following procedure:
   (1) Calculate the degree of the vertices in the group.
   (2) Select the vertex that has the largest degree. If two or more vertices have the same largest degree, randomly select one among them.
   (3) Generate the edges between the vertex in (2) and the other vertices in the group.

2) Example: In the following code for `bubbleSort`, the five `int` in lines 1, 2, 3, 5, and 8 are grouped by the group selection category, where both the variable and the subscript for `array` must be `int` from line 1. Subsequently, at least one `int` must not be selected as the blank element for the unique correct answer. With the employment of this group, we elaborate the improved edge generation method for the constraint graph.

```
1:  public static void bubbleSort(int[] array){
2:    for (int i=array.length-1; i>0; i--)
3:      for (int j=0; j<i; j++)
4:        if (array[j]>array[j+1]){
5:          int tmp=array[j];
6:          array[j]=array[j+1];
7:          array[j+1]=tmp;
8:          for(int k:array){
9:            System.out.print();
10:           System.out.print(",");
11:         }
12:         System.out.println();
13:       }
14:     }
15:   }
16: }
```

Figure 4 illustrates the corresponding five vertices and their incident vertices selected in the pair selection category for $BG = 2$. For example, `int` in line 1 has edges with `(`,`[`, and `]` where `bubbleSort` is not included in the constraint graph due to the condition (1) for the prohibited category. These edges are described by the straight lines with (G). Then, `int` in line 8 has the largest degree 4 among them. Therefore, we select `int` in line 8, and generate the edges between this vertex and the other four vertices for `int` that are described by the dotted lines with (P).

TABLE V
AVERAGE NUMBER OF BLANKS FOR DIFFERENT BG AND CB.

parameter	previous	extended				
BG	1	3	2	1	0	0
CB	1	1	1	1	2	3
# of blanks	24.76	19.12	22.16	25.95	39.98	40.06

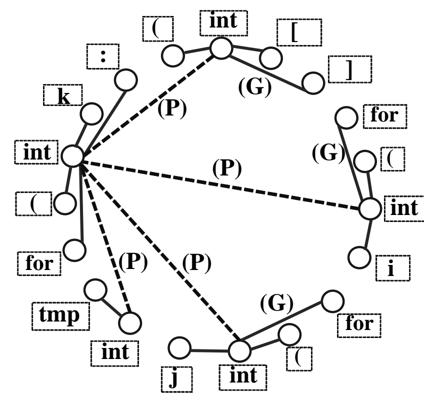

Fig. 4. Example of improved edge generation method for `bubbleSort`.

D. Evaluation of Blank Number Change

First, we evaluate changes of the number of selected blanks by the proposed extensions using 58 Java codes for fundamental data structure or algorithms in textbooks and Web sites [13]-[22]. We apply the previous algorithm and the extended algorithm when BG and CB is changed from 0 to 3 respectively. Table V offers the average number of blanks for the 58 problem codes found by them. The previous algorithm assumes $BG = 1$ and $CB = 1$, and the extended algorithm must adopt $CB = 1$ for $BG \geq 1$, because at most one blank element can be selected continuously to have BG non-blank elements between blank ones, and $BG = 0$ for $CB \geq 2$, because two or more blank elements can be selected continuously.

Table V indicates that the extended algorithm with $BG = 1$ and $CB = 1$ can increase the number of blanks slightly by selecting better edges among them of the constraint graph from the previous algorithm. It also reflects that the larger BG can decrease the number of blanks and the larger CB can increase it. Hence, we confirm that we can control the difficulty of generating fill-in-blank problems by changing the value of these newly introduced parameters.

E. Evaluation of Solution Performance

Then, we evaluate the solution performance of students for fill-in-blank problems generated by applying the extended algorithm to three Java codes where we change the two parameter values while adopting the other extensions.

ISBN: 978-988-14047-1-8
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE VI
PARAMETER VALUES FOR PROBLEM GENERATIONS.

parameter	L1	L2	L3
BG	3	1	0
CB	1	1	3

TABLE VII
STATISTICS OF GENERATED PROBLEMS.

Java code	LOC	# of blanks		
		L1	L2	L3
Euclid	12	6	9	16
TrialDiv	19	14	19	38
ModExp	13	11	14	20

1) Problem Generation: As problem Java codes, we adopted Java codes related to the RSA algorithm, namely *Euclid* (calculate the GCD of two arguments using the Euclid method), *TrialDiv* (calculate the GCD using the trial division method), and *ModExp* (calculate the modulo exponentiation of a big integer) [22]. For the two parameter values in the algorithm extension, we used the three sets in Table VI to generate problems with three levels, L1 (easy), L2 (middle), and L3 (hard), from each code. Table VII shows the LOC (the number of lines) in the problem code, the number of blanks in each problem at each difficulty level. As LOC is larger, the number of blanks increases for any difficulty level.

2) Problem Assignment to Students: Then, we divided the 33 students into three groups, A, B, and C, with the equal number randomly, and assigned one level for each problem to each group so that every student solves the different problems with the different levels. Thus, any student solved any level in our problem assignment. These students are currently taking *Network Security* course and completed *Java Programming* course in the last semester.

3) Solution Performance of Students: Table VIII shows the solving problem level and the percentage of the correctly solved blanks by the students in each group for each code. This table indicates that in each group, the percentage for *TrialDiv* is smaller than the others at any difficulty level (L). Table IX shows the average correct rate for each group, each problem, and each level. First, in the groups, Group C has the highest average correct rate, where we can say that Group C is the best student group. Then, in the problems, the average correct rate for *TrialDiv* is the lowest, because this problem has the larger LOC and the larger number of blanks than other problems at any difficulty level. Finally, in the levels, L3 has the lowest average correct rate. However, L2 has the higher rate than L1, because Group B solved better than Group A for *Euclid* and Group C solved better than Group B for *ModExp*. We need to further investigate the relationship between the two parameter values for the difficulty level and the average correct rate of students, which will be in our future works.

TABLE VIII
SOLUTION PERFORMANCE FOR EACH GROUP.

Java code	*Euclid*		*TrialDiv*		*ModExp*	
Group	L	correct rate	L	correct rate	L	correct rate
A	1	76%	2	71%	3	75%
B	2	87%	3	67%	1	73%
C	3	80%	1	78%	2	84%

TABLE IX
AVERAGE SOLUTION PERFORMANCE.

Group		Problem		Level	
A	73%	*Euclid*	81%	L1	76%
B	71%	*TrialDiv*	70%	L2	79%
C	80%	*ModExp*	77%	L3	71%

VI. CONCLUSION

In this paper, we first presented an analysis of solution results of fill-in-blank problems in *Java Programming Learning Assistant System (JPLAS)* by students taking the Java programming course, and showed that to advance Java programming skills, students need to solve a sufficient number of problems with a less number of answer submission trials. Next, we introduced three extensions of the *blank element selection algorithm*, and evaluated changes of the number of blanks by the extensions for 58 Java codes, and the solution performance of students when we changed the two parameter values in the extension. In future works, we will adopt the better maximal clique algorithm to further increase the number of blanks, generate a large set of fill-in-blank problems using the algorithm, and apply them in Java programming courses.

REFERENCES

[1] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG Int. J. Computer Science, vol. 40, no. 1, pp. 38-46, Feb. 2013.
[2] Tana, N. Funabiki, and N. Ishihara, "A proposal of graph-based blank element selection algorithm for Java programming learning with fill-in-blank problem," Proc. IMECS2015, pp. 448-453, March 2015.
[3] N. Funabiki, S. Sasaki, Tana, and W.-C. Kao, "An operator fill-in-blank problem for algorithm understanding in Java programming learning assistant system," Proc. GCCE 2015, pp. 346-347, Oct. 2015.
[4] 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," Inf. Eng. Express, vol. 1, no. 3, pp. 9-18, Sep. 2015.
[5] N. Ishihara, N. Funabiki, and W.-C. Kao, "A proposal of statement fill-in-blank problem using program dependence graph in Java programming learning assistant system," Inf. Eng. Express, vol. 1, no. 3, pp. 19-28, Sep. 2015.
[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, http://jflex.de/.
[8] jay, http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html.
[9] M. Takahashi, Easy Java, Softbank Creative, 2013.
[10] Y. Kondo, Algorithm and data structure for Java programmers, Softbank Creative, 2011.
[11] ITSenka, http://www.itsenka.com/.
[12] tutorialspoint, http://www.tutorialspoint.com/java/index.htm.
[13] Java program samples, http://www7a.biglobe.ne.jp/~java-master/samples/.
[14] Shellsort, http://www.thelearningpoint.net/computer-science/arrays-and-sorting-shell-sort-with-c-program-source-code.
[15] L. Sinapova, Lecture Notes, http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/Contents.htm.
[16] S. K. Chang, Data structures and algorithms, World Scientific Pub., USA, Oct. 2003.
[17] Dijkstra Algorithm, http://www.ifp.illinois.edu/~angelia/ge330fall09_dijkstra_l18.pdf.
[18] Prim Java, http://cs.fit.edu/~ryan/java/programs/graph/Prim-java.html.
[19] Graph Java, http://www.sanfoundry.com/java-program.
[20] Depth First Search, https://en.wikipedia.org/wiki/Breadth-first_search.
[21] Breadth First Search, https://en.wikipedia.org/wiki/Depth-first_search.
[22] M. Kaminaga, M. Yamada, and T. Watanabe, Cryptography using Java, http://www.morikita.co.jp/books/book/2214.