A Proposal of Graph-based Blank Element Selection Algorithm for Java Programming Learning with Fill-in-Blank Problems

Tana and Nobuo Funabiki

***Abstract*—To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS provides *fill-in-blank problems* for novice students to learn Java by filling blank elements composed of reserved words, identifiers, and control symbols. In this paper, we propose a graph-based *blank element selection algorithm* to select as many blanks as possible such that any blank has the grammatically correct unique answer. Our algorithm first generates a graph by selecting every candidate element in the code as a vertex, and connecting any pair of vertices by an edge if they can be blanked together, where the conditions for simultaneous blanks are defined. Then, it extracts a maximal set of blank elements by seeking a maximal clique of the graph. We verify the algorithm through applications to 100 Java codes, where the answer uniqueness is manually confirmed and the number of blank elements is almost proportional to the number of statements in a code.**

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

I. Introduction

As a reliable and portable programming language, *Java* has been extensively used in industries even at mission critical systems in large enterprises and small-sized embedded systems. Thus, the cultivation of Java programming engineers has been strongly demanded from industries. A lot of universities and professional schools are actually offering Java programming courses to deal with the demands. A Java programming course usually combines grammar instructions by classroom lectures and programming exercises by computer operations.

To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)* [1][2]. JPLAS provides the *fill-in-blank problem* to support self-studies of students. The *fill-in-blank problem* intends for a student to learn the Java grammar and basic programming. This problem shows a Java code with several blank elements to a student, where he/she needs to fill the blanks by typing the correct ones. 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* is a fixed sequence of characters that has been defined in Java grammar to represent the specified function, and should be mastered first by any student. An *identifier* is a sequence of characters defined in the code by the author to represent a variable, a class, and a method. A *control symbol* in this paper intends other elements such as "." (dot), ":" (colon), ";" (semicolon) , "(,)" (bracket), "{, }" (curly bracket).

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 the blank in the code is used as the unique correct answer, because it is not only simple but also helps the code reading by a student. Thus, the original element must be the unique grammatically correct answer for any blank to avoid confusions by a novice student. However, the selection of such blank elements is not easy especially when a teacher needs many blank elements. For example, if all the elements representing an identifier for a variable are blanked, a student cannot answer it at all. At least one element must be remained in this case.

In this paper, we propose a graph-based *blank element selection algorithm* to assist generating fill-in-blank problems in JPLAS. First, our algorithm generates a *compatibility graph* by selecting every candidate element 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 can be blanked simultaneously. Then, it seeks a maximal set of blank elements by extracting a *maximal clique* of the compatibility graph [3]. We evaluate the effectiveness of this algorithm through applying 100 Java codes, where the answer uniqueness is manually confirmed and the number of blank elements is almost proportional to the number of statements in the code.

The rest of this paper is organized as follows: Section II reviews the functions for fill-in-blank problems in JPLAS. Sections III and IV present the blank element selection algorithm. Section V shows evaluation results. Section VI introduces some related works. Section VII provides the conclusion with future works of this paper.

II. Review of Fill-in-blank Problem Functions in JPLAS

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

A. Software Platform

In the JPLAS server, we adopt the *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 to JPLAS through a Web browser.

In implementations of fill-in-blank problem functions, we adopt *JFlex* [4] and *jay* [5]. *JFlex* is a lexical analyzer generator that transforms a Java code into a sequence of

The authors 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.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

lexical units that represent the least meaningful elements. It classifies an element into either of a reserved word, an identifier, a symbol, and an immediate data. For example, a statement `int value = 123 + 456;` is divided into `int`,`value`, `=`, `123`, `+`, `456`, and `;`. *JFlex* cannot identify an identifier as a class, a method, or a variable. Thus, *jay* is used together, because it is a syntactic parsing program based on the LALR method, and can identify an identifier that is a variable/

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

Here, we define some terms for the fill-in-blank problem. A *problem code* represents a Java source code that is used for a fill-in-blank problem. A *question* represents a blank to be filled inside the problem code. A *problem* consists of one problem code with several questions or blanks, their correct answers, and a comment on this problem. An *assignment* consists of a title, one or multiple problems, and a comment on the assignment. Usually, 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. Teacher Functions

The functions for fill-in-blank problems in JPLAS consist of *teacher functions* and *student functions*. In this subsection, we review teacher functions.

1) Code Registration: First, a teacher selects Java source codes suitable for fill-in-blank problems and uploads them in the database of JPLAS. These codes should contain the elements to be learned by students at the corresponding classes and be worth for code reading.

2) Code and Element Type Selection: Then, the teacher selects one problem code and the types of elements to be blanked among reserved words, identifiers, and control symbols using the interface in Figure 1.

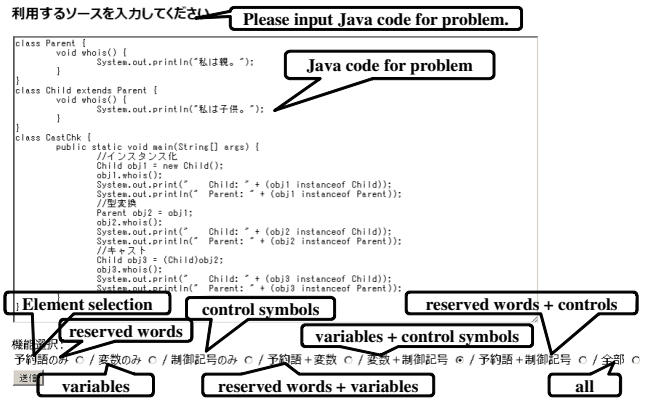


Fig. 1. Code and element type selection.

3) Blank Element Selection: Then, the teacher manually selects the elements to be blanked using the interface in Figure 2.

4) Problem Preview: The teacher can check the preview of the generated problem using the interface in Figure 3. Here, he/she can add a comment on the problem. Then, the problem is stored in the database.

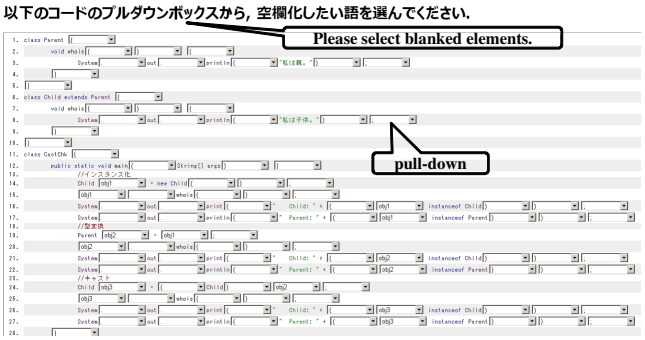


Fig. 2. Blank element selection.

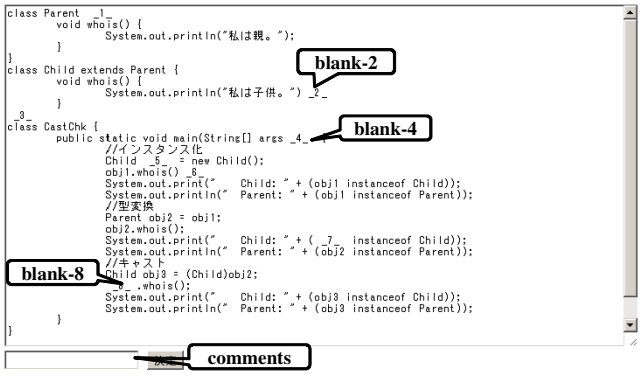
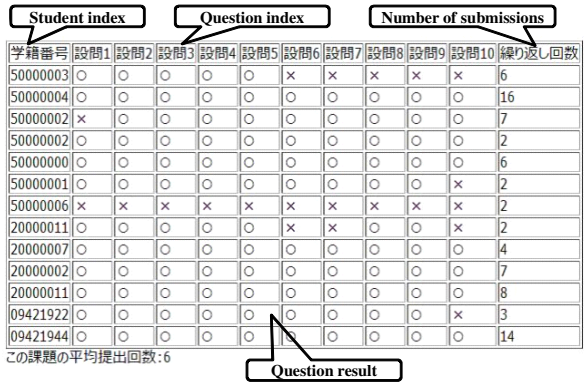


Fig. 3. Problem preview.

5) Assignment Registration: The teacher can register a new assignment for the course by selecting problems in the database. Here, he/she needs to describe the title and the comment for this assignment. Then, the assignment is stored in the database.

6) Score Reference: A teacher can check the answers from the students for the assignments in the course to evaluate their learning situations. For quick evaluations, he/she can overview the number of solving students and the average score among the students for each assignment in the course. For detailed evaluations, he/she can further look at the correctness of the questions and the number of answer submissions for each assignment by every student using the interface in Figure 4.



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

Fig. 4. Assignment answer results by students.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

7) Database Management: The JPLAS database keeps the information of the user names and IDs including teachers and students, the course titles, the problem codes, the problems, and the assignments. When a new teacher starts using JPLAS, the system manager should first register him/her in the database. Then, this teacher can register new courses, and generate new problems and assignments.

When a teacher uses 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 assignments for this course and view their student scores. Also, any student in the student list can access to the assignments and view his/her scores.

Any problem generated by a teacher is managed in the database and can be shared among the teachers. Thus, after this database has become enriched with a variety of problems and assignments, the load for generating assignments can be drastically reduced. It becomes only selecting existing problems or assignments in the database.

D. Student Functions

In this subsection, we review *student functions*.

1) Assignment Selection: First, a student accesses to the JPLAS server using a Web browser. He/she can view the list of the courses where he/she is registered and select one course. Then, he/she can view the list of the assignments in this course to be solved, where for each assignment, the assignment index, the title, the answer submission status, and the answer button are shown. Then, he/she can select one assignment to answer the problems.

2) Assignment Answering: Then, the student can view the direction, the assignment index, the comment, the problem code with questions, and the answer forms for the questions in the assignment as shown in Figure 5. He/she can input the answers into the corresponding forms. Here, we adopt an open-source editor called *CodePress* to improve the readability of the problem code by using the highlighting function in this editor [6].

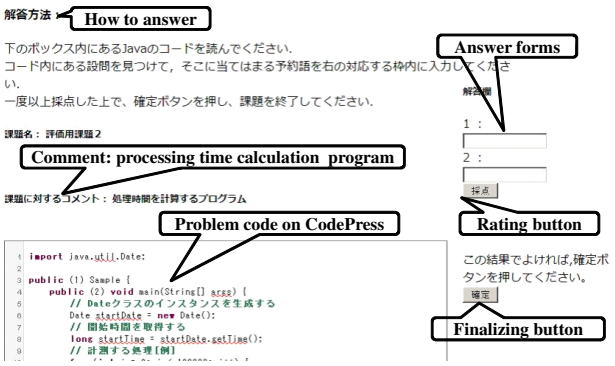


Fig. 5. Assignment answering.

3) Automatic Rating: Then, the student clicks either the ”rating” button or the ”finalizing” button after filling his/her answers to the questions in the forms. When the former button is cricked, the JPLAS server compares the answer with the correct one for each question. It returns ”OK” if they are matched, and ”NG” otherwise. When the latter button is cricked, the answering by the student is finalized, and his/her final answers, the number of submissions by cricking the rating button, and the date/time are stored in the database. A student needs to crick the ”finalizing” button when he/she solves every question correctly or gives up solving some of them. Here, the number of submissions in the database can be used to estimate the difficulty of the assignment and the ability of the student.

4) Score Reference: A student can check his/her score for each assignment using the interface in Figure 6.

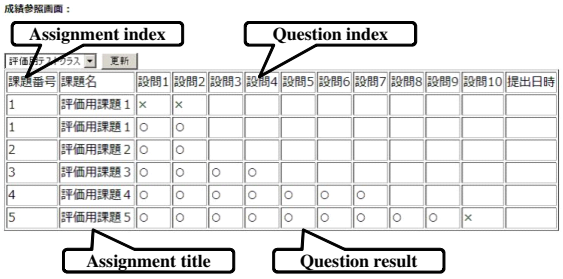


Fig. 6. Score reference table.

E. Score Ranking Graph

A student can also see his/her ranking among the students taking the same course, in terms of the total number of correct answers for all the assignments using the *score ranking graph*. It is expected that any student uses this graph for encouraging his/her study by evaluating the position in solving assignments among the students.

III. Four Categories for Blank Element Selection Algorithm

In this section, we present the four categories to represent the constraints in selecting blank elements with unique answers. These categories are used in the blank element selection algorithm in the next section.

A. Group Selection Category

In the *group selection category*, all the elements related with each other in the problem code are grouped together so that at least one element in each group is not selected for the blank. There are five conditions for this category in this paper.

```
1: class Sample1{
2:   public static void main(String args[]){
3:     int var1 = 10;
4:     float var2 = sampleMethod(var1);
5:     System.out.println("indata="+var1+"
      outdata=" +var2);
6:   }
7:   static float sampleMethod(int p1){
8:     float var3 = (float)(p1*1.08);
9:     return var3;
10:  }
11: }
```

(1) Identifier appearing two or more times in code

The multiple elements representing the same identifier with the same scope in the problem code are grouped together. A *scope* represents the range in the code where a variable, a class, or a method is referred using the same

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

name or identifier [7]. 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 are grouped together.

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

The three or more elements representing the pairing reserved words are grouped together. If all of them are blanked, the unique correct answers may become too hard or impossible. Besides, one element of them can be a good hint to derive the other elements for novice students. They include the following two cases:

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

(3) Data type for variables in equation

The elements representing the data types of the 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 cast data type `int` is also included in the group. Besides, if a method is included in an equation, like line 5 in `Sample1`, the data type for this method is also grouped together. Here, `float` at lines 4 and 7 are grouped.

(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 are grouped together through line 4.

The data type in (3)-(5) must be the same if some elements in these groups are 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 problem code in pairs are grouped together so that at least one element in each pair is not selected for the blank element. There are four conditions for this category in this paper.

(1) Continuously appearing elements in statement

The two elements appearing continuously in the same statement in the problem code are paired. If both of them are blanked, their unique correct answers may not be guaranteed, or may become too difficult for novice students. Due to the same reason, the two elements that are connected with a dot (".") are also paired. For example, in `Sample1`, `int` and `var1` at line 3 are paired, and `System` and `out` at line 5 are paired.

(2) Variables in equation

The elements representing any pair of the variables in an equation are paired. If both are blanked, the unique correct answers become impossible because the reversed order of them 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 combination of them is paired here.

(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, or too difficult for novice students. For them, one element of them can be a good hint to derive another one. They include 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. It happens that even if both are blanked at the same time, the code can be grammatically correct. Besides, novice students should carefully check them in their codes first. For example, in `Sample1`, { at lines 1 and } at line 11 are paired.

C. Prohibition Category

In this category, an element is prohibited from the selection for the blank element because it does not satisfy the uniqueness with the high probability. There are three conditions for this category in this paper.

(1) Identifier appearing only once in code

The element representing the identifier appearing only once in the problem code is selected into this category. If it is blanked, a student cannot answer the original identifier.

(2) Operator

The element representing the operator such as the arithmetic operator: `+`, `-`, `*`, `/`, the comparative operator: `<`, `>`, `<=`, `>=`, `==`, `!=`, and the logical operator: `&`, `|`, `^`, `!` is selected into this category. If an operator is blanked, a student cannot answer the original one unless the proper explanation on the specification related to the operator is given. For example, in `Sample1`, `*` at line 8 is prohibited.

(3) Access modifier

The element representing the access modifier for an identifier is selected into this category. If it is blanked, either of `public`, `protected`, `private` can be often grammatically correct. For example, in `Sample1`, `public` at line 2 is prohibited.

D. Single Selection Category

The remaining elements in the problem code can be blanked alone.

IV. Blank Element Selection Algorithm

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

A. Algorithm Overview

In this algorithm, the *constraint graph* is first generated from the given problem code. In this graph, a *vertex* represents a candidate element for blank, and an *edge* does the constraint such that the incident elements cannot be

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

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 blanked elements with unique answers. This algorithm consists of the four steps that will be described in the following subsections.

B. Vertex Generation for Constraint Graph

In the constraint graph, each vertex represents a candidate element for blank. The candidate elements are extracted from the problem code through the lexical analysis using *JFlex* and *jay*. Each vertex contains the associated information in Table that 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 that are classified into the prohibition category in III-C are removed from the constraint graph.

C. Edge Generation for Constraint Graph

Then, for the constraint graph, an edge is generated between any pair of two vertices or elements that should not be blanked at the same time for the unique correct answers. 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 corresponding elements is randomly selected first, and then, an edge is generated between this selected vertex and each of the other vertices in the same group. Thus, at least this selected vertex is not selected for blank.

D. Example of Constraint Graph

Figure 7 illustrates a part of the constraint graph for `class 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 blanks with unique answers. A *clique* of a graph represents its subgraph such that any pair of two vertices in this subgraph is connected by an edge in the original graph. The procedure for our 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 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 number of selected vertices, remove this vertex from the compatibility graph and go to (5).
(4) Add the selected vertex for the blank, and remove it and its non-adjacent vertices from the compatibility graph.
(5) If the compatibility graph becomes null, terminate the procedure.
(6) Go to (2).

Here, (2) is introduced to sustain the number of blanked *control symbols*, because a problem code usually has a lot of control symbols.

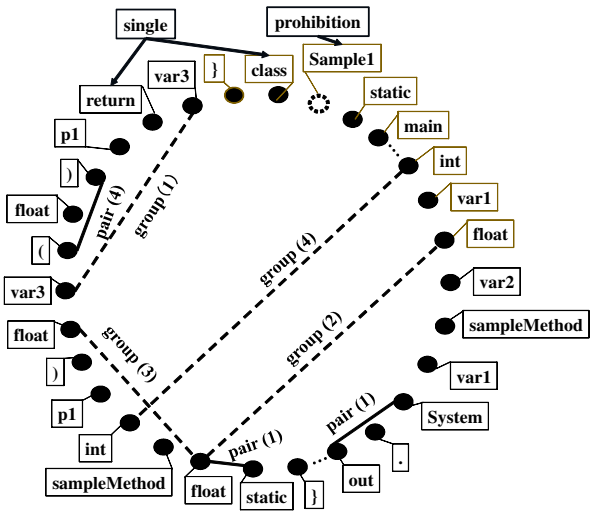


Fig. 7. Constraint graph for `Sample1`.

V. EVALUATION

In this section, we evaluate the proposed blank element selection algorithm for fill-in-blank problems.

A. Uniqueness of Correct Answer

Firstly, we verify the uniqueness of grammatically correct answers to 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, where the number of statements in each code is varied from 6 to 85, and 24 codes have multiple classes or methods. We generated fill-in-blank problems by applying the algorithm to these codes, and asked four students in our group to solve them. These students are currently using Java in their researches and are familiar to the Java programming.

The results show that for 97 codes among them, all the blanks selected by the algorithm have unique answers. For the remaining three codes, two variables can be swapped at two blanks. The following code `Sample2` shows one such example. Grammatically, `outData2` and `outData1` can

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

be filled at _4_ and _5_ respectively, although the reversed ones are correct. To resolve this problem, we additionally show the output result of this code as shown 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

Then, we examine the relationship between the number of statements in a problem code and the number of blank elements selected by the algorithm. Figure 8 shows them for the 100 Java codes. This graph indicates that they are almost proportional to each other except for the first 10 codes. As shown in Table II, these codes have a larger number of statements composed of only curly brackets due to multiple classes/methods and/or multiple branches on conditions. Because 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.

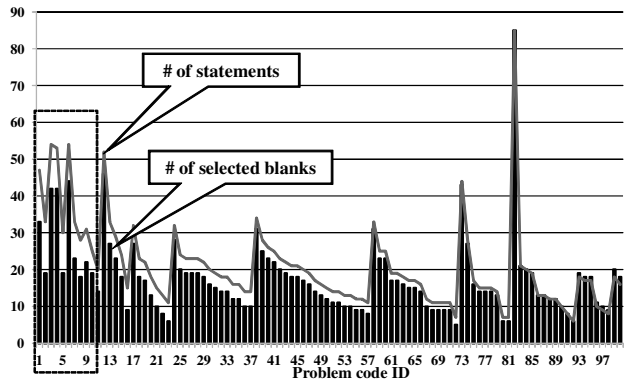


Fig. 8. Number of statements and number of blanks in 100 codes.

VI. Related Works

In [8], Kashihara et al. proposed a method of blanking an important point of data or control flow of a C code to make instructive fill-in-blank problems using *Program Dependence Graph (PDG)* without considerations of semantic aspects of the algorithm. PDG can represent the relationship of data dependency and control flows between commands using a graph. In future studies, we will consider the use of PDG to extract important elements in the code.

In [9], Shinkai et al. proposed 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.

TABLE II
Features of 10 codes with large 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

In [10], Taguchi et al. proposed a programming education assistant system to provide assignments fitting to individual students. The understanding 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 students using the database of them. Because our algorithm may need to generate fill-in-blank problems with various levels to deal with a variety of students, we will consider the use of this technique.

VII. Conclusion

In this paper, we proposed a graph-based *blank element selection algorithm* for fill-in-blank problems in *Java Programming Learning Assistant System (JPLAS)*. We verified the algorithm through applications to 100 Java codes, where the uniqueness of grammatically correct answers is confirmed. In future studies, we will further apply this algorithm to more variety of codes to justify the uniqueness, generate fill-in-blank problems using this algorithm, and assign them to students in Java programming courses in plural universities including our department to verify the effectiveness in Java programming educations.

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 Int. J. Computer Science, vol. 40, no.1, pp. 38-46, Feb. 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," Proc. Int. Symp. Front. Inform. Sys. Netw. Appl., pp. 375-380, March 2012.
[3] M. R. Garey and D. S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, Freeman, New York, 1979.
[4] JFlex, http://jflex.de/.
[5] jay, http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html.
[6] CodePress, http://sourceforge.net/projects/codepress/.
[7] Scope, http://java.about.com/od/s/g/Scope.htm.
[8] A. Kashihara, K. Kumei, K. Umeno, and J. Toyota, "How to make fill-in-blank program problems for learning algorithm," Proc. Int. Conf. Comput. in Education, pp. 776-783, 1999.
[9] 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 Tech. Report, ET, pp. 7-10, Oct. 2010.
[10] 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. 958-96, Feb. 2007.

ISBN: 978-988-19253-2-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)