Information Engineering Express
International Institute of Applied Informatics
2015, Vol.1, No.3, 9 – 18

A Proposal of Value Trace Problem for Algorithm Code Reading in Java Programming Learning Assistant System

Khin Khin Zaw [*] , Nobuo Funabiki[*] , Wen-Chung Kao [†]

Abstract

To assist Java programming educations, we have developed a Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS provides the *element fill-in-blank problem* to help novice students self-study Java programming by filling in blanked elements in a code. However, it is a pity that this problem can be solvable without reading the algorithm in the code, especially if students are familiar with grammar. In this paper, we propose the *value trace problem* to answer the changing values of important variables in a Java code that implements a fundamental data structure or algorithm, so as to improve the code reading capability. To verify the effectiveness, we generated five problems using Java codes for sorting and asked 10 students in our group to solve them.

Keywords: Java programming education, JPLAS, code reading, value trace problem, algorithm, fill-in-blank.

1 Introduction

The programming language *Java* has high reliability and portability with excellent learning environments provided. Java has been extensively used for various practical systems in industries, even at mission critical systems in large enterprises and small-sized embedded systems. Thus, the industries have a strong demand to cultivate more Java programmers. In fact, a lot of universities and professional schools are offering Java programming courses to deal with the demands. A Java programming course usually combines grammar instructions of classroom lectures and programming exercises performed in a computer operations.

To help Java programming educations, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)* [1]. JPLAS provides the *element fill-in-blank problem* to support self-studies of students. It intends for novice students to learn the Java grammar and basic programming. In this problem, a high-quality Java code with blanked elements is exhibited for students to fill in the blanks with correct elements. An *element* represents the least unit in a Java code such as a *reserved word*, an *identifier*, and a *control symbol* [2]. Any answer is marked by string matching with the correct element.

[*] Department of Electrical and Communication Engineering, Okayama University, Okayama, Japan
[†] Department of Electrical Engineering, National Taiwan Normal University, Taipei, Taiwan

In addition, we proposed the *graph-based blank element selection algorithm* to assist generating a feasible element fill-in-blank problem from a Java code, where the unique answer exists for any blank element. This algorithm first 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 algorithm, we defined the conditions that a pair of elements can be blanked simultaneously. Then, the blank elements are selected by extracting a maximal clique of a compatibility of a graph.

Unfortunately, this element fill-in-blank problem can be solved mechanically without reading out the algorithm in the Java code, especially when students are familiar with this problem and the Java grammar. Due to the unique answer constraint, limited choices of elements may exist for many blanks. Actually, we observed that with the increase in the number of solving element fill-in-blank problems, students could reach correct answers much faster than in the beginning. Thus, a new problem that keeps the nature of filling in blanks and marking answers by string matching but requires much deeper code reading is necessary for such students.

In this paper, we propose the value trace problem as a new type of an element fill-in-blank problem in JPLAS. In this problem, students are questioned about actual values of important variables in a Java code implementing a fundamental data structure or algorithm [3]. Basically, we present the *blank line selection algorithm* to blank the whole data in the line of the output data through executing the code where at least one data is changed from the previous one. To verify the effectiveness, we generated five sorting problems, and asked 10 students who have various Java programming skills in our group to solve them.

The rest of this paper is organized as follows: Section 2 presents the generation procedure for the value trace problem in JPLAS. Section 3 presents the blank line selection algorithm as the core part in the procedure. Section 4 shows evaluations of our proposal. Section 5 discusses related works. Section 6 provides the conclusion with future works.

2 Generation Procedure for Value Trace Problem

In this section, we present the generation procedure for the value trace problem in JPLAS.

2.1 Overview of Generation Procedure

The goal of the *value trace problem* in JPLAS for Java programming educations is to give students training opportunities for profound reading and analyzing a Java code that implements a fundamental data structure or algorithm by asking to trace real values of important variables in the code. The *code reading* plays an essential role in writing high-quality codes for any programmer. It is also indispensable in modifying existing codes for some systems, which is common in real worlds. A value trace problem is generated by a teacher with the following steps:

1. Select a high-quality class code for a fundamental data structure or an algorithm.

2. Create the main class to instantiate the class in 1) if it does not contain the main method.

3. Add the functions to write values of important variable in questions into a text file.

4. Prepare the input data file to be accessed by algorithm Java code and the teachers can modify the data in the input data file if necessary.

5. Run the algorithm Java code to obtain the set of variable values in the output text file.

6. Blank some values from the output text file to be filled in by students.

7. Upload the final Java code, the blanked text file, and the correct answer file into the JPLAS server, and add the brief description on the algorithm and the problem, to generate a new assignment.

2.2 Details of Procedure Using Insertion Sort

In this subsection, we describe the detail of each step in the value trace problem generation procedure using a Java code for *Insertion sort* [4][5]. *Insertion sort* always maintains the sorted data list at the lower positions of the input data list. A new data in the input data list is inserted into the sorted list such that the largest data is located at the last position of the expanded sorted list. Thus, the input data list after k iterations has the property where the first $k+1$ entries are sorted. In this paper, we adopt the following code for *Insertion sort*:

2.2.1 Selecting Java Code for Insertion Sort

In this paper, we select the following Java code for *Insertion Sort* in [6].

```
1: class InsertionSort{
2:     //input data is arr[]
3:      public static void insertionSort(int[] arr){
4:          int i, j;
5:          int tmp; //item to be inserted
6:         //start with 1 (not 0)
7:          for (i=1; i<arr.length; i++){
8:              tmp = arr[i];
9:             //smaller values are moving up
10:      for (j=i ; j>0 && arr[j-1]> tmp; j--){
11:                arr[j] = arr[j-1];
12:           }
13:             arr[j] = tmp;
14:             for(int k:arr){
15:                System.out.print(k);
16:                System.out.print(",");
17:             }
18:             System.out.print();
19:         }
20:     }
21: }
```

2.2.2 Creating Main Class

Some Java codes may not contain *main method* but simply *classes* similar to the code in Section 2.2.1. For reference, we call it *algorithm class*. Then, we need to create the main class and instantiate the algorithm class, read input data of the algorithm as well as write the output data [7].

2.2.3 Adding Output Functions

An algorithm is regarded as a well-defined computational procedure of a sequence in computational steps that transform the input values to the output values [8]. Thus, students can study and understand the procedure of the fundamental data structure or algorithm in a Java code by tracing the values of important variables during the transformation of the input values to the output values. It becomes necessary to add the functions of writing such variable values in a text file under *main class* and *algorithm class*. In *Insertion sort*, the values of the variables for sorted data are essential for understanding the algorithm, and should be traced at each iteration by students. Thus, writing these values of variables into a text file is added as functions to complete the *problem code* in generating a value trace problem.

2.2.4 Preparing Input Data file

An input data file should be prepared to be accessed by the *problem code*:

```
2,1,3,5,4,7,6,8,9,10
```

2.2.5 Obtaining Output Data file

After running the *problem code*, the complete output text file for the value trace problem is as follows:

```
1: 1,2,3,5,4,7,6,8,9,10
2: 1,2,3,5,4,7,6,8,9,10
3: 1,2,3,5,4,7,6,8,9,10
4: 1,2,3,4,5,7,6,8,9,10
5: 1,2,3,4,5,7,6,8,9,10
6: 1,2,3,4,5,6,7,8,9,10
7: 1,2,3,4,5,6,7,8,9,10
8: 1,2,3,4,5,6,7,8,9,10
9: 1,2,3,4,5,6,7,8,9,10
```

2.2.6 Blanking Values for Problem Generation

For students to trace the data values, we blank the whole line in the output text file called *change line* where some data is changed from the previous line. Here, the *blank line selection algorithm* is used to select *change lines* to be blanked properly. The detail of this algorithm will be discussed in Section 3. By choosing $blankRate = 50$ for this algorithm, we can obtain the following result:

```
1:___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
2: 1  ,  2  ,  3  ,  5  ,  4  ,  7  ,  6  ,  8  ,  9 ,  10
3:___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
4:___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
5: 1  ,  2  ,  3  ,  4  ,  5  ,  7  ,  6  ,  8  ,  9 ,  10
6:___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
7: 1  ,  2  ,  3  ,  4  ,  5  ,  6  ,  7  ,  8  ,  9 ,  10
8:___ , ___ , ___ , ___ , ___ , ___ , ___ , ___ , ___, ___
9: 1  ,  2  ,  3  ,  4  ,  5  ,  6  ,  7  ,  8  ,  9 ,  10
```

2.2.7 Generating Assignment

After preparing the *problem code*, the blanked text file, and the correct answer file, we upload them to the JPLAS server using the existing function. Then, brief descriptions of this problem is added to help students better understand it.

3 Blank Line Selection Algorithm

In this section, we present the blank line selection algorithm for Section 2.2.6 .

3.1 Idea

In this algorithm, we blank the whole data in one line in the output text file from the *problem code*. To create a more difficult problem, we preferentially select the line where at least one data is changed from the previous line, which is called *change line*. Nevertheless, the number of lines to be blanked or *target lines*, can be specified by the teacher. If the number of *change lines* is smaller than the number of *target lines*, select all the *change lines* and randomly select the remaining number from non *change lines*. If the number of *change lines* is larger, randomly select *change lines* by this number.

3.2 Procedure

The procedure for the *blank line selection algorithm* is described as follows:

1. Calculate the number of *target lines* to be blanked (*targetLine*) from the algorithm input parameter (*blankRate*) and the total number of lines in the output text file (*totalLine*) by $targetLine = blankRate/100 * totalLine$.

2. Count the number of changed data in each line from the previous one in the output text file.

3. Count the number of *change lines* in the output text file (*changeLine*) such that the number in **2** is not zero.

4. If $changeLine = targetLine$, then select all of the *change lines* for blanks.

5. If $changeLine < targetLine$, then select $(targetLine - changeLine)$ non *change lines* to be blanked by repeating the following procedure:

1) Calculate the selection rate (*selectRate*) by $selectRate = (targetLine - changeLine)/(totalLine - changeLine)$.
2) Initialize the number of the selected blank lines (*selectLine*) by *changeLine*.
3) Repeat the following steps:
   (1) Visit the first line in the output text file.
   (2) If this line has been selected to be blanked, go to (4).
   (3) If $random < selectRate$, then select this line to be blanked, and count up by $selectLine++$, where *random* returns a 0-1 random real number.
   (4) If $selectLine = targetLine$, then terminate the procedure.
   (5) If the current line is not the last line in the output text file, then visit the next line and go to (2).
   (6) Go to (1).

6. If $changeLine > targetLine$, then select $(targetLine)$ *change lines* to be blanked by repeating the following procedure:

1) Calculate the selection rate (*selectRate*) by *selectRate=targetLine/changeLine*.
2) Initialize the number of the selected blank lines (*selectLine*) by 0.
3) Repeat the following steps:
   (1) Visit the first line in the output text file.
   (2) If this line has been selected to be blanked, go to (4).
   (3) If *random*<*selectRate*, then select this line to be blanked, and count up by *selectLine++*.
   (4) If *selectLine=targetLine*, then terminate the procedure.
   (5) If the current line is not the last line in the output text file, then visit the next line and go to (2).
   (6) Go to (1).

3.3 Example Problem for Insertion Sort

In the example of Section 2.2.6, this algorithm calculates $targetLine = 5$ (=$50/100 * 9$). It first selects the three *change lines*, then randomly selects the two non *change lines*.

4 Evaluation

In this section, we evaluate our proposal by generating five value trace problems using Java codes for sorting algorithms and apply them to students.

4.1 Applications of Five Value Trace Problems

First, we generated five value trace problems by using the Java codes for *Shell sort* [9], *Quick sort* [10], *Bubble sort*, *Insertion sort*, and *Selection sort*. These algorithms are common and are usually taught in universities. Table 1 shows problem outlines.

Table 1: Five value trace problems for evaluations.

ID	algorithm	LOC	# of blanks
P1	Shell sort	39	17
P2	Quick sort	47	43
P3	Bubble sort	32	10
P4	Insertion sort	29	23
P5	Selection sort	29	24

Then, we asked 10 students in our group who have different skills and knowledge in Java programming to solve them in JPLAS. After that, we requested them to answer the five questions in Table 2 of the questionnaire. For Q1, students should reply with five levels, where 1 is the easiest and 5 is the most difficult. For Q2, they should reply with four levels, where 1 is less than 10 min., 2 is about 15 min., 3 is about 20 min., and 4 is longer than 25 min. Then, for Q3-Q5, students should reply with *yes* or *no* for all five problems.

Table 3 shows the results for the individual problems. Here, the results show the number of students who solved each problem correctly and the average number of their answer

Table 2: Questions in questionnaire.

ID	question
Q1	How difficult is each problem ?
Q2	How long did you spend to answer each problem ?
Q3	Do you understand the algorithm in the code by solving the problems ?
Q4	Do you think the value trace problem is useful for Java code reading ?
Q5	Can you implement the algorithm in Java code by solving the problems ?

submissions where JPLAS can record the submission numbers. This table indicates that among the five value trace problems, the problem for *Quicksort* is the most difficult since two students were not able to solve it and the average number of submissions as well as the average difficulty/spending time levels are the highest. The reason will be analyzed in Section 4.2.

Table 3: Solution and questionnaire results.

ID	# of solving students	ave. # of submissions	ave. level for Q1	ave. level for Q2
P1	10	4.7	2	2.4
P2	8	12.8	3.2	3.5
P3	10	1.8	1.5	1.8
P4	9	3.2	1.3	1.6
P5	10	2.5	1.3	1.5

Table 4 shows the results for Q3-Q5. From Q3 and Q4, nine students among 10 replied that the value trace problem in JPLAS is effective in understanding the algorithm in the Java code and the code reading. However, for Q5, only seven students replied that they have confidence in writing a code for the algorithm even after solving them. From these results, we conclude that the value trace problem is useful and effective for Java code reading, but may not be sufficient for Java code implementations of algorithms.

Table 4: Questionnaire results on effectiveness of value trace problem.

	Q3	Q4	Q5
yes	9	9	7
no	1	1	3

4.2 Difficulty Analysis of Quick Sort

In the previous subsection, the value trace problem for *Quick Sort* is the most difficult. Our analysis on the reason is that *Quick sort* employs the divide-and-conquer strategy. It starts by picking an element from the data list as the *pivot*. Then, it reorders the data list so that

all the elements with values less than the *pivot* come before the *pivot* and the other elements come after it, called *partitioning*. Then, it recursively applies the same procedure to each sub-list at the left side and the right side of the pivot, until the whole list is sorted [10].

In the following problem for *Quick Sort*, the codes from line 1 to line 38 describe the *algorithm class* with added output functions to a text file, the codes from line 39 to line 44 describe *main class*, while the codes from line 46 to line 60 depicts the problems yet to be solved by students. 43 blanks are prepared for students to fill in the correct values. The pivot p is the most important parameter. For each p, the data arrangement is applied for each data set. Thus, to understand the code, students should trace the values of p from the first one to the last and the data arrangement results for each p.

```
1: class Quicksort{
2:      public static int partition(int array[], int left, int right){
3:          int p,tmp,i,j;
4:          p=array[left];
5:          i=left;
6:          j=right+1;
7:          System.out.println("pivot: "+p);
8:          for(;;){
9:              while (array[++i]<p) if (i>=right) break;
10:             while (array[--j]>p) if (j<=left) break;
11:             if (i>=j) break;
12:             tmp=array[i];
13:             array[i]=array[j];
14:             array[j]=tmp;
15:         }
16:         if (j!=left){
17:             tmp=array[left];
18:             array[left]=array[j];
19:             array[j]=tmp;
20:         }
21:         System.out.print("output: ");
22:         for (int k:array){
23:             System.out.print(k);
24:             System.out.print(" ");
25:         }
26:         System.out.println();
27:         return j;
28:     }
29:     public static void quicksort(int a[], int left, int right){
30:         int i;
31:         if (right>left){
32:         i=partition(a, left, right);
33:         System.out.println();
34:         quicksort(a, left, i-1);
35:         quicksort(a, i+1, right);
36:     }
37:   }
38: }
39: public class Quickmain{
40:   public static void main (String[] args){
41:      int [] arr={65,70,75,80,85,60,55,50,45};
42:      QuickSort.quicksort(arr,0,arr.length-1);
43:   }
44: }
45: <Problem>
46: pivot: _1_
```

```
47: output: _2_,_3_ ,_4_ ,_5_,_6_, _7_ ,_8_ ,_9_ ,_10_
48: pivot: _11_
49: output: _12_,_13_,_14_,_15_,_16_,_17_,_18_,_19_,_20_
50: pivot: _21_
51: output: 50 , 45 , 55 , 60 , 65 , 85 , 80 , 75 , 70
52: pivot: _22_
53: output: _23_,_24_,_25_,_26_,_27_,_28_,_29_,_30_,_31_
54: pivot: _32_
55: output: _33_,_34_,_35_,_36_,_37_,_38_,_39_,_40_,_41_
56: pivot: _42_
57: output: 45 , 50 , 55 , 60 , 65 , 70 , 80 , 75 , 85
58: pivot: _43_
59: output: 45 , 50 , 55 , 60 , 65 , 70 , 75 , 80 , 85
```

5 Related Works

In this section, we briefly introduce some related works of the value trace problem. In our survey, no work has been reported for the same problem.

In [12], Smulders presented the *Annotate Code* project for explaining algorithms in introductory programming courses to students that have not yet developed a mental image of them. It allows teachers to create visualizations based on code stepping. As Web applications, users can submit codes and steps through a browser. Each step can be accompanied by a user-generated drawing to creating a step-by-step animation like a debugger.

In [13], Quinson et al. presented the *Programmer's Learning Machine (PLM)* as an interactive exerciser aimed at learning programming and algorithms. It targets students in (semi-)autonomous settings, using an integrated and graphical environment that provides a short feedback loop. This generic platform also enables teachers to create specific programming microworlds that match their teaching goals. PLM provides two main panels to provide information for students to solve exercises.

In [14], Sykes et al. presented the Web-based *Java Intelligent Tutoring System (JITS)* for students in first programming courses. By bringing together recent developments in intelligent tutoring systems, cognitive science, and AI, it constructs an intelligent tutor to help students learn Java programing.

In [15], Osman et al. introduced a visualized learning system to enhance the education of data structure course. It has the capability to display data structure graphically as well as allow its graphical manipulation for students to observe the execution result and track the algorithm execution.

6 Conclusion

In this paper, we propose the *value trace problem* for algorithm code reading in the *Java Programming Learning Assistant System (JPLAS)*. For evaluations, we generate five value trace problems by using Java codes for different sorting algorithms, and ask 10 students to solve them in JPLAS. Then, we analyze their solution and questionnaire results, and the difficulty of the problem for *Quick Sort*. In future studies, we will generate value trace problems under a variety of data structures or algorithms, and apply them to Java programming courses for its effectiveness in Java programming educations.

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, Feb. 2013, pp. 38-46.

[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. Int. MultiConf. Eng. Comput. Sci., March 2015, pp. 448-453.

[3] Data Structures Tutorials, http://cs-fundamentals.com/data-structures/data-structures-tutorials.php.

[4] Insertion, http://interactivepython.org/courselib/static/pythonds/SortSearch/TheInsertionSort.html.

[5] InsertionSort, http://mycodinglab.com/insertion-sort-algorithm/.

[6] Java code, http://www.journaldev.com/585/insertion-sort-in-java-algorithm-and-code-with-example.

[7] JavaMain, http://csis.pace.edu/~bergin/KarelJava2ed/ch2/javamain.html.

[8] Algorithm, http://http://pepole.cis.ksu.edu/~tamotoft/CIS775/F08/Slides/01.pdf.

[9] ShellSort, http://www.thelearningpoint.net/computer-science/arrays-and-sorting-shell-sort-with-c-program-source-code.

[10] QuickSort, http://www.algolist.net/Algorithms/Sorting/Quicksort.

[11] K. K. Zaw and N. Funabiki, "A concept of value trace problem for Java code reading education," Proc. Int. Cong. Adv. Appl. Inform., July 2015, pp. 253-258.

[12] B. Smulders, "Annotate Code, introducing a system for code-stepping based visualization," Master Thesis, Leiden Univ., August 2014.

[13] M. Quinson and G. Oster, "A teaching system to learn programming: the programmer's learning machine," Proc. ITiCSE '15, July 2015.

[14] E. R. Sykes and F. Franek, "An intelligent tutoring system prototype for learning to program Java," Proc. Int. Conf. Adv. Learn. Tech., 2003.

[15] W. I. Osman and M. M. Elmusharaf, "Effectiveness of combining algorithm and program animation: a case study with data structures courses," Issue. Inform. Sci. Inform. Tech., vol. 11, 2014, pp. 155-168.