Value Trace Problems for Graph Theory Algorithms in Java Programming Learning Assistant System

Nobuo Funabiki, Khin Khin Zaw, Minoru Kuribayashi, and Wen-Chung Kao

***Abstract*—Code reading is very important in programming educations for students. Through reading and analyzing high quality codes, they can study how to write a proper code and modify it with given specification. To assist with the studies of Java code reading, we proposed the *value trace problem* in Web-based *Java Programming Learning Assistant System (JPLAS)*. JPLAS has been developed to provide self-learning environments to students by our group. This value trace problem asks students to trace the actual values of important variables in a Java code implementing a fundamental data structure or an algorithm. In this paper, we study value trace problems for *graph theory* algorithms. First, using the *Dijkstra* algorithm, we analyze the requirements and points in a value trace problem for this representative graph theory algorithm. Then, we generate problems for the two graph theory algorithms to examine their problem size and the effectiveness in Java programming studies. Our evaluation results show that value trace problems for *graph theory* algorithms are viable learning tools for algorithm understanding and code reading whereas additional tools are necessary for code writing.**

***Index Terms*—Java programming, JPLAS, code reading, value trace problem, graph theory algorithm.**

I. Introduction

The programming language *Java* is selected as the most popular programming language [1]. Java produces high reliability and portability under fine learning environments, and has been extensively used for various practical systems amongst industries. As a result, Java programming engineers are in high demand from across industries. In fact, a lot of universities and professional schools offer Java programming courses to deal with these demands. A Java programming course usually combines grammar instructions of classroom lectures and programming exercises via computer operations.

To assist teachers and students in doing Java programming exercises, we have developed the Web-based *Java Programming Learning Assistant System (JPLAS)* [2], [3]. In JPLAS, we proposed the *value trace problem* as a new type of the element fill-in-blank problem to cultivate *code reading* capabilities for novice students learning Java programming [4], [5]. This problem allows students to trace the actual values of important variables in a Java code to form a fundamental data structure or an algorithm. The goal of this value trace problem is to offer *code reading* opportunities for students, so that they can analyze and understand the structure as well as behaviors of the code for an algorithm. In addition, it is also expected that students can write and modify the code freely, in order to comply with the required specifications of an assignment or a project using this algorithm.

A value trace problem can be generated by the following seven steps: 1) select a high-quality class code for an algorithm, 2) make the main class instantiating the class in 1) if it does not contain the main method, 3) add functions to output the changed values of important variables into a text file while executing the code, 4) prepare the input data for this algorithm code, 5) run the code to obtain the sequence of variables in the text file, 6) blank some values in the text file to be filled by students, and 7) upload the final Java code, the blanked text file, and the correct answer file into the JPLAS server, then add the brief description to the algorithm for a new assignment. In our previous study [6], we proposed the *blank line selection algorithm* to help blank values in 6). This algorithm basically blanks the whole data in a line of output data during the execution, so that at least one data in the line will be changed (different) from the previous line. For evaluations, we used this algorithm and asked students to solve value trace problems for *Stack*, *Queue*, *Insertion sort*, *Selection sort*, *Bubble sort*, *Shell sort*, and *Quick sort*.

In the *graph theory*, several important algorithms should be regarded as fundamental algorithms in computer science for students. They include the algorithms of *Dijkstra*, *Prim*, *Breadth First Search (BFS)*, *Depth First Search (DFS)*, and *Maximum Flow*. These algorithms have been used in many important practical applications in computer systems, information systems, and network systems [7].

In this paper, we study value trace problems for graph theory algorithms. First, using *Dijkstra* algorithm as a representative graph theory algorithm, we analyzed requirements in value trace problems for graph theory algorithms. Then, we generated problems for two algorithm and examined the size of each problem. Next, we asked 25 students from our department to solve the problems for evaluations of solution performances and effectiveness in learning Java programming.

Many related studies have shown that animation tools can improve teachings of fundamental data structure and algorithms. In [8], Stallmann et al. presented a graph algorithm animation tool called "Galant". It simplifies the work of the animator who designs an animation so that students can create their own by adding visualization directives to pseudocode algorithms. Animation examples of *Dijkstra*, *Kruskal*, Depth-first search, and Insertion sort were also given. In [9], Scott et al. proposed the direct manipulation (DM) language for explaining algorithms by

Manuscript received December 1, 2015; revised March 11, 2016.

N. Funabiki, K. K. Zaw, and M. Kuribayashi 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: jungkao68@gmail.com).

manipulating visualized data structures. In [10], Osman et al. introduces a visualized learning environment that combines the algorithm animation and the program animation, as well as to assess its effectiveness to enhance the computer science education and data structure course of information technology.

The paper is organized as follows: Section II overviews *JPLAS* as the platform for value trace problems. Section III reviews *Dijkstra* algorithm. Section IV presents the generated value trace problem for *Dijkstra* algorithm. Section V shows evaluations of value trace problems for two graph theory algorithms. Section VI concludes this paper with future studies.

II. Overview of JPLAS

In this section, the *JPLAS* is viewed as the platform of providing value trace problems for students using a Web browser.

A. Software Platform of JPLAS

JPLAS is a typical Web application system. Fig. 1 illustrates the 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.

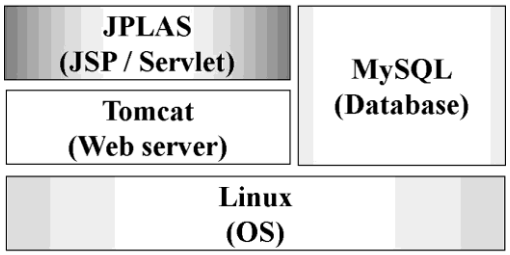

Fig. 1. Software platform of JPLAS.

B. Two Problems Types in JPLAS

JPLAS mainly provides two types of problems, namely the *code writing problem* [2] and the *fill-in-blank problem* [3] to support self-studies of students at various learning levels. JPLAS can assist teachers in reducing their workloads of evaluating codes and writing comments, while boosting students' motivations with quick feedbacks at the same time. This system is expected to improve Java programming education environments in any institute around the world.

1) *Code Writing Problem:* The *code writing problem* intends to help students to learn how to write Java source codes from scratch. The implementation of this JPLAS function is based on the *test-driven development (TDD) method* [11], which uses an open source framework *JUnit* [12]. *Junit* automatically tests the answer code in the server to verify its correctness when submitted by a student. Thus, a student can easily repeat the learning cycle of writing, testing, modifying, and resubmitting a code by him/herself. However, the drawback of this function is that any student needs to write a code so as to be tested with the test code prepared by the teacher on *JUnit*. Thus, this function may be considered to be too advanced for novice students learning Java programming.
2) *Fill-in-blank Problem:* The *fill-in-blank problem* allows 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, called a *problem code*, is shown to a 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.

A *reserved word* is a fixed sequence of characters that has been defined in Java grammar to represent a specified function, and should be mastered first by the students. An *identifier* is defined by the author as a sequence of character in the code to represent a variable, a class, or a method. A *control symbol* in this paper includes other grammar elements such as "." (dot), ":" (colon), ";" (semicolon), "(,)" (bracket), "{, }" (curly bracket). The correctness of each answer from a student is checked through string matching with the corresponding correct answer in the server. The value trace problem in this paper has been implemented in JPLAS as a variant of the fill-in- blank problem that uses the same functions in user interface and the answer checking.

III. Dijkstra Algorithm

In this section, we review *Dijkstra* algorithm that finds the shortest path from a given node to the other nodes in a given weighted graph [13].

A. Algorithm Overview

Dijkstra algorithm computes a solution for the single source shortest path problem in a weighted graph $G = (V, E)$ where each edge in E has a non-negative weight [14].

1) It starts by assigning initial values for the distances from the starting node *s* to the other nodes in *G*.
2) It operates in steps where the shortest distance from node *s* to another node is improved.

B. Pseudo Code for Dijkstra Algorithm

The pseudo code for *Dijkstra* algorithm is described as follows:

```
1: Procedure single-source-Dijkstra(G, w, s)
2: begin
3:    for each vertex u in V
4:    begin
5:      dist[v] = INFINITY
6:      pred[v] = NULL
7:  end
8:  add all the vertices in V to Q
9:  while (!ISEMPTY (Q))
10:  begin
11:     Extract from Q a vertex u such that
          dist[u] is minimum
12:    remove u from Q
13:    for each vertex v adjacent to u do
14:    if dist[v] > dist[u] + w(u,v)
15:    then
16:    begin
17:      dist[v] = dist[u] + w(u,v)
18:     pred[v] = u
19:    end
20:  end
```

```
21: end
```

For this pseudo code, a connected weighted graph $G = (V, E)$ with a cost function w to an edge, and a source node s are given as the inputs, and a shortest path tree T is generated as the output. In this pseudo code, `dist[v]` represents the minimum weight of the path from s to v. It is initialized by `INFINITY` which represents a larger value of any path weight. The list `Q` contains all the nodes whose shortest paths have not yet been found. `pred[v]` represents the parent vertex of v in T.

C. Requirement Analysis in Value Trace Problem for Dijkstra Algorithm

As mentioned earlier, the goal of value trace problems is to give code reading opportunities to students, so that they can analyze and understand the structure and behaviors of a code for an algorithm, while they can write and modify the code freely under required specifications in an assignment or a project when using this algorithm. Thus, we extract the points or parts of the code where students tend to find the reading and writing to be difficult.

After interviewing some students in the Java programming course in our department, we found the following three points to be considered in value trace problems for Dijkstra algorithm:

P1: How to give the information of a graph $G = (V, E, W)$ as an input to the algorithm:

Unlike a sorting algorithm, a graph is composed with complex information of vertices V, edges E, and weights W. An edge must be given to a pair of vertices with its weight. The adjacent vertex information in *line 13* in the pseudo code is also important. Thus, students may find the corresponding code parts to be difficult.

P2: How to implement *line 9* in the pseudo code:

This algorithm uses the list Q that contains the vertices whose shortest paths are not found. Novice students are not familiar with handling this kind of list which includes the initialization and the termination judgement.

P3: How to implement *line 11* in the pseudo code:

Using the list Q, this algorithm finds a vertex whose distance is the shortest among those undiscovered. However, the corresponding part of the code is considered to be hard for novice students to understand.

IV. Value Trace Problem for Dijkstra Algorithm

In this section, we present the value trace problem for *Dijkstra* algorithm based on the requirement analysis in Section III-C.

A. Adopted Java Classes

In this paper, we adopted the following three Java classes to generate the value trace problem for *Dijkstra* algorithm.

1) *WeightedGraph Class: WeightedGraph* class defines the necessary procedures to handle the input graph for *Dijkstra* algorithm. It contains the methods of *setLabel*, *getLabel*, *addEdgeWeight*, *getEdgeWeight*, and *neighbors*. *setLabel* method sets the label for the specified vertex. *getLabel* method gets the label of the specified vertex. *addEdgeWeight* method assigns the weight to the edge specified by the incident vertices. *setLabel* and *addEdgeWeight* methods are used to generate a weighted connected graph. *neighbors* method returns the list of the neighbor vertices of the vertex in the argument, which is used for line 13 in the pseudo code in Section III-B. *getEdgeWeight* method returns the weight of the edge in the argument, which is used for lines 14 and 17 in the pseudo code.

```
1: public class WeightedGraph {
2:   public int [][] edges;
3:   public String [] lables;
4:   public WeightedGraph(int verSize) {
5:     edges = int[verSize][verSize];
6:     labels = new String[verSize];
7:   }
8:   public int vertexSize() {
9:      return labels.length;
10:   }
11:   public void setLabel(int vertex ,
                    String label) {
12:     labels[vertex] = label;
13:   }
14:   public Object getLabel(int vertex) {
15:     return labels[vertex];
16:   }
17:   public void addEdgeWeight(int source,
                int target, int weight) {
18:     edges[source][target] = weight;
19:   }
20:   public int getEdgeWeight(int source,
                int   target) {
21:     return edges[source][target];
22:   }
23:   public int [] neighbors(int vertex) {
24:     int count = 0;
25:       for (int i = 0; i <
edges[vertex].length;
              i ++) {
26:       if (edges[vertex][i]>0) count++;
27:     }
28:     final int [] answer = new int[count];
29:       count = 0;
30:         for (int i = 0; i <
edges[vertex].length;
              i ++) {
31:       if (edges[vertex][i] > 0)
              answer[count++] = i;
32:     }
33:     return answer;
34:   }
35:}
```

2) *DijkstraMethod Class: DijkstraMethod* class produces the shortest path tree T, which finds the shortest path from the source node s to another node in an ascending order of the distance in the graph G [15].

```
1: public class DijkstraMethod {
2:   public static int[] dijkstra(
          WeightedGraph G, int startV) {
3:     final int [] dist = new
                  int[G.vertexSize()];
4:     final int [] pred = new
                  int[G.vertexSize()];
5:     final boolean[] visited =
                new
boolean[G.vertexSize()];
6:    for (int i = 0; i < dist.length; i ++)
{
7:       dist[i] = Integer.MAX_VALUE;
```

```
8:      }
9:      dist[startV] = 0;
10:     for (int i = 0; i < dist.length-1;
                  i ++) {
11:      final int nextV = minVertex(G, dist,
                                    visited)
;
12:        visited[nextV] = true;
13:        final int [] neighV =
                        G.neighbors(nextV);
14:       for (int j = 0; j < neighV.length;
                                      j ++) {
15:          final int nbrV = neighV[j];
16:          final int newD = dist[nextV] +
              G.getEdgeWeight(nextV, nbrV);
17:           if (dist[nbrV] > newD) {
18:             dist[nbrV] = newD;
19:             pred[nbrV] = nextV;
20:           }
21:       }
22:     }
23:     return pred;
24:    }
25:    private static int
minVertex(WeightedGraph
        G, int [] dist, boolean [] visit) {
26:      int minD = Integer.MAX_VALUE;
27:      int minV = -1;
28:      for (int i = 0; i < dist.length;
                  i ++) {
29:       if (!visit[i] && dist[i] < minD) {
30:          minV = i;
31:          minD = dist[i];
32:        }
33:      }
34:      return minV;
35:    }
36: }
```

3) *Main Class: Main* class generates an input graph to the algorithm using *WeightedGraph* class, and finds the shortest path tree using *DijkstraMethod* class.

```
1: class Main {
2:   public static void main (String[] args)
{
3:      final WeightedGraph inG =
                    new WeightedGraph(6);
4:      inG.setLabel(0, "v0");
5:      inG.setLabel(1, "v1");
6:      inG.setLabel(2, "v2");
7:      inG.setLabel(3, "v3");
8:      inG.setLabel(4, "v4");
9:      inG.setLabel(5, "v5");
10:     inG.addEdgeWeight(0,1,2);
11:     inG.addEdgeWeight(0,5,9);
12:     inG.addEdgeWeight(1,2,8);
13:     inG.addEdgeWeight(1,3,15);
14:     inG.addEdgeWeight(1,5,6);
15:     inG.addEdgeWeight(2,3,1);
16:     inG.addEdgeWeight(2,4,7 );
17:     inG.addEdgeWeight(3,4,3);
18:     inG.addEdgeWeight(5,4,3);
19:     DijkstraMethod.dijkstra(inG,0);
20:   }
21: }
```

B. References

Using the Java code and the output file, we generate a value trace problem for *Dijkstra* algorithm considering the three points in Section III-C.

1) *Problem for* **P1***:* For **P1**, *Main* class provides the graph information of this code, namely, the vertex labels, the edge weights, and the adjacent vertices to each vertex, by using *WeightedGraph* class. Since the trace of the variables for the labels and weights are obvious, the following statements are inserted to find the values of the corresponding variables to the adjacent vertices after *line 18* in *Main*:

```
A1: System.out.println("What are the
neighbors
                for each vertex ?");
A2: for (int i = 0; i < inG.vertexSize();
                i ++) {
A3:
System.out.print(inG.getLabel(i)+"->");
A4:    int [] neighV = inG.neighbors(i);
A5:    for (int j = 0; j < neighV.length;
                j ++) {
A6:    System.out.print(" "+ neighV[j]+",");
A7:    }
A8:    System.out.println();
A9: }
```

Then, by blanking the numbers randomly in the output file with 50% probability, the following value trace problem for **P1** is generated:

```
What are the neighbors for each vertex ?
v0-> 1, _1_ ,
v1-> _2_ , 3, _3_ ,
v2-> 3, 4,
v3-> _4_ ,
v4->
v5-> 4,
```

2) *Problem for* **P2** *and* **P3***:* For **P2**, *DijkstraMethod* class handles the list *Q* using an array `visited[]`. For **P3**, *DijkstraMethod* class realizes the procedure in *minVertex* method using an array `dist[]`. These arrays are defined for the vertices. Because `visited[]` is a Boolean variable and is used in *line 11*, the variable `nextV` should be traced instead. `nextV` can only be correctly traced when `visited[]` is traced accurately. To trace the values of the corresponding variables, the following statement is inserted after *line 9* in *DijkstraMethod*:

```
B1: System.out.println("What are values of
    "minVertex" and updated "dist[]" at each
    iteration step ?");
```

and after *line 12*:

```
B2: System.out.println("step"+(i+1)+"->"
    +nextV+", ");
```

and after *line 20*:

```
B3: System.out.println("("+nbrV+","
   +dist[nbrV]+ ")");
```

and after *line 21*:

```
B4: System.out.println();
```

Then, by blanking the numbers randomly in the output file with 50% probability, the following value trace problem for **P2** and **P3** is generated:

```
What are values of "minVertex" and updated
"dist[]" at each iteration step ?
```

```
step 1-> _5_ ,(1,2)( _6_ , _7_ )
step 2-> _8_ ,(2,10)( _9_ ,17)(5,8)
step 3->5,( _10_ , _11_ )
step 4->2,( _12_ , _13_ )(4, 11)
step 5-> _14_ ,( _15_ , _16_ )
step 6->4,
```

V. Evaluations

In this section, we evaluate the application of value trace problems for two graph theory algorithms, *Dijkstra* and *Prim*, to students in our department.

A. Value Trace Problem for Prim

In this evaluation, we adopted the Java code for *Prim* [16] which has the almost same code structure as *Dijkstra*. Actually, the only difference is the update of "dist[]", where the smallest edge weight between a visited vertex and each unvisited vertex is stored instead of the distance from the starting node for *Dijkstra*. Thus, we should consider the three points in Section III-C, because these algorithms are as the first step evaluation of value trace problems for graph theory algorithms. Here, we made a small change to the code for *Prim*. To let students carefully read the code for *Prim*, we changed *line 30* in *DijkstraMethod* class to `dist[i] <= minD` so that the vertex with the largest index is selected there, unlike the smallest index used for *Dijkstra*.

B. Size of Generated Value Trace Problems

First, we evaluate the problem size. Table I shows the number of vertices in the adopted graph, the number of lines (NOL) in the problem code, and the number of blanks (NOB) for each value trace problem. This table indicates that NOL is much larger and NOB is smaller, if compared with the average ones (NOL=35.2, NOB=23.4) with fundamental data structures and sorting algorithms in [5] where the applied blank selection algorithm is of 50% probability. One reason of the larger NOL comes from the graph generation procedure. Thus, it is our future goal to show important progress such as the problem code and the answer forms on the user interface to reduce the difficulty of solving problems for novice students.

TABLE I: Size of Value Trace Problem for Two Graph Theory Algorithms

algorithm	# of vertices	NOL	NOB
Dijkstra	6	123	16
Prim	6	120	16

C. Solutions by Students

For student evaluations of the solution performances, we asked 25 sophomore students in our department who are currently taking both the Java programming course and the graph theory course, to solve the value trace problems of the two graph theory algorithms. In previous semesters, they completed the C and C++ programming courses in the department before this Java course. In addition, they are learning the grammar for Java programming. After they solved the two value trace problems, we asked them to answer the 10 questions in Table II with five levels (1: No, 2: Rather No, 3: Neutral, 4: Rather Yes, 5: Yes) for the questionnaire. Table III shows the results for this questionnaire, where the average correct answer rate and number of answer submissions in JPLAS among the students are also depicted to evaluate the solution performances by them.

TABLE II: Question in Questionnaire

ID	question
Q1	Did you understand *Dijkstra/Prim* algorithm?
Q2	Did you understand the code for *Dijkstra/Prim* algorithm?
Q3	Can you write a code for *Dijkstra/Prim* algorithm?
Q4	Do you think the value trace problem for *Dijkstra/Prim* algorithm is easy?
Q5	Do you think the value trace problem is useful for code reading for *Dijkstra/Prim* algorithm?

TABLE III: Results in Solution and Questionnaire

ID	*Dijkstra*	*Prim*
correct answer rate (%)	100	91.25
# of submissions	14.6	6.68
Q1	4.12	3.96
Q2	2.84	2.8
Q3	1.64	1.56
Q4	3.28	3.04
Q5	3.48	3.4

First, we analyze the results for each question to the both algorithms. From Q1, most students can understand the algorithms sufficiently after solving these value trace problems, whereas the scores of some student's the mid-term examination in the graph theory course are much worse than the results shown above. From Q2, students feel that their understandings of the Java codes for the algorithms are moderate. From Q3, most students feel that they do not have enough confidence in writing the codes for the algorithms. This suggests that the study of code reading through solving current value trace problems is not taught enough for code writing. Our future work will further discuss the necessity to improve and use different learning tools together for code writing.

From Q4, many students think these value trace problems are not too hard or easy to solve, which indicates that the levels are suitable for students who have just started learning Java programming. This observation is also supported from the high correct answer rates. From Q5, most students agree that the value trace problem is useful for code reading study.

Then, we analyze the differences of the results between the two algorithms. From the correct answer rate and the number of submissions, students spent a long time to solve the problems for *Dijkstra* with higher answer submissions but with more correct answer rates than *Prim*. The reduce of the answer submissions between the two may suggest that students find it easier when they first understand the code for *Dijkstra* than *Prim*, despite the change of the critical statement described in Section V-A. However, this small change of code may turn the evaluation of students in Q1-Q4 to worse results for *Prim*.

D. Summary of Evaluations

Here, we summarize our evaluation results for the value trace problems for *Dijkstra* and *Prim* algorithms in this paper.

- The number of statements becomes much larger, compared with the number of blanks.
- Through solving value trace problems, students can

obtain sufficient understanding of graph theory algorithms, moderate grasp of code reading, yet lack of confidence in code writing.

- Through reading similar codes, students can speed up code reading.
- The levels of value trace problems generated for graph theory algorithms are suitable for students who have just started studying Java programming.

VI. Conclusion

In this paper, we generated value trace problems for two typical graph theory algorithms, and evaluated their problem sizes, solution performances, and effectiveness in code reading when applied to sophomore students. The results show that value trace problems for graph theory algorithms are viable learning tools for algorithm understanding and code reading, whereas additional tools are necessary for code writing. In future works, we will generalize important points for graph theory algorithms, improve the value trace problem as a learning tool of code reading including showing the problem code on the user interface, investigate the combined use of additional tools for code writing, and evaluate them in Java programming course.

References

[1] S. Cass. The 2015 top ten programming languages. [Online]. Available: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages/?utm so

[2] 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, Feb. 2013.

[3] Tana, N. Funabiki, and N. Ishihara, "A proposal of graph-based blank element selection algorithm for Java programming learning with fill-in-blank problem," in *Proc. International Multi Conference of Engineers and Computer Scientists*, pp. 448-453, March 2015.

[4] K. K. Zaw and N. Funabiki, "A concept of value trace problem for Java code reading education," in *Proc. 4th International Congress on Advanced Applied Informatics*, pp. 253-258, July 2015.

[5] 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, Sep. 2015.

[6] K. K. Zaw and N. Funabiki, "A blank line selection algorithm for value trace problem in Java programming learning assistant system," in *Proc. IEICE Society Conf*., pp. S19-S20, Sep. 2015

[7] D. S. Hochbaum. Graph algorithms and network flows. [Online]. Available: http://www.ieor.berkeley.edu/_hochbaum/files/ieor266-2012.pdf

[8] M. Stallmanny, J. Cockrell, T. Devriesz, A. McCabex, and M. Owoc, "Galant: A graph algorithm animation tool," *Tech. Report*, North Carolina State Univ., June 2014.

[9] J. Scott, P. J. Guo, and R. Davis, "A direct manipulation language for explaining algorithms," in *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 45-48, July 2014.

[10] W. I. Osman and M. M. Elmusharaf, "Effectiveness of combining algorithm and program animation: A case study with data structure course," *Issues in Informing Science and Information Technology*, pp. 155-168, vol. 11, 2014.

[11] K. Beck, *Test-driven Development: by Example*, Boston, MA: Addison-Wesley, 2002.

[12] JUnit. A simple framework to write repeatable tests. [Online]. Available: http://www.junit.org/

[13] S. K. Chang, *Data Structures and Algorithms*, River Edge, NJ: World Scientific Pub., Oct. 2003.

[14] Dijkstra algorithm: Short terms and pseudocode. [Online]. Available: http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html

[15] The Java Program: Dijkstra.java. [Online]. Available: http://cs.fit.edu/~ryan/java/programs/graph/Dijkstra-java.html

[16] The Java Program: Prim.java. [Online]. Available: http://cs.fit.edu/~ryan/java/programs/graph/Prim-java.html

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.

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 (MTU), Myanmar, in 2011, respectively. She is currently a Ph.D. candidate in the Graduate School of Natural Science and Technology at Okayama University, Japan. Her research interests include educational technology and Web service systems.

Minoru Kuribayashi received B.E., M.E., and D.E degrees from Kobe University, Kobe, Japan, in 1999, 2001, and 2004. From 2002 to 2007, he was a research associate in the Department of Electrical and Electronic Engineering, Kobe University. In 2007, he was appointed as an assistant professor at the Division of Electrical and Electronic Engineering, Kobe University. Since 2015, he has been an associate professor in the Graduate School of Natural Science and Technology, Okayama University. His research interests include digital watermarking, information security, cryptography, and coding theory. He received the Young Professionals Award from IEEE Kansai Section in 2014.

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 in the Department of Electrical Engineering and the dean in the 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.