8th International Conference on Information Technology Based Higher Education and Training, 10th to 13th July 2007, Kumamoto, JAPAN

A Fault Injection Method for Generating Error-correction Exercises in Algorithm Learning

Ryota Itoh[1], Hiroyuki Nagataki[1], Fukuhito Ooshita[1], Hirotsugu Kakugawa[1], Toshimitsu Masuzawa[1]

[1]Osaka University, 1-3, Machikaneyama-cho, Toyonaka, Osaka, Japan

ABSTRACT

In this paper we propose a method for generating error-correction exercises for undergraduate students in computer science who learn algorithms. Our main goal is to inject faults automatically into a correct source code that implements an algorithm to be studied. The proposed method utilizes design paradigm of the algorithm to determine effective fault types and positions in a source code.

We have developed a prototype system and evaluated the appropriateness of the generated exercises to algorithm study. We carried out error-correction exercises in an algorithm class, and most students evaluated that the exercises are effective for algorithm study.

1 INTRODUCTION

Because the knowledge of algorithms is fundamental in computer science[1], learning and practicing algorithms are important issues. In many algorithm classes, programming exercises are carried out. Teachers expect that the exercises help students to deepen their understandings of algorithms. For example, in an exercise to implement an abstract algorithm in a form of concrete code, teachers expect students to fully understand the algorithm to write a complete code. However, for the exercises of writing a code from scratch, students are often bothered by syntax errors and waste time, and students have less chance to concentrate their efforts to understand the algorithm[2].

To prevent the obstacle, we focused on the learning methodology known as *learning from mistakes*. This is a learning process of a student by recognizing the difference between his/her wrong answer and a correct one, and by correcting mistakes of his/her answer. Inspired by the learning process, we introduce *error-correction exercises* for algorithm learning. The outline of the exercises is as follows: (1) A teacher presents a source code that implements an algorithm to study despite it contains some faults. (2) Students must find all the faults in the presented code, and must explain why it is wrong, and (3) Students must fix the wrong code.

Because students have no need to write a source code from scratch in the error-correction exercises, they are not bothered with syntax detail of a programming language. Moreover, an answer of a student has only a small difference from a source code presented to students, a teacher spends only little time to check the answer. In contrast with it, for an exercise that each student writes a complete source code from scratch, a teacher must spend much time to check the answer.

Our primary goal is to develop a fault injector program which automatically injects faults into a correct source code that implements an algorithm to be studied, and faults injected in a source code must be effective for students to understand the algorithm. To this end, the concept of *algorithm design paradigm* (e.g., divide-and-conquer, greedy, dynamic-programming, etc.) is used in the fault injector program to help to determine fault types and positions in a source code.

2 RELATED WORKS

Related works on learning support systems for algorithms, such as RAPTOR[2] and SFC[3] are proposed as educational support tools for algorithm development. These systems provide visual programming environments for learning, and learners are not bothered by syntax of a programming language. Although such approaches seem friendly for learners, learners must learn native notation for programming. On the other hand, in our system, learners use standard programming languages such as C that they are familiar with.

As another related works based on *learning from mistakes*, learning systems for algorithm design[4] and computer programming[5] are proposed. In the systems, learning process is based on mistakes by learners themselves. However, our method presents a source code with faults to learners and the faults are injected on purpose, learning process of which may not be the same exactly. However, its learning process is essentially the same because learners recognize difference between correct and wrong answers, and fix the wrong answers. Therefore, our approach is expected to have essentially the same effect as the approach based on learning from mistakes.

As related approach for automatic generation of exercises, a system to generate fill-the-blank exercises for source code is proposed in [6]. In fill-the-blank exercises, learners immediately know where they should focus to consider in a source code because blanks are presented. On the other hand, in our approach, learners must read whole the source code carefully to find faults. For students to answer why a presented source

code with faults is wrong, they must be aware of correctness of algorithms. On the other hand, in fill-the-blank exercises, learners can answer by simply writing a part of source code what they remember without understanding correctness of algorithms.

3 ERROR-CORRECTION EXERCISES

3.1 Appropriate Faults for Exercises

The faults injected in a source code must be appropriate for students to understand the algorithm to be studied. We consider that a fault is an *appropriate fault for exercises* if it satisfies both two conditions. (1) A source code that includes the fault causes error on runtime, such as it computes wrong results, array index is out of range, and it goes into an infinite loop, (2) The fault is difficult to find and fix without full understanding of the algorithm.

The above definition is to exclude faults that are not important to understand essentials of algorithms. For example, trivial syntax errors are excluded. Although understanding syntax and semantics of a programming language is important to implement algorithms, it is not an aim of learning algorithm itself.

3.2 Automatic generation of error-correction exercises

To create an error-correction exercise manually, a teacher must prepare a complete source code that implements an algorithm to teach, and inject some faults into the source code. This process is time consuming because he/she must carefully consider and check whether the faults to be injected are appropriate faults for exercises or not for the algorithm to teach.

To reduce the workload of a teacher, the system proposed in this paper aims to automatically generate error-correction exercises with appropriate faults.

It makes easy to generate many exercise problems from (1) a single source code or (2) some source codes of different implementation of the same algorithm. Because many exercise problems can be generated, a teacher can assign different problems for each student so that the difficulty of the problem matches him/her, and it is also effective to prevent cheating, for example.

4 AUTOMATIC FAULT INJECTION

In this section we present our method to inject faults into source code automatically. Our method consists of the following steps: (1) decide a point to inject faults, which we call *fault-position*, in a source code, (2) replace the code fragment in the fault position with faulty one that is appropriate for error-correction exercises.

4.1 Decision of Fault Positions

Because what fault is appropriate for exercises is different for each algorithm, our method should determine fault positions depending on which algorithm is used in a source code. However, it is difficult to identify an algorithm used in a source code automatically. To solve this problem, we use the *algorithm design paradigm* the algorithm is based on. Algorithm design paradigm, we call it simply *paradigm*, is a framework of algorithm structure, such as divide-and-conquer and dynamic programming. Most algorithms are based on one or more paradigms. In this paper four paradigms, divide-and-conquer, recursion, dynamic programming and greedy method are considered. To determine fault positions, paradigm type is given in addition to a source code. Below, we describe fault-positions and faults that are appropriate for exercises for each paradigm.

```
1  #define N 10
2  int a[N] = { RANDOM N };
3   /* each element is a random integer from 0 to N */
4
5  void merge(int l[], int r[], int a[],
6        int llen, int rlen, int len){
7  /* snip  */
8  /* merge two sorted arrays l[llen] and r[rlen] */
9  /*  to make an array a[len] (len = llen+rlen) */
10 }
11
12 void mergeSort(int a[], int len){
13   int i;
14   int mid;
15   int left[len], right[len];
16   if(len > 1){
17     mid = len / 2;
18     for(i=0;i<mid;i++) left[i]=a[i];
19     for(i=0;i<len-mid;i++) right[i]=a[mid+i];
20     mergeSort(left, mid);
21     mergeSort(right, len-mid);
22     merge(left,right,a,mid,len-mid,len);
23   }
24 }
25
26 int main(){
27   mergeSort(a, N);
28 }
```


Figure 1: Example: mergesort

Paradigm 1: Divide-and-Conquer Divide-and-Conquer is the paradigm to solve a problem by dividing it into two or more independent smaller sub problems, solving them recursively, and combining the solutions of them to construct a solution of the original problem. Structure of the program is as follows: **(1) [solve]**: If the problem is small enough, solve it directly. **(2)**: If the problem is not small enough, **(2-1) [divide]:** Divide the problem into sub problems. **(2-2) [recursion]:** Solve sub problems recursively. **(2-3) [combine]:** Combine the solutions.

As fault positions of algorithms in this paradigm, the followings are adopted: (a) Statements or expressions for recursive calls, (b) Statements that update variables which are used as arguments of recursive calls, (c) Function-calls or return statements which appear after recursive calls, and (d) Conditional statements that checks the size of a problem. (a) corresponds to **divide**, (b) to **divide** and **recursion**, (c) to **combine** and (d) to **solve**.

Figure 1 is a sample source code which implements mergesort algorithm based on the divide-and-conquer

paradigm. In the source code, **A**,**B**,**C** and **D** are selected as the fault-positions: two recursive-calls in **A**, statements in which variables used as augments of recursive call (left, right, mid) are updated in **B**, a function call which comes after recursive-call in **C**, and a conditional statement which checks the size of the problem in **D**.

Paradigm 2: Recursion In an algorithm based on Recursion paradigm, recursive call is an important point in execution. As fault positions of algorithms in this paradigm, the followings are adopted: (a) Statements or expressions for recursive calls, (b) Statements that update variables which are used as arguments of recursive calls, and (c) Conditional statements for recursive calls.

Paradigm 3: Dynamic Programming Although Dynamic Programming paradigm is similar to Divide-and-Conquer, in Dynamic Programming, a table is used for solutions of sub problems.

In a source code based on this paradigm, a variable for the table (table variable) is important for executing the program. As fault positions of algorithms in this paradigm, the followings are adopted: (a) Assignment statements to the table variable, (b) Statements that update variables which appear in (a), and (c) Conditional statements to execute (a).

Paradigm 4: Greedy method Greedy method is the paradigm to obtain an optimal solution by iterating the best choice at each iteration step. As fault positions of algorithms in this paradigm, the followings are adopted: (a) Assignment statements which appear in the iteration, whose execution are controlled by conditional statements. (b) Statements that update variables which appear in (a), and (c) Condition statements to execute (a).

Paradigm 5: Others Algorithms that are not based on paradigms listed above are categorized into this class. In a source code of such algorithms, conditional, iteration and function call statements are candidates for fault-positions.

4.2 Fault Injection

After deciding the fault-positions from a source code, one or more positions are selected from the fault-positions randomly, and faults are injected in the positions. To inject faults automatically, we adopt a syntax-directed fault pattern injection scheme. By the syntax-directed fault pattern injection, a code fragment that corresponds to a subtree in a parse tree of a source code is substituted with other code fragment. Table 1 shows a set of rules for substitution.

However, injected faults based on the rules are not always appropriate faults for algorithm learning. So we add some restrictions based on semantics so that injected faults should be appropriate ones.

Table 1: Rules for syntax-directed fault pattern injection

Syntax of fault position	Fault pattern to inject
integer constant `c`	integer constant `0`, `c+1`, `c-1`
integer variable	concatenate `+1` or `-1`; another variable
arithmetic operator	another operator
logical operator	another operator
arithmetic expression (e.g., `A + B`)	delete a term (e.g., `A`)
logical expression (e.g., `A && B`)	delete a term (e.g., `B`)
function call	delete
conditional statement (e.g., `if (S) A else B`)	delete conditional part (e.g., `A`)

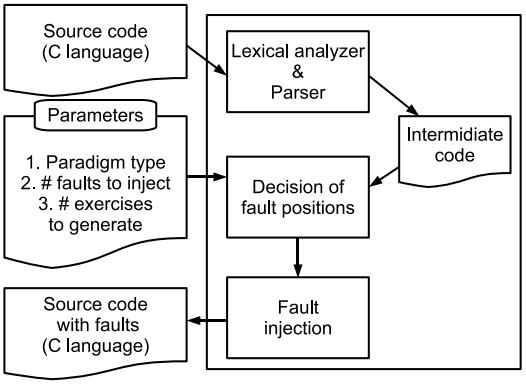

Figure 2: System overview

a) Illegal or unnatural expressions A fault pattern 'integer constant `0`' applied to division may yield an expression `i/0`, for example. But this is an obvious error, and it does not lead students to algorithm learning. So such a fault is not injected.

Expressions such as `i*1` or `i+0` are not injected either. Such expressions are unnatural so that learners can find fault positions intuitively. So they are not appropriate for algorithm learning.

b) Syntax errors A fault pattern 'another variable' may yield an expression that referrer can be undefined variable or assignment of different type of value. Such errors are not appropriate for algorithm learning itself.

So the variable is changed from `i` to `j` if `j` satisfies all of the following conditions: (1) `i` and `j` appear in the same scope, (2) variable types of `i` and `j` are the same, (3) an assignment statement of `j` to `i`, or comparison of `i` and `j` exists in the source code.

5 IMPLEMENTATION

We developed a prototype system of the proposed method. The overview of the system is shown in Figure 2. A correct source code in C is given by a teacher to the system with three parameters (1) a paradigm type, (2) the number of faults to inject, and (3) the number

of source code files to generate. Parameter (3) specifies the number of source code files to generate that contain different faults. In the system, a correct source code is parsed and translated into an intermediate code. An intermediate code is essentially a parse tree for a source code file, and it is used in the process of syntax-directed fault injection. Then, according to the given paradigm type, fault positions are decided in a source code file. Finally, faults are injected according to the rule set for syntax-directed fault pattern injection as shown in Table 1. Although our method assumes that a programming language for a source code is a subset of C, it can be applied to other languages, such as Pascal and Java. This system is developed on Windows XP (SP2). A subsystem that translates a source code in C into an intermediate code is written in C, about 1000 lines of code. Subsystems for deciding fault positions and injecting faults are written in Scheme, about 3600 lines of code. GCC 3.4.4 is used for C compiler, Flex 2.5.4 and Bison 2.1 are used for lexical analyzer generator and parser generator, respectively. SCM 5e2 is used for Scheme interpreter.

6 EVALUATION

In this section we present evaluation results of the proposed method based on comparison of automatically generated faults and manually created ones.

Faults that are generated automatically is evaluated by two measures, *recall rate* and *precision rate*.

6.1 Recall Rate

We presented seven source codes and asked six experienced persons to create three faults for each source code. They created 125 faults in total. By eliminating identical faults, we obtained 92 sample faults.

Recall rate, in this paper, is defined as $100|Y|/|X|$ (in percent), where X is a set of faults that are created by experienced persons, and Y is a set of faults in X which can be generated by the system. If $X = Y$, recall rate is 100%. Larger recall rate is better because the system generates more faults that can be created by experienced persons.

Results for recall rates are shown in Table 2. We computed two types of recall rates, named *position-only recall rate* and *position & fault recall rate* for sample faults created by experienced persons. In computing position-only recall rate, only fault positions are considered and injected faults are ignored. On the other hand, in computing position & fault recall rate, both fault positions and injected faults are considered. The average of the position-only recall rate is 77.2%. This result shows that our method properly decides positions to inject faults. However, average of position & fault recall rate is 51.1%, which means that half of faults by our method are not created by experienced persons.

Table 2: Recall rate

	Position-only Recall Rate	**Position & Fault Recall Rate**
Quicksort	50.0%	37.5%
Binary search (recursion)	83.3%	58.3%
Binary search (iteration)	92.9%	64.3%
Knapsack problem	83.3%	58.3%
Partial sums problem	81.8%	45.5%
Dijkstra's algorithm	63.6%	45.5%
Selection sort	87.5%	50.0%
Average	77.2%	51.1%

Table 3: Cross-review of manually-generated faults

	#samples rated highly		
	by all evaluators	by 4 or 5	by less than 4
Quicksort	4	10	2
Binary search (recursion)	7	4	1
Binary search (iteration)	7	6	1
Knapsack problem	8	3	1
Partial sums problem	8	3	0
Dijkstra's algorithm	6	4	1
Selection sort	6	7	3
Total	46(50.0%)	37(40.2%)	9 (9.8%)

These results seems that selection of faults to inject are not proper despite our system finds appropriate positions in a source code where to inject faults. We further analyze the results below.

We carried out a cross-review of 92 fault samples created by experienced persons. We asked experienced persons to evaluate all fault samples and to decide whether each fault sample is appropriate for exercises or not. Table 3 shows the result for this survey. For example, in case of quicksort algorithm, 4 fault samples are rated highly by all 6 persons, 10 fault samples by 4 or 5 persons, and 2 fault samples by less than 4 persons. This shows that each fault sample may not be rated highly and equally by every persons. We classified fault samples by the number of persons who created each of them. Fault samples classified as *duplicated samples* are created by two or more persons, and those classified as *unique samples* are created by only one person. Table 4 shows that automatically generated faults by our system occupy 71.4% of duplicated samples which was created by manually. On the other hand, automatically generated faults occupy only 45.1% of unique samples. This result implies that most of the faults that are supported by many persons are generated by our system.

Table 4: Recall rate of duplicated samples

	Duplicated samples	Unique samples
#samples	21	71
Position-only recall rate	85.7%	74.6%
Position & fault recall rate	71.4%	45.1%

Table 5: Precision rate

	#samples rated highly		
	by all evaluators	by 4 or 5	by less than 4
Quicksort	4	3	3
Binary search (recursion)	5	2	3
Binary search (iteration)	5	2	3
Knapsack problem	7	3	0
Partial sums problem	5	5	0
Dijkstra's algorithm	6	4	0
Selection sort	4	6	0
Total	36(51.4%)	25(35.7%)	9 (12.8%)

6.2 Precision Rate

We generated 10 faults for each of seven source codes, which are the same code as used in the previous subsection. Then we asked the six experienced persons again to evaluate whether each of generated faults is appropriate or not for exercises. We also asked each of them to write a comment if he/she decides that a presented fault is not appropriate for exercises.

Precision rate, in this paper, is defined as $100|X|/|Y|$ (in percent), where X is a set of faults in Y that are rated highly by experienced persons, and Y is a set of faults generated by the system. If $Y = X$, precision rate is 100%. Larger precision rate is better because the system generates less faults that are not appropriate for exercises.

Result for precision rate is shown in Table 5. In total, 87.1% of faults generated by the system were rated highly by at least four persons. Remarkably, None of the faults were evaluated as useless for exercises by all 6 persons. This survey shows that our system properly generates faults that are appropriate for exercises, and it rarely generates useless faults for exercises.

Comments for faults that are marked as not appropriate for exercises are classified into two categories below: (a) Students can intuitively find the fault without understanding the algorithm, and (b) Although the behavior of the presented code is different from that of algorithm, computed result is correct by coincidence. Based on the comments, we modified our system so that it did not generate faults of such types. However, we found that some instances of such faults appear to be appropriate for exercises in other context. Context-aware fault injection is left as a future research topic.

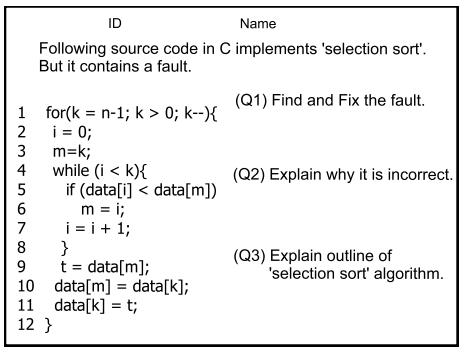

Figure 3: An example of exercise

Table 6: Choice items for Q4

Choice 1	The exercise deepened my knowledge
Choice 2	My misunderstand was fixed by the exercise.
Choice 3	I was confused by the exercise.
Choice 4	The exercise didn't deepen my knowledge
Choice 5	Others

7 EXERCISE IN CLASSROOM

We carried out error-correcting exercises in an algorithm class for sophomore students in computer science of Osaka University. Students have taken Pascal and C programming courses. Each exercise took 15 minutes long, in which students answered questions(Q1-Q3) and questionnaire for the exercise(Q4, Q5).

For a given source code, students answered the following questions: (Q1) Find and fix one or more faults, (Q2) Explain why the source code is incorrect, (Q3) Explain outline of the algorithm. Q3 has been included in the 2nd exercise and later. Figure 3 shows an example exercise used in 1st experiment. In Q4 students selected one of options shown in Table 6. Choice 1 or 2 means they could get deeper understanding with the exercise, and Choice 3 or 4 means the opposite. In Q5 students commented anything about the exercises.

We generated multiple questions for each exercise, and randomly selected one was shown to each student. Exercises were generated by our prototype system, although the 3rd exercises was generated manually. Algorithm used in the 3rd experiment wasn't based on any of paradigm we have defined, and the system couldn't generate any fault pattern even with 'other' paradigm. The statistics of exercises are shown in Table 7.

Table 8 shows the result of answers for Q4. In every exercise over 75% of students selected Choice 1 or 2, which means that most students think that the error-correction exercise is effective for understanding algorithm. Table 9 is the result of correct answers for Q1-Q3 grouped by answer of Q4. '1 or 2' is the group of students answered Choice 1 or 2 for Q4, and '3 or 4' is the group of students answered Choice 3 or 4. Only the average rate of four experiments was shown for the group '3 or 4', because the number of students chose 3

Table 7: Statistics of exercises in class

	Exp.1 Binary search	Exp.2 Selection sort	Exp.3 Shell sort	Exp.4 Quick sort
#students answered Q1,Q2,Q3	72	73	69	67
#students answered Q4,Q5	58	66	49	44
#samples of source codes	6	3	3	3

Table 8: Distribution of #learners for Q4

	Exp.1	Exp.2	Exp.3	Exp.4
Choice 1	45	48	30	32
Choice 2	8	10	7	6
Choice 3	2	1	1	2
Choice 4	3	2	4	3
Choice 5	-	5	7	1

or 4 for Q4 were very few for every exercise. Among the students chose 1 or 2 for Q4, over 30% of them remarked Q2 incorrect. On the other hand, over 20% of the students choiced 3 or 4 for Q4 remarked Q2 correct. The result means that there are some students that self-assesment and actual result of their own learning skill are different. It is likely that feedback process is important in the exercise, by which students recognize their actual learning skill.

As the answers for Q5, 67 comments in total were gathered in the four experiments. We classified the comments into 7 groups, which is shown in Table 10. Most of comments were positive, and students recognized the purpose of the exercises which is to deepen their knowledge of algorithm.

8 CONCLUSION

In this paper we proposed the method for generating error-correction exercises automatically. This method is to inject faults into a correct source code, by using algorithm design paradigm to determine fault positions and injecting faults by rules for syntax-directed fault patterns, which is well-designed for the purpose of algorithm education.

We evaluated our prototype system that implemented our method, and confirmed that the faults generated by the method were appropriate for exercises. We also used the generated exercises in an algorithm class, and most students evaluated that error-correction exercise is effective to deepen their knowledge of algorithms.

Table 9: The percentage of questions answered correctly for each learner group by Q4

	Choice for Q4				
	1 or 2				3 or 4
	Exp.1	Exp.2	Exp.3	Exp.4	Exp.1-4
#learners	53	58	37	38	18
Q1	69.8%	93.1%	75.7%	80.0%	55.6%
Q2	67.9%	69.0%	54.1%	84.2%	27.8%
(partially or fully correct)	84.9%	79.3%	64.9%	73.7%	44.4%
Q3	-	75.9%	67.6%	68.4%	53.8%
(partially or fully correct)	-	96.6%	97.3%	100%	100%

Table 10: Comments for Q5

comment	#comments
'Understanding of the algorithm is necessary to solve the question'	14
'The exercises has opportunities to read source-codes carefully'	9
'It is interesting and like a game'	10
Other positive comments	8
Negative comments	1
Suggestion about the exercise from students' point of view	17
Others	8

References

[1] SIGCSE, "Computing curricula 2001," http://www.sigcse.org/cc2001/

[2] M.C. Carlisle, T.A. Wilson, J.W. Humphries, and S.M.Hadfield "RAPTOR: a visual programming environment for teaching algorithmic problem solving," Proc. the 36th SIGCSE technical symposium on Computer science education, pp.176-180, Feb. 2005.

[3] T. Watts, "The SFC editor a graphical tool for algorithm development," Journal of Computing Sciences in Colleges, Vol.20, No.2, pp.73-85, Dec. 2004.

[4] D. Ginat, "The Greedy Trap and Learning from Mistakes," Proc. the 34th ACM Computer Science Education Symposium - SIGCSE, ACM Press, pp.11-15, Feb. 2003.

[5] K. Chiken, A. Hazeyama, Y. Miyadera, "A Programming Learning Environment Focusing on Failure Knowledge," Proceedings of the International Conference on Computers in Education, pp.1911-1920, 2004

[6] A. Kashihara, K. Kumei, K. Umeno, J. Toyoda, "How to Make Fill-in-Blank Program Problems and Its Evaluation," Transactions of the Japanese Society for Artificial Intelligence, Vol.16, No.4, pp.384-391, 2001 (in Japanese).