Detection of Random Correction from Source Code Snapshots

Yu OHNO
Dept. Information Engineering
National Institute of Technology, Nara College, Nara, Japan 639-1080
+81 743-55-6000
a0899@stdmail.nara-k.ac.jp

Hidetake UWANO
Dept. Information Engineering
National Institute of Technology, Nara College, Nara, Japan 639-1080
+81 743-55-6000
uwano@info.nara-k.ac.jp

Shinji UCHIDA
Dept. Information Engineering
National Institute of Technology, Nara College, Nara, Japan 639-1080
+81 743-55-6000
uchida@info.nara-k.ac.jp

ABSTRACT

Classifying student’s situation helps teachers to improve educational effect. In this paper, authors propose two metrics to classify the student's "random correction." Random Correction is an action that source code correction without understanding the exercise contents. We select a programming course with Online Judge System as a target, then analyze the characteristics of random correction from recorded snapshots. The result of the experiment showed that students who cannot reach perfect score have high value of both metrics; 1) a degree of imbalance corrections between source code lines, 2) the number of submitted revisions.

CCS Concepts

• Applied computing→Education • Social and professional topics→Computing education→Computing education programs→Information systems education • Social and professional topics→Computing education→Computing education programs→Software engineering education

Keywords

Programming Education; Online Judge System; Activity Estimation;

1. INTRODUCTION

Classifying student’s situation helps improve educational effect in various classes. Several studies have classified student’s situation in course [1, 2, 3]. Also many studies classify student’s situation in programming course with snapshots [4, 5, 6, 7, 8]. Snapshot is a set of the program state contains source code, date of create/update, errors and so on. Existing studies grasp student who falls “pitfall” during a course from the snapshots. In this paper, authors target programming course with Online Judge System (OJS) to classify students who need teacher’s assistance.

OJS is a system designed for evaluation of source code submitted by system users. The system compiles, executes and scores source code automatically based on the execution result. Students is notified their source code’s score immediately after the submission of the code. OJS operates with (or as a plugin of) Learning Management System (LMS), submits the source code to prepared exercises, and student learns by receiving score.

In programming course without OJS, students submit the source code only once and cannot see the score, therefore the students cannot realize the source code is not fulfill the requirement. Conversely in programming course with OJS, students know the latest submission is not fulfill the requirements immediately. Therefore they can try to correct the error until the source code fulfill the requirements.

In this paper, we aim to classify "random correction" which is observed in programming course with OJS. In the programming course with OJS, students repeatedly submit source code until the requirements are fulfilled (i.e. they obtain the perfect score.) However, some students attempt to obtain perfect score by correct the source code repeatedly and randomly. Fulfill the exercise requirements with random correction disturb the student understandings for new syntax and/or algorithm. We propose two metrics to classify the random correction from the edit history of source code. Classification of random correction allows teachers encourage the student to understand their errors or lecture contents.

DOI: http://dx.doi.org/10.1145/12345.67890

2. RELATED WORKS

Fujiwara et al. analyzed novice programmer’s source code edit history recorded during programming exercises in two viewpoints: 1) heuristic analysis by skilled programmers and 2) Levenshtein distance at the token level source code based on lexical analysis [4]. As a result, authors found several patterns in novice behavior while they falls pitfalls, i.e. novices edit a code snippet that unrelated to the content of the exercise. Authors also found fluctuation of edit distance will denote the situation that novice is in pitfalls.

Jadud et al. proposed a method to classify behaviors of error creation and correction during programming lectures for novices [5]. The method uses Error Quotient (EQ) which is calculated by the number of errors creation and correction. The research result shows two significant negative correlations: between EQ and exercise scores, and between EQ and grade of a lecture exam. Watson et al. proposed modified EQ metrics Watwin, that consider how much time student think for errors [6]. Ahadi et al. classify students with machine learning based on EQ and Watwin [7].

Blikstein et al. analyzed the situation and behavior of the students in the open-ended programming tasks. Blikstein calculated the number of compiles, the size of the source code, the number of compile success/failure, etc. from the snapshot [8].

Our research targets a programming course with OJS, whereas the previous studies intended to a programming course without OJS. Our research also differs from previous studies in that we focus on random correction as a classification targets.

3. PREPARATION

3.1 Programming Course with OJS

Target lecture is "programming 1" for novice students in our college: Programming exercise follow teacher's lecture about a study-unit, such as "for" statement or array data. Each student edits a source code at online IDE (Integrated Development Environment) then submits to OJS. OJS compiles and executes the submitted source code and judges whether it matches the test case prepared by the teacher. A test case is a set of an input and expected output for testing whether the created program satisfies the requirements. OJS calculates the score by the following formula according to the number of the matching output.

$$score = \frac{\#\text{Matches of test cases}}{\#\text{Test cases}} * 100\ [\text{points}] \qquad (1)$$

The system displays the judge result including score, correct/incorrect of each test case, and/or compile error and runtime error. The student repeatedly corrects and submits the source code until the score reaches 100 points (perfect score) or the time limit is come. The all source code is recorded as a revision with the submission date, time, and score.

3.2 Random Correction

We define "random correction" as an action that source code correction without understanding the exercise contents. Figure 1 shows an example of random correction. The figure shows an edit history of a certain student in a certain task. Each square describes code snippet at each revisions. The student submitted a source code twenty times within five minutes from first revision. The student corrects only the argument of the print statement through the all revisions. The student is considered to change the statement without any consideration for cause of the error because the student applied the same correction pattern in Revision 2 and Revision 6. Students who make a random correction may tend to apply their correction pattern multiple times to a particular line which is considered to be the cause of the error. As an ideal condition, students should consider and investigate the cause of point-reduction (or errors) to understand their misunderstandings. However, if the student reach a right answer by random correction, they finish the exercise without understanding the specifications and algorithms of the given program, the newly learned syntax, and so on.

Also non-novice programmers make random corrections during their programming. The non-novice programmer tries endless debugging and see what happens [9]. Ben-Ari et al. defines such behavior as bricolage and states that it is inefficient to iterate source code correction without considering the cause of the error. Classification of random correction during exercise allows teachers to encourage the students to understand the study-units.

4. PROPOSED METRICS

We propose two metrics to classify random correction activity from student's source code edit histories. The classification helps teachers to support student understandings. The metrics calculated based on the following two hypotheses for random correction in certain time period t. In this paper, we uses elapsed time from first revision of each student in each exercise as t.

H1: Fix same code line frequently

H2: Correct and/or compile source code frequently

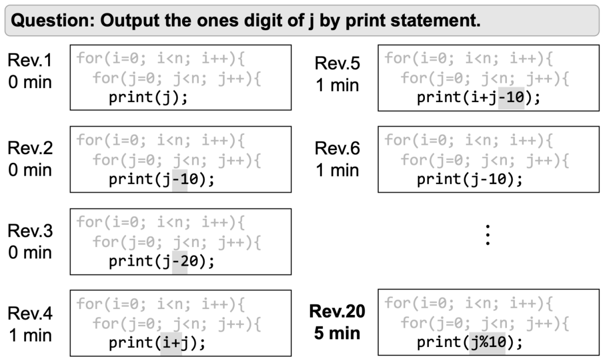
$Freq(t)$ is a degree of imbalance corrections between source code lines. A student who makes a random correction may tend to apply his/her correction pattern multiple times to a particular line which is considered as a cause of the error: that is, the student correct a single line repeatedly (hypothesis H1.) To calculate the metric, we assign ID to each line of the source code (Figure 2.) When a new line is added at any revision, a new ID is assigned to the line. $Freq(t)$ is a variance which calculated from the number of corrections at each $ID = \{id_1, id_2, \cdots, id_k, \cdots, id_N\}$.

$$Freq(t) = \frac{1}{N}\sum_{k=1}^{N}\left(\frac{c_k}{s} - \mu\right)^2 \qquad (2)$$

Here, t is the elapsed time from first revision, c_k is the number of corrections at id_k, s and μ are calculated from following formulas.

$$s = \sum_{k=1}^{N} c_k\,, \qquad \mu = \frac{1}{N}\sum_{k=1}^{N}\frac{c_k}{s} \qquad (3)$$

We suppose that $Freq(t)$ tends to high by random correction.



Rev.1
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(j);
```
Rev.2
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(j-10);
```
Rev.3
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(j-20);
```
Rev.4
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(i+j);
```
Rev.5
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(i+j-10);
```
Rev.6
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(j-10);
```
Rev.20
```
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    print(j%10);
```

Figure 1. An example of Random Correction

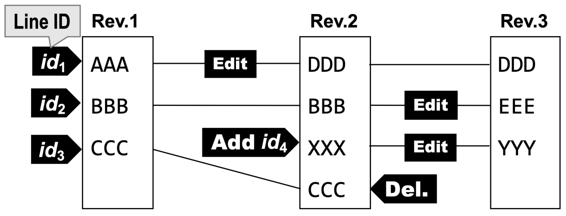


Figure 2. Line ID Assign

$Revs(t)$ is the number of revisions submitted within t minutes from first revision. When students make a random correction, they tends to apply their correction pattern multiple times to a particular line which is considered as the cause of the error. Therefore, the number of corrections and compilations within a time increases (hypothesis H2.)

5. EXPERIMENTS

This section shows an experiment result to evaluate the effectiveness of two metrics, $Freq(t)$ and $Revs(t)$.

5.1 Dataset

We uses OJS logs which is collected at programming lecture in our college as a dataset. The course is a 90-minutes lecture of Java programming basics for novice students with exercise. Course teacher makes the exercise, then collects/evaluates the student submissions through CodeRunner which provides OJS functions to Moodle Learning Management System (LMS). CodeRunner records the source code submitted by the students with the assignment ID, student ID, revision number, submission date, and *score*. In addition, students who obtain perfect score in first revision or submit once (i.e. $Revs(t) = 1$) are excluded from analysis because $Freq(t)$ can not be calculated if $Revs(t) = 1$. In this paper, we conduct experiments on 1,474 source codes (11 exercises, 227 people) collected from June 21 to September 1, 2017.

5.2 Evaluation

We focus on the change in the score of the first revision for each exercise by each student, $score_1$, and the score of latest revision within t, $score(t)$. In the programming lecture with OJS, students who reach the perfect score (100) seldom make random correction because they already know how to correct source code. Students who reach 1-99 points revision after several 0 point revisions grasp the part of the code that should be correct via previous revisions. After the 1-99 points revision, some of the students find the cause of the error, then reach perfect score in a few revisions. On the other hand, some of them make random correction without thinking about the cause of error, hence these students stay at 1-99 points in following revisions. Also, students who stays zero points cannot even grasp where to modify source code. In this paper, each student is classified as two groups from $score_1$: $score_1 = 0$ and $1 \leq score_1 \leq 99$. Also the same students are classified as three groups from $score(t)$: $score(t) = 0$, $1 \leq score(t) \leq 99$, and $score(t) = 100$. Figure 3 shows six groups which is divided from different score transitions through $score_1$ to $score(t)$. Here, $S_{(0,mid)}$ means the score of student's first revision is 0, then the score changes to the range 1-99 at latest revision within the time t. We evaluate a difference of $Freq(t)$ and $Revs(t)$ among groups

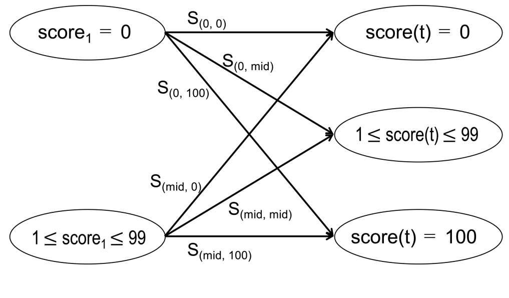


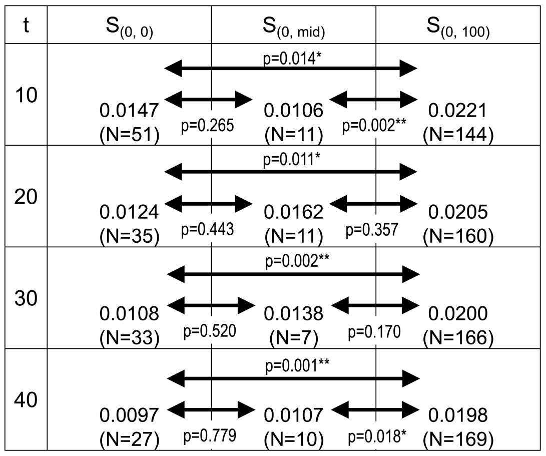
Figure 3. Six Group based on $score_1$ and $score(t)$

to understand whether these metrics relate with scores changes. In this paper, $S_{(mid,0)}$, $S_{(mid,mid)}$, and $S_{(mid,100)}$ are excluded from the analysis since a lack the number of data. We aim to classify random corrections during exercise, hence early detection from first revision is required. In this paper, we use 40 minutes as the maximum of t and calculate each metrics and score in every 10 minute for analysis.

5.3 Results and Discussions

Figure 4 and 5 shows the average values of $Freq(t)$ and $Revs(t)$ with the results of the Welch's t-test at $t = 10$, 20, 30 and 40, respectively. As shown in Figure 4, $Freq(t)$ of $S_{(0,100)}$ is the highest in all t, and significant difference with $S_{(0,0)}$. Also the smaller $Freq(t)$ was observed when the t is increased at $S_{(0,0)}$ and $S_{(0,100)}$. Figure 5 shows $Revs(t)$ is higher in the order of $S_{(0,mid)}$, $S_{(0,0)}$, $S_{(0,100)}$; and $S_{(0,0)}$ is significantly higher than $S_{(0,100)}$ at all t. In all groups, $Revs(t)$ increases as t increases.

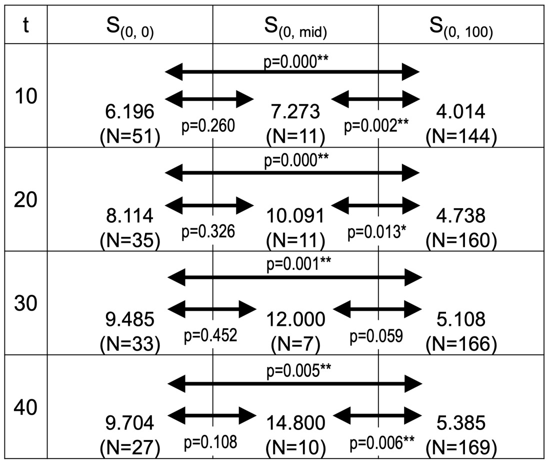
The results showed the $S_{(0,100)}$ group has the highest $Freq(t)$ and the lowest $Revs(t)$ compared with other groups. This indicates that the students in the $S_{(0,100)}$ group have reached correct



t	$S_{(0,0)}$		$S_{(0,mid)}$		$S_{(0,100)}$	
10	0.0147 (N=51)	p=0.265	0.0106 (N=11)	p=0.002**	0.0221 (N=144)	p=0.014*
20	0.0124 (N=35)	p=0.443	0.0162 (N=11)	p=0.357	0.0205 (N=160)	p=0.011*
30	0.0108 (N=33)	p=0.520	0.0138 (N=7)	p=0.170	0.0200 (N=166)	p=0.002**
40	0.0097 (N=27)	p=0.779	0.0107 (N=10)	p=0.018*	0.0198 (N=169)	p=0.001**

*p < 0.05, **p < 0.01

Figure 4. Average Value of $Freq(t)$ (N=206)



t	$S_{(0,0)}$		$S_{(0,mid)}$		$S_{(0,100)}$	
10	6.196 (N=51)	p=0.260	7.273 (N=11)	p=0.002**	4.014 (N=144)	p=0.000**
20	8.114 (N=35)	p=0.326	10.091 (N=11)	p=0.013*	4.738 (N=160)	p=0.000**
30	9.485 (N=33)	p=0.452	12.000 (N=7)	p=0.059	5.108 (N=166)	p=0.001**
40	9.704 (N=27)	p=0.108	14.800 (N=10)	p=0.006**	5.385 (N=169)	p=0.005**

*p < 0.05, **p < 0.01

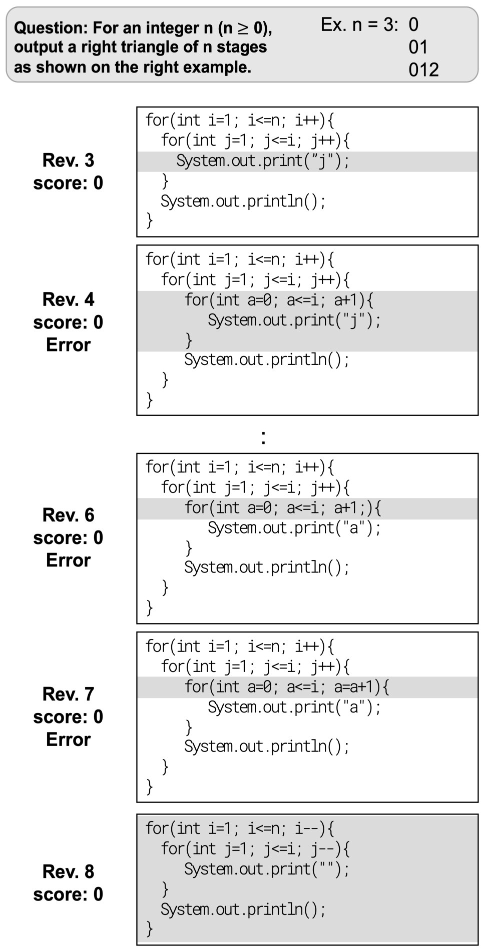
Figure 5. Average Value of $Revs(t)$ (N=206)

answers by correcting the fewer lines within the several revisions. The students seemed to understand the subjects of the exercise and/or the cause of the error, hence the student could correct with a few modification. Such correction pattern may cause from a strict scoring method by the OJS. OJS evaluates source code by matching the characters between the execution results and the test case. Therefore, even if the algorithm of the submission is correct to an assignment, OJS judges the code as incorrect when the format of output (e.g. number of blanks and the letters' uppercase/lowercase) are different. We found multiple cases of the correction pattern in students' edit history such as source code outputs "Output: 0" or "OutPut: 0" for correct output "OutPut:0". Also, the students corrected these errors in several revisions, and reached to the correct answer.

In contrast, $S_{(0,0)}$ group has a significantly higher $Revs(t)$ and lower $Freq(t)$ compared with $S_{(0,100)}$ group. This indicates that the students in the $S_{(0,0)}$ group corrects a wide range of source code with many revisions, however they cannot reach the correct answer. Figure 6 shows a edit history of a student in $S_{(0,0)}$ group. The student change a wide range of the source code and occurs multiple compile error. These edit pattern shows the student has less understanding of errors, and try to correct the code by random correction. Therefore the lower $Freq(t)$ may useful for random correction detection.

$Revs(t)$ of the $S_{(0,mid)}$ group is higher than the other two groups and $Freq(t)$ is higher than $S_{(0,0)}$. From these results, students of $S_{(0,mid)}$ group partially reached correct answers by correcting specific lines through many revisions compared to the students in the $S_{(0,0)}$ group. The result shows the $S_{(0,mid)}$ group performs random correction as defined in this paper (H1 and H2 in Section 4.) Figure 7 shows an edit history of the student in the $S_{(0,mid)}$ group. The student stay 0 score from first revision to revision 5, then get 25 score at revision 6. After the revision 8, the student submits source code with small modification five times in four minutes, however the score is still low. The each modification shows the student has less understanding about the cause of low score, and try to correct the code by random correction.

The result also shows the differences between $S_{(0,0)}$ and $S_{(0,100)}$ groups. In both groups, the higher t value means the lower $Freq(t)$, and the higher $Revs(t)$, however the $S_{(0,0)}$ group has a stronger tendency. The number of student in $S_{(0,0)}$ decreases at later t, because some student reach the correct source code during the modifications, then move to the $S_{(0,100)}$. Hence the student in $S_{(0,0)}$ at later t means the student stays non-perfect score in the long time. The lower $Freq(t)$ and the higher $Revs(t)$ of $S_{(0,0)}$ in later t indicates that the student modifies source code in a wider range through many revisions, that means the student cannot



Question: For an integer n (n ≥ 0), output a right triangle of n stages as shown on the right example. Ex. n = 3: 0 / 01 / 012

Rev. 3 score: 0
```
for(int i=1; i<=n; i++){
  for(int j=1; j<=i; j++){
    System.out.print("j");
  }
  System.out.println();
}
```

Rev. 4 score: 0 Error
```
for(int i=1; i<=n; i++){
  for(int j=1; j<=i; j++){
     for(int a=0; a<=i; a+1){
        System.out.print("j");
     }
     System.out.println();
  }
}
```

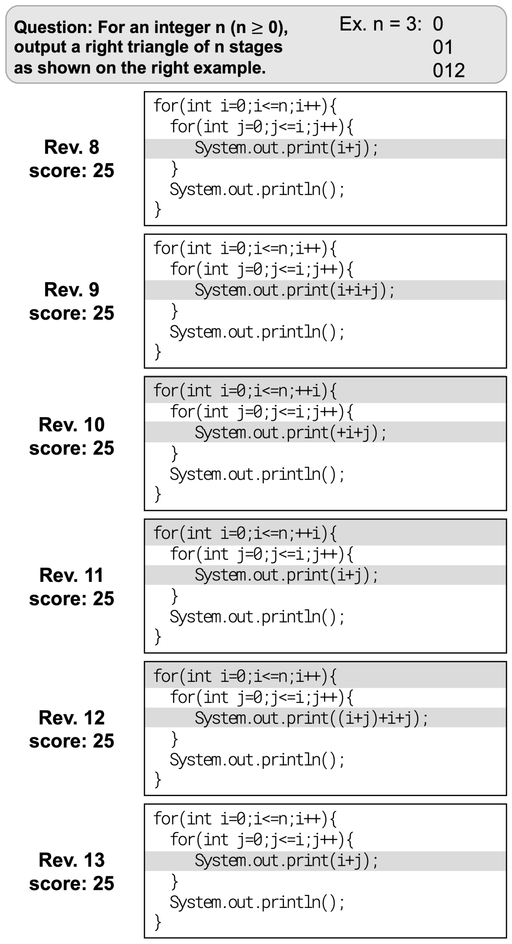
:

Rev. 6 score: 0 Error
```
for(int i=1; i<=n; i++){
  for(int j=1; j<=i; j++){
     for(int a=0; a<=i; a+1;){
        System.out.print("a");
     }
     System.out.println();
  }
}
```

Rev. 7 score: 0 Error
```
for(int i=1; i<=n; i++){
  for(int j=1; j<=i; j++){
     for(int a=0; a<=i; a=a+1){
        System.out.print("a");
     }
     System.out.println();
  }
}
```

Rev. 8 score: 0
```
for(int i=1; i<=n; i--){
  for(int j=1; j<=i; j--){
     System.out.print("");
  }
  System.out.println();
}
```

Figure 6. Example of $S_{(0,0)}$ Student's Edit History



Question: For an integer n (n ≥ 0), output a right triangle of n stages as shown on the right example. Ex. n = 3: 0 / 01 / 012

Rev. 8 score: 25
```
for(int i=0;i<=n;i++){
  for(int j=0;j<=i;j++){
     System.out.print(i+j);
  }
  System.out.println();
}
```

Rev. 9 score: 25
```
for(int i=0;i<=n;i++){
  for(int j=0;j<=i;j++){
     System.out.print(i+i+j);
  }
  System.out.println();
}
```

Rev. 10 score: 25
```
for(int i=0;i<=n;++i){
  for(int j=0;j<=i;j++){
     System.out.print(+i+j);
  }
  System.out.println();
}
```

Rev. 11 score: 25
```
for(int i=0;i<=n;++i){
  for(int j=0;j<=i;j++){
     System.out.print(i+j);
  }
  System.out.println();
}
```

Rev. 12 score: 25
```
for(int i=0;i<=n;i++){
  for(int j=0;j<=i;j++){
     System.out.print((i+j)+i+j);
  }
  System.out.println();
}
```

Rev. 13 score: 25
```
for(int i=0;i<=n;i++){
  for(int j=0;j<=i;j++){
     System.out.print(i+j);
  }
  System.out.println();
}
```

Figure 7. Example of $S_{(0,mid)}$ Student's Edit History

understand the cause of low scores. The result suggests that the metrics is useful indicator to predict future score of each student based on their source code edit history. Similar tendency was observed at $Revs(t)$ in $S_{(0,mid)}$ (the higher value in the later t.) However the number of student at each t is small, hence the future research is required.

6. CONCLUSION

In this paper, we propose metrics to classify random correction from the snapshot of the source code submitted to the OJS. $Freq(t)$ represents a degree of imbalance corrections between source code lines, and $Revs(t)$ represents the number of submitted revisions. The result of the experiment shows that students who cannot reach perfect score had high value of both metrics. Also the snap shots of non-perfect score students describe that they correct the source code randomly. We think distinction of random correction from student activity during programming exercise increases the chance of efficient coaching. Dynamic detection of random correction by these metrics will useful functional enhancement of OJS to efficient teaching.

As a future work, time series analysis of metrics is an important work for more detail understanding of random correction. The result of snap shot analysis suggests that student who takes a non-zero (and non-perfect) scored submission after the continued zero-point submissions start a random correction. That is, even students who make a random corrections did not such correction method all the time. They identifies the code snippet which need to modify via OJS output, then starts "tries often enough" method to get a perfect score. Comparing the metrics before and after the score change will clarify the change of student activity.

Classification of random corrections using machine learning is also a future work. OJS with auto-classification of random correction (or other student activities) improves the education effect of e-class without teachers. In this paper, we analyzed the metrics and score at the point in the time t elapsed from the first revision. Score estimation at after the certain time (e.g. $t + 30$) from current metrics is an interesting work.

7. REFERENCES

[1] R. Yoshihashi, D. Shimada, and H. Iyatomi, "Feasibility Study on Evaluation of Audience's Concentration in the Classroom with Deep Convolutional Neural Networks," In International Conference on Teaching, Assessment and Learning for Engineering (TALE 2014), pp.288-292, 2014.

[2] B. Baradwaj and S. Pal, "Mining Educational Data to Analyze Students' Performance," International Journal of Advanced Computer Science and Applications, Vol.2, No.6, pp.63-69, 2011.

[3] S. Pal, "Analysis and Mining of Educational Data for Predicting the Performance of Students," International Journal of Electronics Communication and Computer Engineering, Vol.4, No.5, pp.1560-1565, 2013.

[4] K. Fujiwara, K. Fushida, H. Tamada, H. Igaki, and N. Yoshida, "Why Novice Programmers Fall into a Pitfall?: Coding Pattern Analysis in Programming Exercise," In International Workshop on Empirical Software Engineering in Practice (IWESEP 2012), pp.46-51, 2012.

[5] M. Jadud, "Methods and Tools for Exploring Novice Compilation Behavior," In International Workshop on Computing Education Research (ICER 2006), pp.73-84, 2006.

[6] C. Watson, F. W. B. Li, and J. L. Godwin, "Predicting Performance in an Introductory Programming Course by Logging and Analyzing Student Programming Behavior," In International Conference on Advanced Learning Technologies (ICALT 2013), pp.319-323, 2013.

[7] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen, "Exploring Machine Learning Methods to Automatically Identify Students in Need of Assistance," In Annual International Conference on International Computing Education Research (ICER 2015), pp.121-130, 2015.

[8] P. Blikstein, "Using Learning Analytics to Assess Students' Behavior in Open-ended Programming Tasks," In International Conference on Learning Analytics and Knowledge (LAK 2011), pp.110-116, 2011.

[9] M. Ben-Ari, "Constructivism in computer science education," Journal of Computers in Mathematics and Science Teaching, Vol.20, No.6, pp.45-73, 2001.