A Blank Element Selection Algorithm for Element Fill-in-blank Problems in Client-side Web Programming

Huiyu Qi, Nobuo Funabiki, Khaing Hsu Wai, Flasma Veronicha Hendryanna, Khin Thet Mon, Mustika Mentari, and Wen Chung Kao

***Abstract*—Nowadays, *web applications* play central roles in information systems using the Internet. Then, *client-side web programming* using *HTML*, *CSS*, and *JavaScript* should be mastered first by novice students. Previously, we have presented the *element fill-in-blank problem (EFP)* for its self-study. An EFP instance requests to fill in the blank elements in the given source code by referring to the screenshots of the corresponding web page. The correctness of any answer is marked through *string matching*. However, these blanks were manually selected by considering the importance of elements and the uniqueness of their correct answers. In this paper, we propose a *blank element selection algorithm* to automatically generate a new EFP instance from a given source code for *client-side web programming*. We define the seven rules on blank element selections from the code, and implement the procedure in *Python* using the open source *BeautifulSoup* and *regular expressions*. ‘For evaluations, we applied the algorithm to the 47 source codes used for manual generations and obtained the better EFP instances with more blanks. Besides, we verified the effectiveness by generating 10 new instances with the algorithm and assigning them to 40 students. In addition, we extended its application to three source codes for games and verified the effectiveness by assigning them to 20 students, to further validate the applicability of the algorithm in EFP instance generations. We also evaluated the relationships between the number of blanks, the number of lines in source codes, the submission times and answer rates of students to further assess the adaptability of the algorithm. These results allow us to measure the algorithm's versatility in generating a wide range of EFP instances and contributes to comprehensive understanding of instance difficulties and learning outcomes.**

***Index Terms*—client-side web programming, element fill-in-blank problem, blank-element selection, algorithm, Python, BeautifulSoup, regular expression, correlation coefficient.**

Manuscript received March 24, 2023; revised December 23, 2023.

H. Qi is a PhD student of the Department of Information and Communication Systems, Okayama University, Okayama, Japan, email: pbr17iqa@s.okayama-u.ac.jp

N. Funabiki is a professor of the Department of Information and Communication Systems, Okayama University, Okayama, Japan, email: funabiki@okayama-u.ac.jp

K. H. Wai is a PhD student of the Department of Information and Communication Systems, Okayama University, Okayama, Japan, email: khainghsuwai@s.okayama-u.ac.jp

K. T. Mon is a PhD student of the Department of Information and Communication Systems, Okayama University, Okayama, Japan, email: p3x78b2r@s.okayama-u.ac.jp

M. Mentari is a PhD student of the Department of Information and Communication Systems, Okayama University, Okayama, Japan, email: pqt85hm5@s.okayama-u.ac.jp

F. V. Hendryanna is an employee of the PT Telkom Indonesia Tbk, Malang, Indonesia. e-mail: veronichaflasma@gmail.com

W. C Kao is a professor of the Department of Electrical Engineering National Taiwan Normal University Taipei, Taiwan. e-mail: jungkao@ntun.edu.tw

I. Introduction

Nowadays, most people around the world agree with values that computers will bring to all the aspects of our private and professional lives. Computers can flexibly solve diverse practical problems in societies by adopting proper programs to them. It can be said that *programming* provides conveniences and solutions for us. As a result, a variety of computer systems have been designed and implemented with different architecture and programming languages, where *programming* plays significant roles. Among various computer systems that have been used, *web applications* will be the most important, providing vital tools and various services in our daily lives with use of the Internet. *Web-client programming* using *hyper text markup language (HTML)*, *cascading style sheets (CSS)*, and *JavaScript* is essential to provide dynamic behaviors of web pages on web browsers in web applications [1]. With its rich libraries and short coding capabilities, *web-client programming* has become a common way to implement a variety of user interfaces, both on personal computers and smartphones. Nevertheless, even now, *web-client programming* courses are not offered as standard courses in many universities around the world, due to time and instructors limitations. More conventional and fundamental programming languages such as em C or *Java* should be taught first before *web-client programming* is taught in the university curriculum. Therefore, there is a strong desire to have self-learning tools to promote *web-client programming* among people who have not formally studied it in universities. Previously, we have developed a web-based *programming learning assistant system (PLAS)* for self-studies of popular programming languages, such as *C*, *C++*, *Java*, *Python*, and *JavaScript*. PLAS provides several types of programming exercise problems at different difficulty levels to meet the different learning needs of students. By solving offered exercise problems in PLAS, students can gradually advance the stage of programming learning. In particular, PLAS provides the *grammar-concept understanding problem (GUP)* [2], the *value trace problem (VTP)* [3]-[5], the *code modification problem (CMP)* [6], the *element fill-in-blank problem (EFP)* [7][8], the *code completion problem (CCP)* [9], and the *code writing problem (CWP)* [10]. For any problem type, the correctness of the student's answer is automatically checked by *unit test* for CWP or by *string matching* with the correct answers for the others. The outline and the learning goal of each problem type are described as follows:

- *GUP* reminds the knowledge and concepts of reserved words and common libraries in the source code, for *grammar* study.
- *VTP* questions the values of important variables and output messages in the source code, for *code reading* study.
- *CMP* asks to modify the source code to satisfy the requested output, for *library use* study.
- *EFP* requests to fill in the blank elements in the source code with their originals by understanding the syntax and semantics, for *code understanding* study.
- *CCP* is different from *EFP* only in no show of the locations of the missing elements.
- *CWP* needs to write the source code from scratch that can pass the given test code, for *coding* study.

It is expected that by solving these problem types in this order, students can continue programming learning without dropping out, and can reach the sufficient level of proficiency. For *web-client programming*, we have implemented *GUP* and *VTP* so far. Previously, we have presented the *element fill-in-blank problem (EFP)* for self-study of *web-client programming*. An EFP instance requests to fill in the blank elements in the given source code by referring to the *screenshots* of the corresponding web page. The correctness of any answer is marked through *string matching* with the stored unique correct answer. In a web page, *HTML* and *CSS* use *tags* in the *document object model (DOM)* to define the static components. Then, *JavaScript* refers to them to provide dynamic changes or actions to the components through libraries. Since it is important to understand how to relate the three languages together in a source code, the corresponding elements are often blanked in the EFP instance. An EFP instance intends that a student can effectively understand the source code including use of *tags* with *libraries* by reading it while watching the screenshots of the web page. In our previous studies, we generated 21 EFP instances on *basic grammar topics* [11] and 26 ones on *applicative topics* [12] respectively. Based on the generated instances, we also analyzed the rules for the blank element selection [13]. We evaluated the EFP instances by asking students to solve them and analyzing the results. It was concluded that the instances on *basic grammar topics* are suitable for novice students, whereas the instances on *applicative topics* are hard for them. However, in previous studies, the blank elements were manually selected by considering the importance of the elements and the uniqueness of their correct answers. This manual selection can cause the high load to a teacher in generating new EFP instances. Actually, it may not be easy to correctly select the blank elements from a source code that have unique correct answers. Therefore, the algorithm to automatically select the blank elements from the given code is desired. In this paper, we propose a *blank element selection algorithm* to automatically generate a new EFP instance from a given source code for *client-side web programming*. For this algorithm, we define the *seven rules* on the blank element selection from the code. We implement the algorithm procedure in *Python* using an open source *BeautifulSoup* [14] and *regular expressions* [15]. For evaluations of the proposal, we applied the algorithm to 47 *client-side web programming* source codes that were used in manual EFP generations, and examined the differences of the generated EFP instances from the previous ones. Then, the same results were basically obtained except for random tiebreak resolutions when two or more elements have the same priority. Thus, the correctness of the proposal was confirmed. Besides, we generated 10 EFP instances from newly collected source codes by applying the algorithm, and assigned them to students in Okayama University and State Polytechnic of Malang who have studied *C* and *Java programming* in the department courses but have not studied *web-client programming* in any course. It is noted that some of them had solved EFP instances for other programming languages. Their solution results confirmed the validity of the proposal in applications to new source codes. In addition, we extended the evaluation of the algorithm to the learning field of game-based *web-client programming*. We newly generated three EFP instances using the source codes for simple video games by applying the algorithm. These source codes have longer lines than the previous. These instances are assigned to 20 students where the solution results confirmed the validity. Furthermore, we examined the relationships between the number of lines of the source code, the number of generated blanks, the submission times and the correct answer rate of students. Notably, an EFP instance from a longer code and a higher number of blanks generally exhibited the lower correctness rates. This finding emphasizes the importance of providing students with progressively challenging instances, which will allow them to build *web-client programming* competence over time. The rest of this paper is organized as follows: Section II introduces adopted technologies in the proposal. Section III discusses related works in literature. Section IV reviews our previous works. Section V presents the blank element selection algorithm. Section VI, Section VII, Section VIII and Section IX evaluates the algorithm by comparisons with manual generations, new instance generations, game-based instance generations, and instance characteristics using correlation coefficient respectively. Section X concludes this paper with future works.

II. Adopted Technologies

In this section, we review adopted technologies to implement the proposed algorithm.

A. BeautifulSoup

BeautifulSoup is included in the *Python's built-in standard library*. It is an open source parser for *web scraping* that retrieves and analyzes data from *HTML* and *XML* files. *HTML* are widely used in websites that are published on the Internet. It offers a high degree of cross-platform flexibility. In the algorithm implementation, the *BeautifulSoup* library is used to parse the web page to find the necessary elements by using the *regular expression* together. It has the moderate execution speed and the high document fault tolerance.

The parsing process using *BeautifulSoup* in a Python program is given as follows:

1) Import *BeautifulSoup* from *bs4*. For example, *from bs4 import BeautifulSoup*
2) Read the *HTML* file by `f: html = f.read()`.
3) Make the *BeautifulSoup* object with `'html.parser'` by `soup = BeautifulSoup (html, 'html.parser')`.

4) Use the *find_all()* function under the *soup* object to find the name of the element that needs to be matched. For example, `soup.find_All('tag name')` is used.

B. Regular expression

Regular expression represents a sequence of special characters that can help checking whether a given string matches the certain pattern or not.

The matching process using *regular expression* in a *Python* program is given as follows:

1) Import the *regular expression* module as *re*. For example, *import re*.
2) Use *re.compile(pattern, flag=0)* to convert *regular expressions* to objects. This function generates a *regular expression* object based on a pattern string and optional flags arguments. This object has a set of methods for *regular expression* matching and replacement.
3) Use the *search()* function to search the *HTML* files for the existence of the string *regular expression* declared in the *re.compile()* function.
4) If the string character is exists, the first successful match will be returned.

III. Related Works

In this section, we discuss related works in literature in web programming, regular expression, and web scraping.

A. Web Programming

In [16], Lahtinen et al. studied the difficulties in learning programming to support developing learning materials for basic programming courses through an international survey of opinions from teachers and students. They include program design to solve the task, functionality divisions into procedures, bug finding, error handling, and library use.

In [17], Knutas et al. presented an automated assessment system for web programming assignments. It was implemented using the *Cypress* end-to-end testing framework and integrated with *Learning Management Systems (LMS)* using the *Learning Tools Interoperability (LTI)* interface.

In [18], Kar et al. showed the gap between the skill requirements of software industries and web programming courses in universities, and proposed two different courses for frontend programming and backend one.

In [19], Arawjo et al. proposed a strategy for teaching programming using gamified semantics, blending game elements with programming exercises to increase engagement and understanding. It outlines the method's efficacy in fostering active learning and improved comprehension of programming concepts through interactive and playful educational experiences.

B. Regular Expression

In [20], Chapman et al. explored the backgrounds of use of *regular expressions*, and the characteristics and behavioral similarities in open source *Python* projects. They analyzed about 4,000 projects in *GitHub* and extracted nearly 14,000 unique *regular expression* patterns. The most difficult part of using *regular expressions* is composing and reading them. They proposed a method to measure the similarity of behaviors between different *regular expressions* by generating strings that match one *regular expression* and pair testing the rest. They found that capturing the contents of parentheses, searching for separators and matching alternative values are common behaviors.

In [21], Larson et al. presented *Automatic Checking of Regular Expressions (ACRE)*. *ACRE* takes a *regular expression* as the input and performs 11 different checks on it based on common mistakes. It is simple to use where the user just enters a *regular expression* and presses the button. The incorrect part is highlighted.

C. Web Scraping

In [22], Onyenwe et al. introduced the steps of using *BeautifulSoup* to extract information from e-commerce web pages. The main steps process scraping data from the pages. They conducted experiments using the largest online shopping site in Finland. They developed web pages, made web scraping source codes, and processed scraped data. The results showed that this approach to e-commerce sites can improve product search operations.

In [23], Thivaharan et al. presented three *Python* libraries, *BeautifulSoup*, *LXml*, and *RegEx*, which can be used to extract digital contents scattered across the Internet. *RegEx* has the inherent drawback of the limited rule extractions of web pages with more internal *tags*. As a result, it can perform only moderately complex contexts. Other two libraries have the ability to extract web page contents in critical contexts. They demonstrated overwhelming advantages of *RegEx* in different scenarios.

In [24], Darmawan et al. aimed to determine the performance of the web scraping method with the application of multi-processing. Four web crawling methods were selected in the experiment, namely *CSS selector*, *HTML DOM*, *RegEx*, and *XPath*. The results show that *RegEx* uses the least memory size, *XPath* takes the least CPU time, and *CSS Selector* method takes the smallest bandwidth usage.

In [25], Khder et al. proposed that the web scraping is a highly useful tool in the information age, and an essential one at different fields in any company that wishes to maintain the online presence. They introduced the application of data crawling in various fields, such as AI, data science, big data, business intelligence, cloud computing, and cyber security. Moreover, the automatic data extraction or web scraping is becoming more prevalent in corporate and academic research projects.

In [27], Singrodia et al. reviewed various aspects of *Web Scraper*. The authors describe the working principle, advantages and disadvantages of web scrap systems, and finally the application of the web scrap system. Web scraping is an automatic web data extraction function instead of manual copying. The goal of web scraper is to focus on the conversion of unstructured data while saving it in an organized database. It provides error-free data, saves the time to provide quick results and stores all the data stored in one location. This facilitates access and makes analyzing the data easier. This article also introduces tools for *Web Scraping*, including the *rvest platform* and *regular expressions*.

In [28], Andersson et al. proposed a structured way to build a *Python-based web scraper* that collects data from

TimeEdit and saves it. The user can then upload this text file to a dynamic website where the data is extracted from the file and saved to a database that is predetermined and unique to that user. The pre-study showed that it was feasible to build the web scraper in pure code, but a lot of time had to be spent analyzing how *TimeEdit* worked and how to scrape the right information. By continually iterating on the problems encountered, it was possible to build the entire platform with some shortcomings.

In [29], Opera et al. investigated the opportunities to extract data from web pages. They proposed solutions for extracting historical data from websites that do not provide APIs or csv files. The data is extracted from this website using *BeautifulSoup* and *Selenium* libraries. However, they simulated a case using only *Selenium* and showed an almost 6-fold increase in execution time. Therefore, the authors recommend using both *BeautifulSoup* and *Selenium* libraries for this task.

D. Correlation Analysis

In [30], Sahoo et al. provided a comprehensive guide to leveraging Python for exploratory data analysis (EDA). It covers essential techniques, libraries like Pandas and Matplotlib, and practical examples, empowering researchers and data scientists to efficiently analyze and visualize datasets for insights and informed decision-making in diverse fields.

In [31], Gupta et al. investigated sentiment analysis on Twitter using machine learning. Employing various algorithms in Python, the study explores the effectiveness of sentiment classification. The findings contribute to understanding sentiment trends on Twitter and the performance of machine learning models in this context.

In [32], Sial et al. conducted a comparative analysis of data visualization libraries, focusing on Matplotlib and Seaborn in Python. Evaluating their features and performance, it provides insights into their strengths and weaknesses. The study aids researchers and data scientists in selecting the most suitable library for diverse visualization tasks, enhancing data exploration.

In [33] et al. provided a comprehensive overview and comparison of free Python libraries for data mining and big data analysis. Assessing the features and capabilities of various libraries, it guides researchers and analysts in selecting appropriate tools for extracting insights from large datasets, contributing to the field of data science.

IV. Review of Previous Works

In this section, we review our previous works on the *element fill-in-blank problem (EFP)* for *web-client programming*.

A. EFP Instance for Web-client Programming

In an EFP instance for *web-client programming*, a source code composed of *HTML*, *CSS*, and *JavaScript* with several blanks and a set of screenshots in the corresponding web page are given to a student. The first screenshot illustrates the web page that is generated by the source code. The second or other screenshot is the web page that will be generated when the user takes some input action on the page. For example, click a button on the web page. Students are asked to understand the source code and the output page by referring to the screenshots and filling in the blanks with the appropriate elements. The correctness of each answer will be checked by *string matching* with the original element in the source code.

B. Blank Element Candidates

The source code for *web-client programming* typically is composed by three different languages, *HTML*, *CSS*, and *JavaScript*. Therefore, in an EFP instance, the following elements in the source code can be blank:

- HTML: tag element, property, id and its name, output text message,
- CSS: property,
- JavaScript: reserved word, identifier(function and variable name), id and its name, library class/method, output text message.

The tag element in *HTML* is the most basic unit in the *HTML* language and is the most important component of *HTML*. Here, *tag* is a fixed sequence of characters that is being described in the *HTML* syntax defined to represent a specific function. For example, the JavaScript code is described between `<script>` tag and `</script>` tag in the source code. *Property* is a fixed string of characters that is defined in *HTML* or *CSS* syntax to represent a property of a particular kind. *Id* is a string of characters, which is defined by the code author and stands for a DOM component in *HTML*. *Reserved word* is a fixed sequence of characters that has been defined in *JavaScript* grammar to represent a specific function. *Identifier* is a character sequence, defined by the code author, representing a variable or a function.

From our observations in a lot of simple source codes in *web-client programming*, we found that the elements for the following seven components should be selected for blanks from a source code: 1) tag element, 2) CSS syntax, 3) JavaScript identifier, 4) JavaScript reserved word, 5) JavaScript library class/method, 6) Id name, and 7) text message.

C. Limitations of Blank Element Selection

To be a unique correct answer for any blank, some rules must be followed at selecting blanks.

First, all the elements representing the same *id* or *identifier* should not be blanked in the source code. Otherwise, it becomes impossible for a student to fill in them with the original element. For example, *tag* appears in pair in *HTML*. If all of them are blanked, it would be very difficult for many students to answer them correctly. Thus, one of them should be left as a hint. Actually, in our blank selections, only one element among them is randomly selected for the blank to make it easy.

Second, the property representing the *text size* or the *font type* should not be blanked in the source code, because it is difficult or impossible to distinguish them from the screenshot of the web page. On the other hand, the *color* property can be easily distinguished when a basic color is used, which can be blanked.

D. Example of Element Fill-in-blank Problem Generation

Here, we explain the manual EFP instance generation procedure using a simple example of ID=4 for *basic topics* in [11].

```
Problem #4 source code
<html>
<head>
 <title>contentChange</title>
 <script>
  function myFunction(){
    document.getElementById("myPar").
        innerHTML="Hello World";
    document.getElementById("myDiv").
        innerHTML="How are you?";
    document.getElementById("myBtn").
        innerHTML="Stop Click";
  }
 </script>
</head>
<body>
  <h1>My Web Page</h1>
  <p id="myPar">I am a paragraph.</p>
  <div id="myDiv">I am a div.</div>
  <p>
   <button onclick="myFunction()" id="myBtn">
       Click here</button>
  </p>
  <p>When you click the button above, the two
      elements will change.</p>
</body>
</html>
```

Fig. 1: Source code.

```
Problem #4 source code
< html>
< head>
 < title>contentChange< /title>
 < script>
  _1_  myFunction(){
    document.getElementById("myPar").
        innerHTML=" _2_ ";
      _3_ .getElementById(" _4_ ").innerHTML="
          How are you?";
    document. _5_ ("myBtn"). _6_ =" _7_ ";
  }
 < /script>
< /head>
< body>
  < h1> _8_ < /h1>
  < p id=" _9_ "> I am a paragraph.< /p>
  < div id="myDiv"> _10_ < /div>
  < p>
  < button onclick=" _11_ " id=" _12_ "> _13_
      < / _14_ >
  < /p>
  < p>When you click the button above, the two
      elements will change.< /p>
< /body>
< /html>

Answer:
function,,Hello World,,document,,myDiv,,
getElementById,,innerHTML,,Stop Click,,
My Web Page,,myPar,,I am a div.,,
myFunction(),,myBtn,,Click here,,button,,
```

Fig. 2: Input text file.

1) Source Code: First, a proper source code should be selected to generate the EFP instance that can cover the topics in *web-client programming* to be studied there. In this EFP example, the source code in Figure 1 is selected to study the topic of using JavaScript functions to change a text message by clicking a button. In the generated web page, when the button is clicked, the two text messages on the paragraphs and the note inside the button will be changed. Through solving this instance, it is expected for a student to learn how to call the *JavaScript function* by the *HTML* button click using the *id* and how to change the messages inside and outside the button.

2) Source Code with Blanks and Correct Answers: Next, the blank elements are manually selected from the source code, to make the *input text file* to the *answer interface generator* that has been implemented by *Java*. Here, the file in Figure 2 was generated from the source code in Figure 1. Each blank element is expressed by two underbars with the blank number, such as `_1_`. The correct answers to the blanks are included at the last of the file, where the answers to different blanks are separated by the double commas `,,`.

3) Answer Interface: Finally, the *answer interface generator* is executed with the input text file, to generate the necessary *HTML/CSS/JavaScript* files for this EFP instance. After that, the screenshots of the necessary web pages are added into the *HTML* file manually, so that students can answer the blanks correctly by referring to them.

Figure 3 shows the answer interface. It runs on a web browser, and allows both the online and offline use, since the answer marking is processed by running the *JavaScript* program on the browser. The correct answers are encrypted using *SHA256* to avoid cheating by students.

The left side of the interface shows the source code with the answer forms to fill in the corresponding blanks. The right side shows the screenshots of the initial web page and the page after the button click. When the "Answer" button in the interface is clicked, the *JavaScript* marking function is executed to check the correctness of the answers through *string matching* with the correct ones. If the answer is not correct, the background color of the input form becomes red. If it is correct, it becomes white. A student can repeat answer submissions until all the answers become correct or he/she gives up answering them.

V. Blank Element Selection Algorithm

In this section, we present the seven rules for the blank element selection, the blank element selection algorithm based on them, and its implementation in Python.

A. Seven Rules for Blank Element Selection

In the algorithm, the following seven rules are applied to select the blank elements from the given source code that consists of *HTML*, *CSS*, and *JavaScript*:

1) *Tag element* in *HTML* should be selected because the same *tag* appears in pair where either one should be selected randomly for the blank so that the remaining one will be the hint.
2) *Property* in *HTML* and *CSS* should be selected when the value can be guessed from the screenshot. In our implementation, *color* and *width* attributes are selected.
3) *JavaScript identifier* should be selected because the same *identifier* appears in *HTML* twice or more times

Problem #4

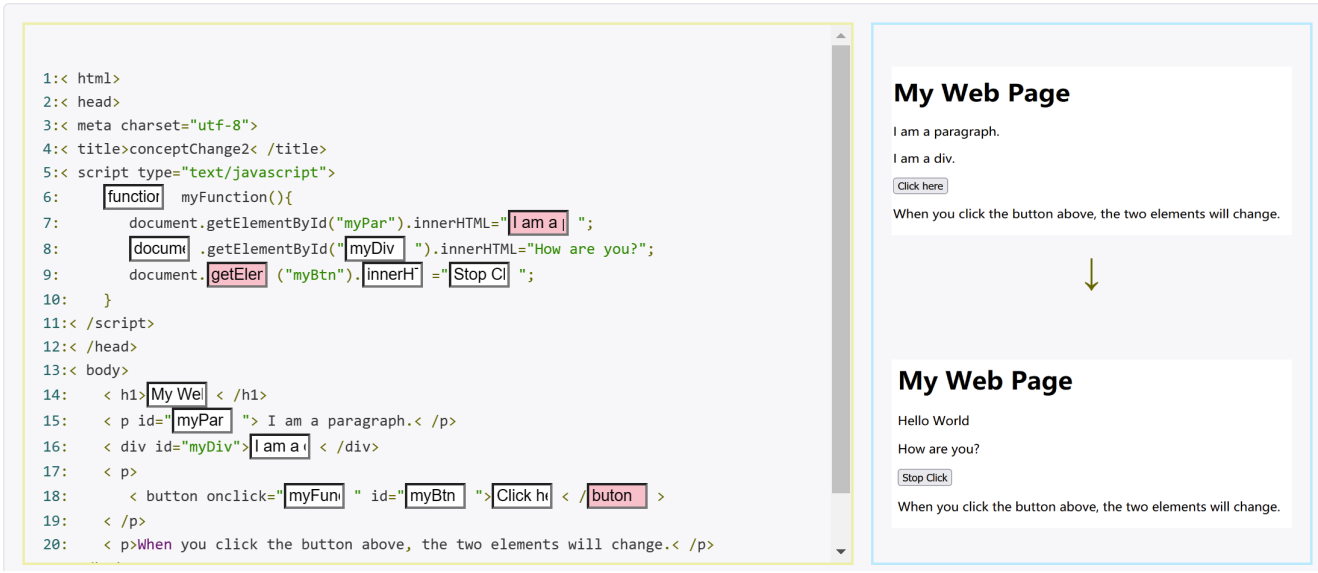


Fig. 3: Interface of ID=4 for *basic topics*.

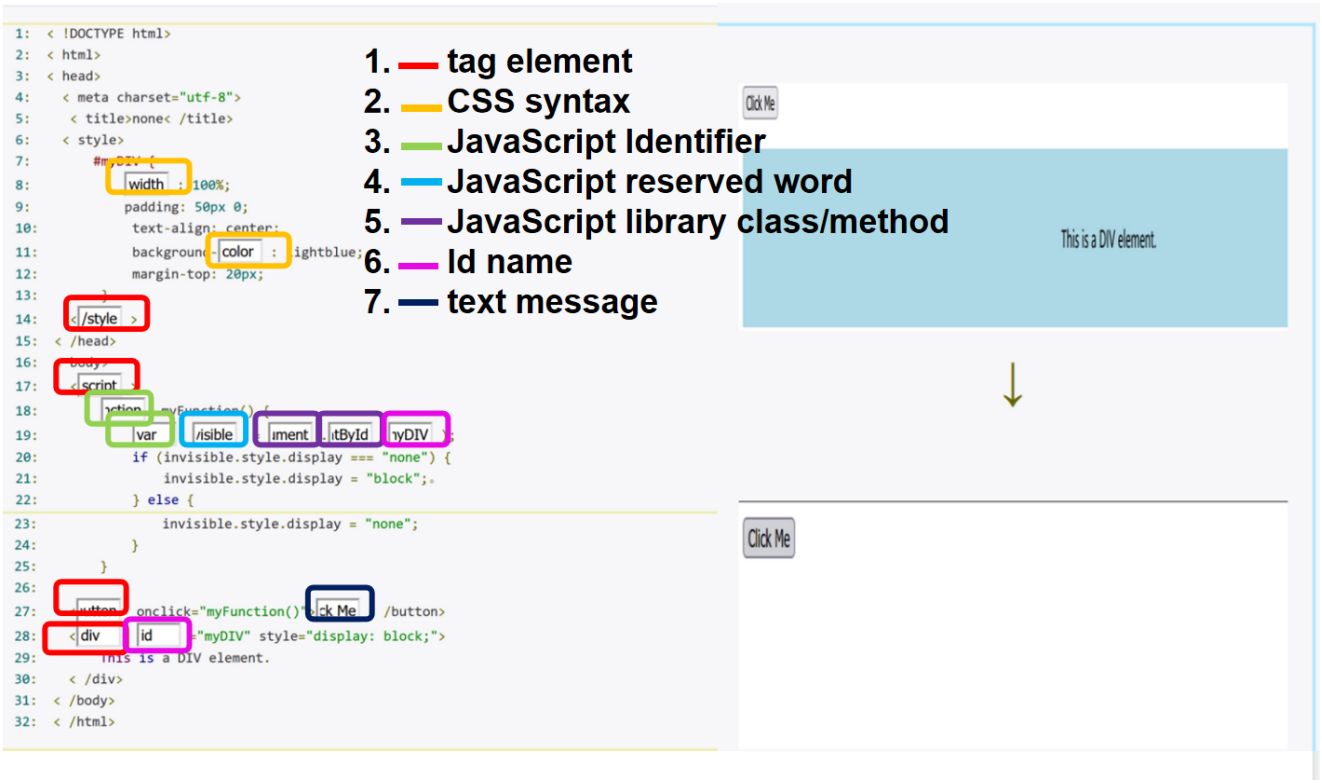


Fig. 4: Interface of ID=10 for *basic topics* by algorithm.

where the elements except for the randomly selected one should be blanked so that the selected one will be the hint.

4) *JavaScript reserved word* should be selected because it has been defined in *JavaScript* grammar.
5) *JavaScript library class/method* should be selected because it has been defined in *JavaScript* programming.
6) *Id name* in *HTML* and *JavaScript* should be selected because the same *id* appears both in *HTML* and *JavaScript* where either one should be selected randomly for the blank so that the remaining one will be the hint.
7) *Text message* in *HTML* and *JavaScript* should be selected because it appears in the screenshot of the generated web page.

B. Flexibility in Rule Application

In this subsection, we will explore the flexibility of the blank element selection rules.

1) Adjusting the instance difficulty with the blank element selection rules: a notable feature of the blank element selection rules is the ability to adjust the difficulty of

generated EFP instances. By skillfully manipulating the rules, we can generate EFP instance sets of varying complexity.

The *property selection rule* plays a key role in this process. This rule selects elements based on their positions in the nested *HTML* structure and their relevance to the *CSS* style. For example, if our goal is to let students solve more complex *web-client programming* instances, the *CSS syntax* rule could be configured to select the elements that are deeply embedded in the *HTML* hierarchy, such as the element's position, length width, font, and other properties. Conversely, if the goal is to provide basic grammar practices suitable for novice students, the rules can be adapted to target elements closer to the surface of the *HTML* structure, such as a simple background color. In this way, the *property selection rule* becomes an effective educational tool for educators to adjust the complexity of the instance according to their teaching goals.

2) Customizing the learning objectives with the blank element selection rules: the blank element selection rules are regarded as invaluable tools for educators to customize specific learning objectives for the EFP instances. These rules cover all the aspects of *web-client programming*, making it possible to align the instances with different educational objectives. Educators can choose to apply a single rule or a combination of multiple rules, depending on the desired educational focus. This precision ensures that students master the specific concepts in *web-client programming*. Consider a situation where the goal is to deepen understanding of *JavaScript* grammar. By applying the rules specifically related to *JavaScript* grammar, instances to achieve this specific educational goal can be generated. This strategy helps students establish clear learning objectives and directs them to focus on mastering important aspects of *web-client programming*.

   Figure 5 exemplifies the ability of the *blank element selection rules* to align learning objectives. This instance focuses on *JavaScript* grammar. The ability to adapt is aided by the algorithm's ability to customize the selection of the blank elements to align with various concepts. As a result, each instance can be designed to achieve the predefined learning objectives. This approach to instance generations greatly increases the relevance and effectiveness in learning *web-client programming*. Through providing a series of learning situations, each of which meets different educational objectives, educators can provide students with more engaging and productive learning experiences in the field of *web-client programming*.

In summary, incorporating the flexible blank element selection rules into EFP instance generations gives us the ability to design instances that are not only highly relevant to the *web-client programming* learning process but also have clear learning goals associated with each rule. This adaptability extends to customizing instance difficulties, refining learning objectives, and ultimately resulting in targeted and effective learning experiences in *web-client programming*.

C. The Method of Blank Element Selection

Through the technologies presented in Section III. Here we use two main methods to automatically select the blank elements. The first method is using the *regular expression* and the second is using the *BeautifulSoup*.

1) Selecting Blank Elements Using Regular Expression: Among the seven rules described above, we will use *regular expression* to match the corresponding string elements in the first six rules. For example, matching the variable names. As is well known, *JavaScript* define variables in the format `var a = 1;`. So first, we use the *re. compile()* function to find the content between var and the equal sign, and then delete the space to get the final variable name.

2) Selecting Blank Elements Using BeautifulSoup: For the last text information, we will use *BeautifulSoup* for string matching. For example, in Figure 14 we use the `find_all('button')` function to search for the button element in the *HTML* file. Then we use the *string()* function to extract the text content of the button directly.

D. Algorithm Procedure

Based on the above mentioned seven rules, we present the algorithm procedure of automatically selecting the blank elements from a given source code to generate a new EFP instance for *web-client programming* as follows:

1) Select the rules from the seven ones for the blank element selection to be applied if necessary. Otherwise, all the seven rules will be applied.
2) Use *BeautifulSoup* and *regular expression* to find every element to be blanked out by the rule from the source code, the implementation details of each rule are described in this Section V-G.
3) Replace each found element in the source code with the specified notation for a blank to make the input text file.
4) Write the replaced original elements after the source code in the input text file as the correct answers.

E. EFP Instance Generation Procedure

Using the algorithm, a new EFP instance for *web-client programming* can be created by the procedure below:

1) Choose a *web client programming* source code from a website or textbook that contains the elements to be learned.
2) Generate the web page through the operation of the source code on a web browser to collect the necessary screenshots.
3) Apply the *blank element selection algorithm* to make the input text file.
4) Run *answer interface generator* with the input text file to generate the *HTML*, *CSS*, and *JavaScript* files for the answer interface on a web browser.
5) Replace the elements in the source code by the *HTML* entities with their numbers and names automatically, since the *HTML* tags become not visible on the browser otherwise.
6) Complete the new EFP instance by adding the collected screenshots to the *HTML* file.

Fig. 5: Interface of ID=4 for *JavaScript* grammar.

```
1  < html>
2  < head>
3   < title>none< /title>
4    < style>
5      #myDIV {
6       _1_ : 100%;
7       padding: 50px 0;
8       text-align: center;
9       background- _2_ : lightblue;
10      margin-top: 20px;
11     }
12   < _3_ >
13 < /head>
14 < body>
15   < _4_ >
16    _5_  myFunction() {
17      _6_  _7_ =  _8_ . _9_ ( _10_ );
18     if (invisible.style.display === "none") {
19       invisible.style.display = "block";
20     } else {
21        invisible.style.display = "none";
22     }
23    }
24   < /script>
25   < _11_ onclick="myFunction()"> _12_ < /
       button>
26   < _13_  _14_ ="myDIV" style="display: block;
       ">
27     This is a DIV element.
28   < /div>
29 < /body>
30 < /html>
31
32 Answer:
33 width,,color,,/style,,script,,function,,var,,
34 document,,getElementById,,"myDIV",,/script,,
35 Click Me,,/button,,id,,/div,,
```

Fig. 6: Input text file of ID=10 for *basic topics* by algorithm.

F. Application Example

Here, we discuss an application example of the proposal.

Figure 6 shows the *input text file* that is generated by the proposed algorithm. The original source code makes the web page that shows the message "This is a DIV element" with the *lightblue* background and erases it with no background when the "Click Me" button is clicked.

Figure 3 shows the answer interface by applying the *answer interface generator* to this file. This EFP instance contains 14 blanks that cover all of the seven rules as suggested in the figure. It intends that students understand the usage of the display attributes in the style method.

G. Algorithm Implementation

Now, we present the algorithm implementation for each of the seven rules using the EFP instance in Figure 4.

1) Tag Element Selection Rule: A tag in *HTML* is the important element to determine the layout of the web page. In this paper, we selected commonly used tags. To extract the tags from the source code by *string matching*, the *regular expression* is used. Figure 7 shows the *Python* code, where 1) the *random* module is imported, 2) the *random.random()* function is used to generate a random number smaller than 1, 3) if the random number is smaller than 0.5, the starting tag, such as <button>, is extracted, and otherwise, the ending tag, such as </button>, extracted.

2) Property Selection Rule: A *property* in *HTML* and *CSS* determines the attribute of the web page. Among a lot of properties, only *color* and *width* are selected for the blank, because they can be known from the screenshots. Figure 8 shows the *Python* code to extract the string of *color* using the *regular expression*. The *group()* function is used to extract the first *color* in the code and avoid every *color* in the source code to be blanked.

3) JavaScript Indentifier Selection Rule: A variable or a function is considered in the indentifier selection rule.

1) variable: A new variable is first defined after `var`. Figure 9 shows the *Python* code to find a variable name

```
import random
randomNum = random.random()
if randomNum <= 0.5:
  tag = re.compile(r'<button>')
  resultTag = tag.findall(html)
  resultTagStr = ''.join(resultTag)
  if resultTagStr:
      blanks.append(resultTagStr.strip('<>'))
  else:
      pass
else:
    tag = re.compile(r'<\Wbutton>')
  resultTag = tag.findall(html)
  resultTagStr = ''.join(resultTag)
  if resultTagStr:
     blanks.append(resultTagStr.strip('<>'))
  else:
     pass
```

Fig. 7: Blank Selection Rule and Algorithm for Tag

```
cssAttr = re.compile(r'color')
resultCssAttr = cssAttr.search(html)
if resultCssAttr:
   blanks.append(resultCssAttr.group(0))
else:
   pass
```

Fig. 8: *Python* code for *color* selection rule.

in the source code. The *search()* function is used with the *regular expression* to represent a variable name pattern.

```
varName = re.compile(r'var\s(.*)\s\=')
resultVarName = varName.search(html)
if resultVarName:
   blanks.append(resultVarName.group(1))
else:
   pass
```

Fig. 9: *Python* code for variable name in identifier selection rule.

2) function: A function is one of the basic components in *JavaScript* that encapsulates a block of statements that can be called and executed repeatedly. A function name ends with `(...)`. Figure 10 shows the *Python* code to find a function name in the source code. The *search()* function is used with the *regular expression* to represent a function name pattern.

```
funcN = re.compile(r'\s\w*[(][)]')
resultFuncTwo = funcN.search(html)
if resultFuncTwo:
  blanks.append(resultFuncTwo.group(0))
else:
  pass
```

Fig. 10: *Python* code for function name in identifier selection rule.

4) JavaScript Reserved Word Selection Rule: Regarding the *Javascript* reserved word, only the basic one is considered, such as *var*, *return*, and *function*. Besides, the commonly used word for *web-client programming* is also considered, such as *alert* and *prompt*. Figure 11 shows the *Python* code to find `var` as an example *JavaScript reserved word* in the source code.

```
resultVar = re.search('var', html)
if resultVar:
  blanks.append(resultVar.group())
else:
  pass
```

Fig. 11: *Python* code for JavaScript reserved word selection rule.

5) JavaScript Library Class/Method Selection Rule: *JavaScript* offers rich libraries to write source codes easily and simply to generate web pages for *web-client programming*. For example, *getElementById()* returns the element object that matches the specific *id*. Figure 12 shows the *Python* code to find *getElementById()* using the *regular expression* as an example *JavaScript library class/method* in the source code.

```
get = re.compile(r'getElementById')
resultGet = get.search(html)
if resultGet:
   blanks.append(resultGet.group())
else:
   pass
```

Fig. 12: *Python* code for JavaScript library class/method election rule.

6) Id Name Selection Rule: The *id* is used to describe the link of an object among *HTML*, *CSS*, and *JavaScript*, which must be mastered by students. Figure 13 shows the *Python* code to find an *id name* using the *regular expression*.

```
y = re.compile(r'[(]("".*"")[)]')
resultIdName = y.search(html)
if resultIdName:
   blanks.append(resultIdName.group(0).strip('
       ()'))
else:
   pass
```

Fig. 13: *Python* code for id name selection rule.

7) Text Message Selection Rule: A *text message* often appears in a web page to describe the information to be informed. Since writing a code to output a text message in a web page is the first step of studying *web-client programming*, it is very important. Figure 14 shows the *Python* code to find the *text messages* in the *button* elements in the source code using the *BeautifulSoup* library where `soup` represents the *BeautifulSoup* object. The *button* elements in *HTML* are extracted from `soup` using the *find_all()* function. Then, the *text message* in each *button* element is extracted using the *string* function.

VI. Evaluation by Comparisons with Manual Generations

First, we evaluate the validity of the blank element selection algorithm by comparing the generated EFP instances with the manual ones in our previous studies.

```
for txtBtn in soup.find_all('button'):
  resultText = txtBtn.string
  if resultText:
    blanks.append(resultText)
  else:
    pass
```

Fig. 14: *Python* code for text message selection rule.

A. Comparison Summary

Tables I and II show the topic, the total number of lines and the number of lines for *JavaScript*, and the number of manually selected blanks for each of the EFP instances for *basic topics* and for *applicable topics* in our previous studies, respectively. For comparisons, the number of blanks generated by the proposed algorithm for the same source code is also shown there.

In the EFP instances for *basic topics*, the total number of manually generated blanks was 229, whereas the total number of automatically generated blanks was 325. In the EFP instances for *applicable topics*, the total number of manually generated blanks was 403, whereas the total number of automatically generated blanks was 530. The results show that the algorithm selected more blanks than the manual.

When the generated EFP instances from the same source code are compared, it is observed that more *HTML* tags and *JavaScript* library methods are selected by the algorithm, while the others are the same except for random selections. Thus, the correctness of the proposal was confirmed.

```
<html>
<head>
  <title>ColorChange</title>
  <script>
    function color(str) {
      document.body.style.backgroundColor =
          str;
    }
  </script>
</head>
<body>
   <input type="button" value="turn to red"
       onclick="color('red')" />
   <input type="button" value="turn to yellow"
        onclick="color('yellow')" />
   <input type="button" value="turn to blue"
       onclick="color('blue')" />
   <input type="button" value="turn to green"
       onclick="color('green')" />
</body>
</html>
```

Fig. 15: Source code of ID=6 for *basic topics*.

At the algorithm selection, only one element among multiple ones for the same word is selected for blank. On the other hand, at the manual selection, two or more elements were sometimes selected for blanks. Therefore, for four instances, the manual selection selected more blanks than the algorithm one. For example, in the source code of ID=6 for *basic topics* in Figure 15, the same words appear several times to make the page in Figure 16. In such an instance, the manual selection selected more blanks than the algorithm selection by repeatedly selecting the same words. In future works, we will improve the algorithm to properly select blank elements for the same word when they appear several times in the source code, by considering the importance or difficulty among them as the blank element.

TABLE I: EFP instances for basic topics

ID	topic of lines	manual blanks	algorithm blanks
1	object1	5	10
2	object2	5	12
3	changing content1	6	17
4	changing content2	14	21
5	alert() function	4	13
6	changing color	16	13
7	Date() function	10	20
8	prompt() function (subtraction)	6	8
9	prompt() function (add)	8	11
10	easy counter	11	21
11	click button	11	14
12	onmouse over and out	10	10
13	setTimeout() function	7	14
14	array	10	11
15	multiplication calculation	10	13
16	change background	16	17
17	try catch	10	19
18	try catch final (number)	14	22
19	try catch final (NaN)	18	21
20	custom timer	11	18
21	fixed timer	19	20
	total blanks (average)	229 (10.9)	325 (15.5)

TABLE II: EFP instances for advanced topics

ID	topic	manual blanks	algorithm blanks
1	radio button (alert message)	10	18
2	radio button (show selected value)	16	23
3	radio button (text message)	24	18
4	checkbox (alert message)	18	18
5	checkbox (disable option)	12	19
6	checkbox (default option)	16	20
7	video	14	22
8	video (control function)	28	25
9	image function (modify image size)	9	17
10	image function (change image)	8	18
11	mouse pointer	8	11
12	select function (default option)	19	28
13	select function (disable option)	17	22
14	select function (output option)	16	26
15	select function (remove option)	13	24
16	image function (upload image)	14	22
17	first css web page	15	23
18	change page	14	23
19	change border image	18	21
20	table tr td	27	19
21	progress bar	17	20
22	canvas (simple rectangle)	13	14
23	canvas (gradient font)	12	14
24	canvas (gradient font)	14	23
25	canvas (repeat pattern)	16	19
26	camera	23	24
	total blanks (average)	403 (15.5)	530 (20.4)

B. EFP Instance Difference between Manual and Algorithm

As an illustrative example, Figures 17 and 18 show the manually generated input text file and the algorithm generated one for the source code of ID=10 for *basic topics*, respectively. When they are compared, the algorithm generated instance has more blanks for tags and library elements than the manual one, because the algorithm will select all the possible elements for them. On the other hand, the manual selection may miss selecting several elements. Therefore, the effectiveness of the proposal is confirmed.

VII. Evaluation by New Instance Generations

Next, we evaluate the effectiveness of the algorithm for *web-client programming* study by newly generating 10 EFP instances using the algorithm, and assigning them to 40

Problem #6

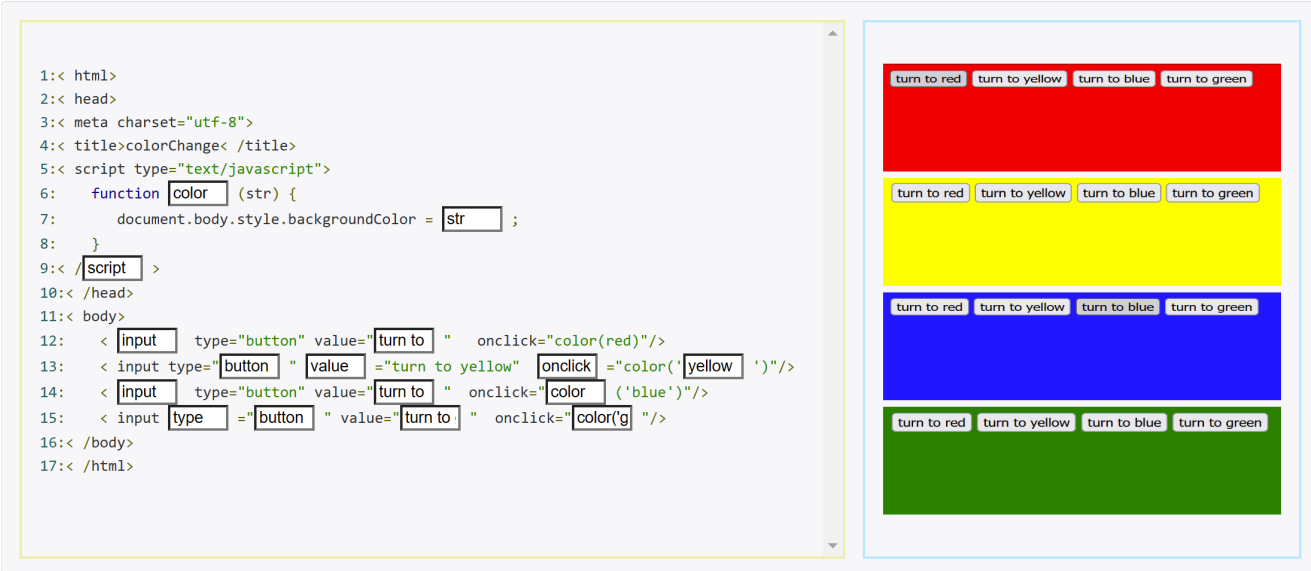


Fig. 16: Interface of ID=6 for *basic topics*.

```
1  < html>
2  < head>
3    < title>easy< /title>
4  < /head>
5  < body>
6    < p> _1_ < /p>
7    < _2_   onclick=" _3_ "> _4_ < /button>
8    < p id="demo">0< /p>
9    < _5_ >
10     var counter = 0;
11   _6_  add() {
12     return  _7_  += 1;
13   }
14   function myFunction(){
15     _8_ .getElementById(" _9_ ").innerHTML =
           _10_ ();
16   }
17    < / _11_ >
18 < /body>
19 < /html>
20
21 Answer:
22 Global variable count,,button,,myFunction(),,
23 Count,,script,,function,,counter,,document,,
24 demo,,add,,script,,
```

Fig. 17: Manual Blank Generation Output file

```
1  < html>
2  < head>
3    < title>easy< /title>
4  < /head>
5  < body>
6   < p> _1_ < _2_ >
7   < button onclick="myFunction()"> _3_ < _4_ >
8   < p  _5_ = _6_ > _7_ < /p>
9   < script>
10   _8_   _9_  = 0;
11   _10_   _11_  {
12     _12_  counter += 1;
13   }
14   function myFunction() {
15     _13_ . _14_ ("demo"). _15_  = add();
16   }
17  < _16_ >
18 < /body>
19 < /html>
20
21 Answer:
22 Global variable count,,/p,,Count,,/button,,id
23 ,,"demo",,0,,var,,counter,,function,,add(),,
24 return,,document,,getElementById,,innerHTML,,
25 /script,,
```

Fig. 18: Automatic Blank Generation Output file

students in Okayama University, Japan, and State Polytechnic of Malang, Indonesia, who have not studied *web-client programming* formally.

A. Generated EFP Instances

Tables III shows the topic, the total number of lines and lines of *JavaScript (JS)* in the source code, and the number of blanks for each of the 10 additional instances of EFP generated. As different topics from the previous instances, they cover image usages, text boxes, styles, iframes, and event objects.

TABLE III: New EFP instances generated by automatic blank generation system.

ID	topic	total # of lines	# of JS lines	# of blanks
1	show Id value	19	6	14
2	change image	19	10	11
3	img function(onload)	15	5	6
4	textbox function(onrest)	18	5	6
5	textbox function(onselect)	16	5	10
6	print webpage	16	5	6
7	create iFrame	19	7	10
8	captalize text	17	6	11
9	show coordinate	17	5	11
10	make element invisible	32	10	14
	total (average)	188(18.8)	64(6.4)	99 (9.9)

B. Solution Results of Individual Instances

First, we separately analyzed the solution results for the 10 EFP instances. Figure 19 displays the average correct rate for

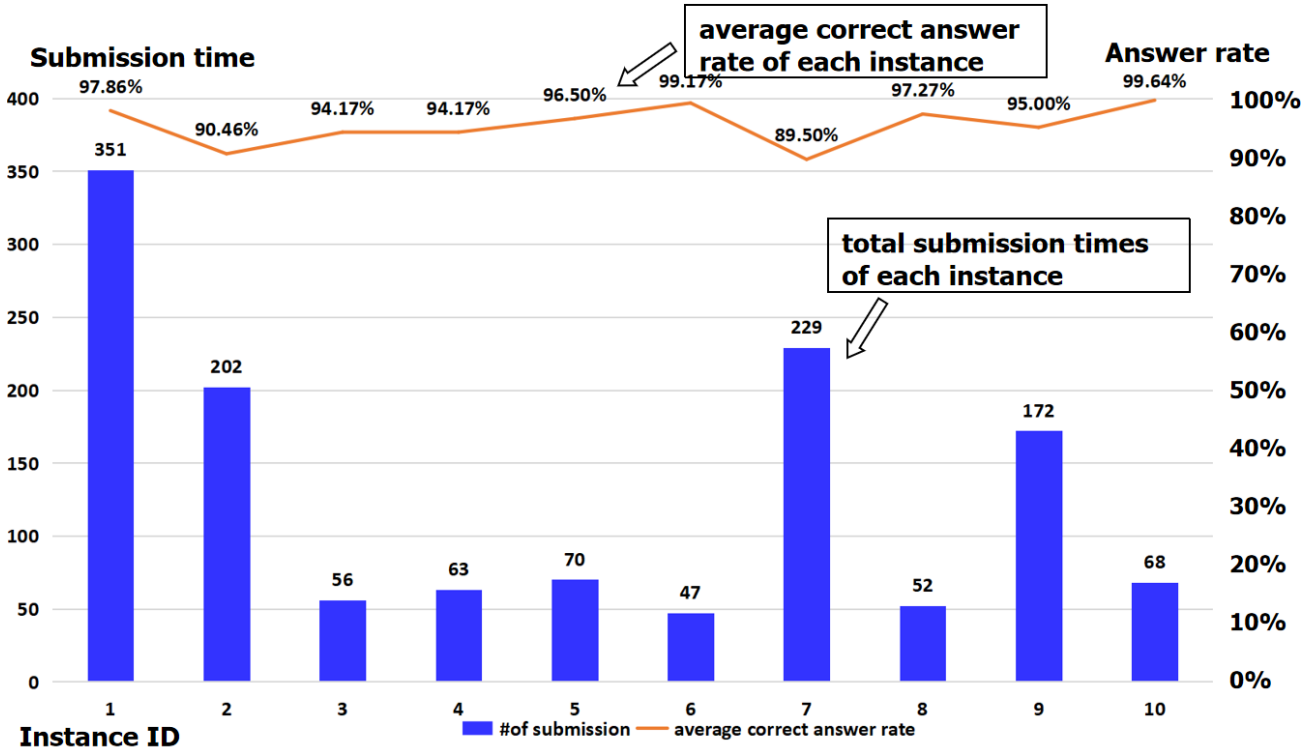


Fig. 19: Solution results for individual instances.

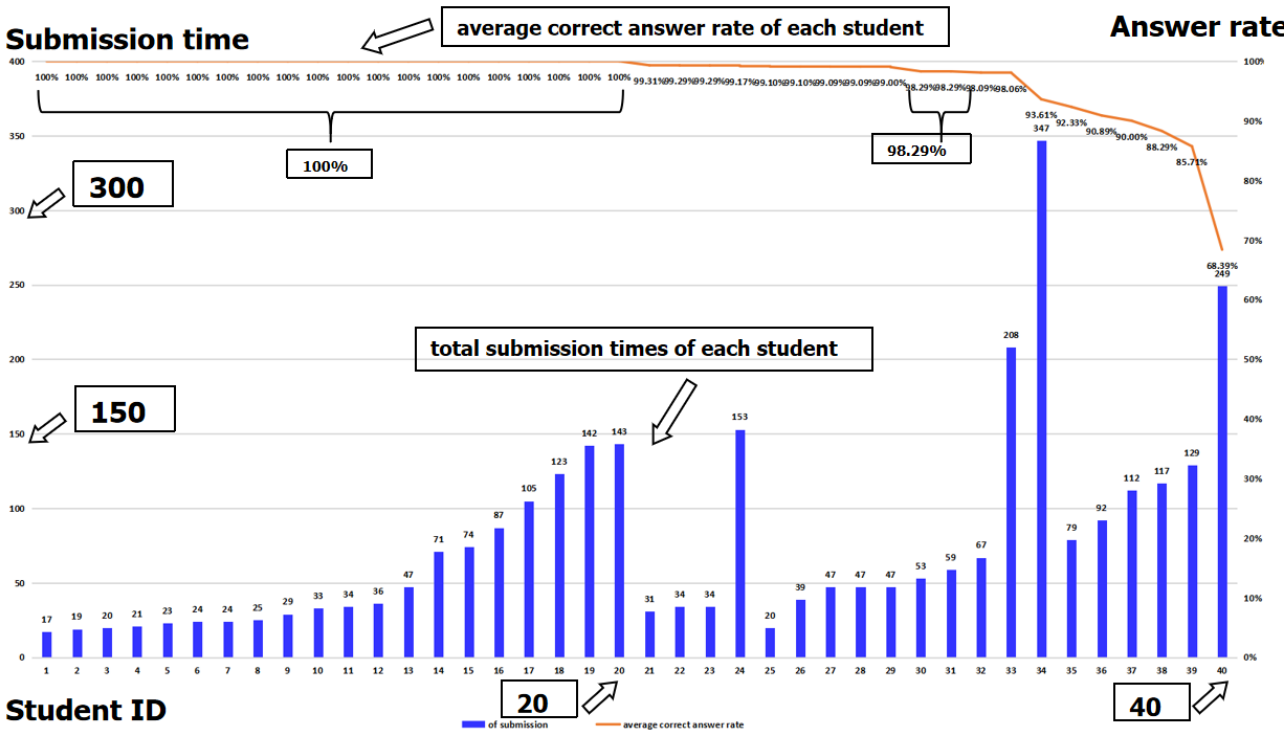


Fig. 20: Solution results for individual students.

each instance and the total number of 40 student submitted answers. The average correct rate for an instance is 95.37% and the average number of submissions is 131.00. The best correct rates are 99.17% and 99.64% for instances with ID=6 and 10, correspondingly.

On the other hand, the instances at ID=2 and 7 resulted in the worst rate 90.46% and 89.50% respectively. We will analyze the reasons to investigate how to improve them.

In the instance at ID=2, from `image = document.getElementById('myimage')` and `image.src.match("bulbon")`, it will be difficult for students to know that match `"bulbon"` in `src="/images/pic_bulboff.gif"` becomes true. In the instance at ID=7, the concept and use of *iframe* will be

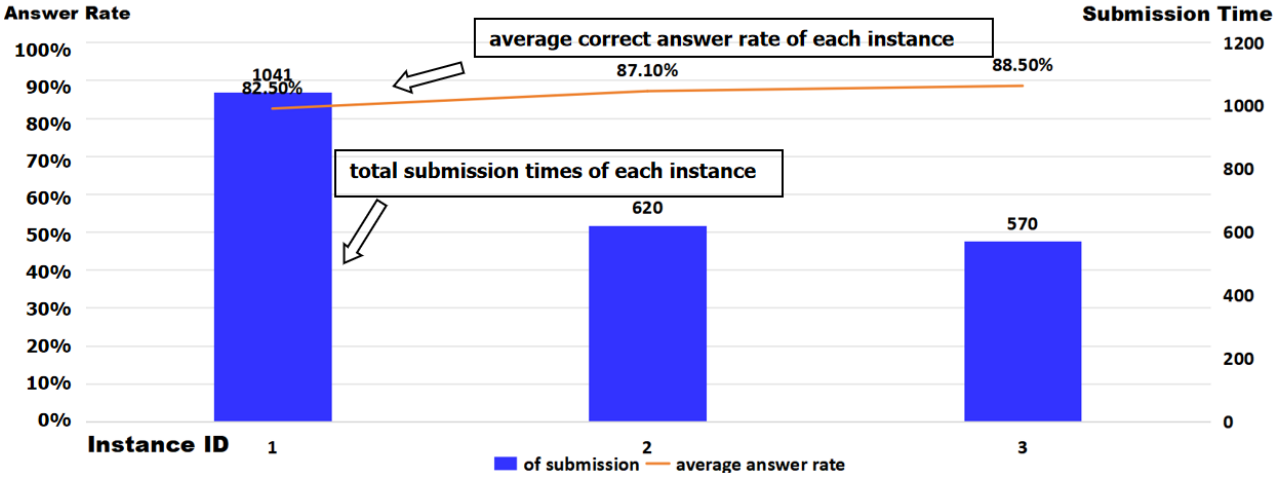


Fig. 21: Solution results for individual instances.

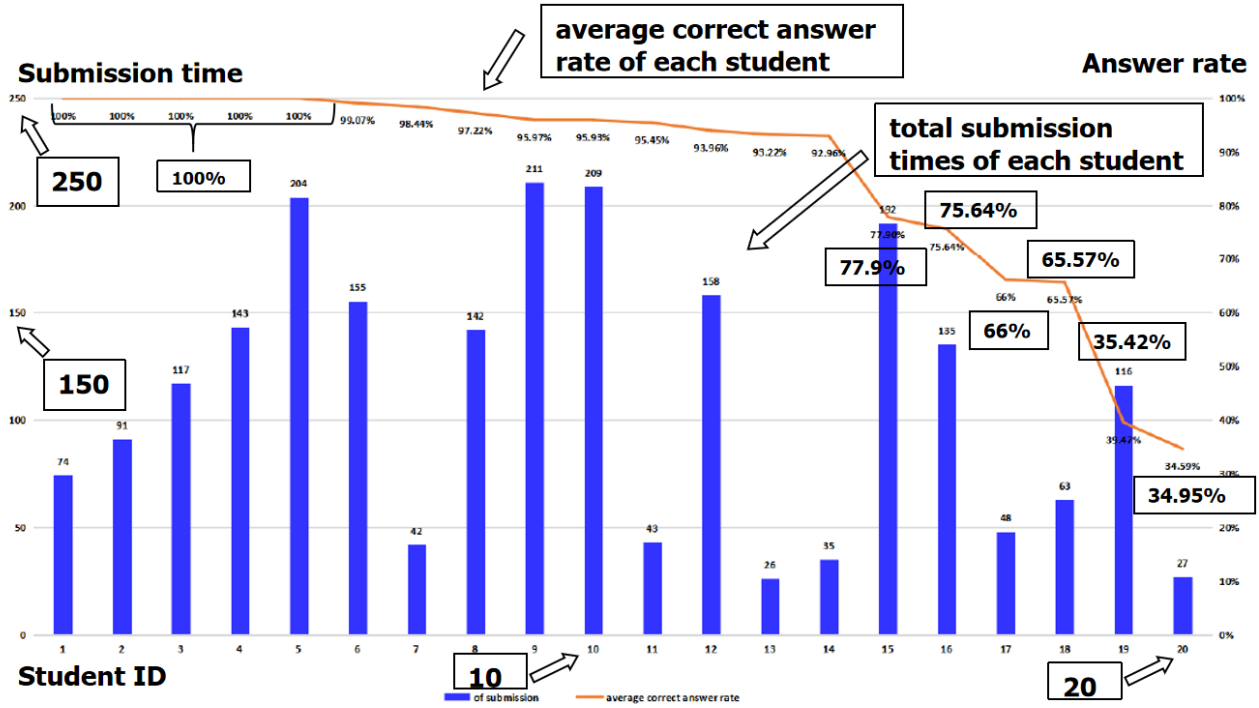


Fig. 22: Solution results for individual students.

difficult for students. Their proper guide documents can be helpful to understand them, which will be in future works.

C. Solution Results of Personal Students

Next, we analyze the solution results of the 20 students individually. Figure 20 shows the average correct answer rate and the total number of answer submissions among the 10 instances by each student.

This figure is in descending sequence by student performances. The average correct rate is 97.36% and the average number of submission is 75.35. There are twenty students (50%) who obtained a perfect score of 100%. The very best student answered all 10 instances and only submitted 17 times, meaning that only 7 were incorrect.

One student did reach the 70% rate, although he/she submitted answers 249 times. This student did not study programming well.

The both results suggest that the EFP instances generated by the algorithm are not difficult for students who have not studied *web-client programming* formally. By collecting source codes and applying the algorithm, teachers can easily generate new EFP instances. Thus, the effectiveness of the proposed algorithm for *web-client programming* study is confirmed.

VIII. Evaluation by Game-based Instance Generations

Next, we present three new EFP instances generated using the algorithm with source codes for simple video games on a web browser. These game-based instances will add more fun elements to learning experiences and enrich the evaluations of the algorithms. This approach may introduce a novel and

TABLE IV: Game-based EFP instances generated by proposal.

ID	topic	total # of lines	# of JS lines	# of blanks
1	Guess Number	43	18	36
2	Guess Word	60	50	53
3	Snake Game	107	86	60
	total (average)	210(70.0)	154(51.3)	149(49.7)

engaging way for students to interact and learn with these instances. By incorporating *gamification* elements into the learning process, students will find the EFP instances more interesting.

A. Generated EFP Instances

Table IV shows the topic, the total number of lines and the number of *JavaScript (JS)* lines in the source code, and the number of blanks for each of the three game-based instances.

B. Solution Results of Individual Instances

Figure 21 displays the average correct answer rate and the total number of answer submissions for each instance by 40 students. The average rate is 86.03% and the average number of submissions is 554. The best correct rates are 88.50% for the instance with ID=3.

In the "Guess the Number" game with ID=1, students need to guess one number within 1-10. If the number is small, the page prompts "Try a bigger number." If the number is larger, the page prompts "Try a smaller number." Although the source code in this instance is relatively short compared with the others, it still contains 36 blanks. This means that students need to fill out most of the whole code, by understanding the game logic, processing the user inputs, and generating the responses. Understanding and implementing the game logic can be a significant barrier for students, especially, for novice students of *web-client programming*.

In the "Guess the Word" game with ID=2, the source code involves more complex *JavaScript* codebase. In this instance, students need to fill in 53 blanks, by understanding the rules of the game, managing the user inputs and feedback. The source code involves more complex logic than the previous instance. This game requires a player to guess one letter at a time to reveal the hidden words. The complexity is greater because it needs to check the guessed letter and update the displayed word, by using arrays to select words and keep track of the guessed letters. Debugging this type of the interactive code will be complex for students.

In the "Snake Game" with ID=3, this EFP instance is the most challenging one among the three. It has 60 blanks and 86 lines. Students need to understand the rules of the game, including how the snake moves, how the snake grows when it eats food, and how the game ends when the snake hits a wall or itself. Understanding these game mechanics can be challenging for learners, especially, for students new to game developments.

C. Solution Results of Personal Students

Next, we analyze the solution results of the 20 students individually. Figure 22 shows the average correct answer rate and the total number of answer submissions among the three instances by each student. It is noted that the students are sorted in descending of their performances. The average rate is 95.37% and the average number of submissions is 65.50. Five students (25%) solved all of them correctly, where the least number of submissions is 74.

Unfortunately, six students (30%) among 20 did reach the 75% correct rate. It suggests that a significant portion of the students struggled with the game-based EFP instances that are more complex and challenging than previous ones, and may not reach sufficient programming levels. While complexity can facilitate deeper understanding of the problems, overly difficulty may frustrate and discourage students to solve them. Therefore, it is necessary to provide proper hints for these instances, which will be in future works.

D. Evaluation by Analyzing Instance Characteristics

Furthermore, we provide a comprehensive assessment of the EFP instances generated by the algorithm, focusing on the relationships between the number of lines in the source code, the number of blanks, the submission times and the correct answer rates by the students.

In our previous studies of EFP for *Java programming*, longer source codes typically caused greater difficulties for students. We found the similar results for *web-client programming*. Table V shows the relationships between the number of lines in the source code, the number of blanks, the submission times and the correct answer rates by the students for the 13 EFP instances generated by the algorithm.

In the first 10 EFP instances, each instance has up to 32 code lines and up to 14 blanks. The higher performances of students suggest that they cleared basic educational goals and understood introductory programming concepts.

In contrast, in the remaining three instances, each instance has up to 107 lines and up to 60 blanks. This long code length and high number of blanks lowered the student performances. The code length and the number of blanks can be the key determinants of the instance difficulty affecting the student performance.

Here, we found that the correct rate is higher and the number of submissions is lower among the last three instances, although the code length and the number of blanks are both increased. The reason will come from the fact that source codes in the last two instances have repetitions of certain parameters or patterns, which can make it easier for students to recognize the missing elements. For example, certain variables functions, or elements that have similar names or attributes appear multiple times. This repetition can help to identify the expected elements in the code.

Besides, the source codes in them were designed with interactivity in mind. *JavaScript*, *HTML*, and *CSS* enable dynamic and engaging user interfaces. Some patterns will emerge in them for users to interact with interfaces by clicking buttons, entering data, or triggering animations, which is a part of the game logic and will influence students in solving the problems.

IX. Evaluation by Correlation Coefficients Between Instance Characteristics

In this section, we incorporate another evaluation method using *correlation coefficient* to further enhance the depth and accuracy of our study in this paper.

TABLE V: Analysis of instance characteristics.

ID	topic	total # of lines	# of blanks	# of submission time	# answer rate
1	show Id value	19	14	351	97.86%
2	change image	19	11	202	90.46%
3	img function (onload)	15	6	56	94.27%
4	textbox function (onrest)	18	6	63	94.17%
5	textbox function (onselect)	16	10	70	96.50%
6	print webpage	16	6	47	99.17%
7	create iFrame	19	10	229	89.50%
8	captalize text	17	11	52	97.27%
9	show coordinate	17	11	172	95.00%
10	make element invisible	32	14	68	99.64%
11	Guess Number	43	36	2042	82.50%
12	Guess Word	60	53	620	87.10%
13	Snake Game	107	60	579	88.50%
	total (average)	398(30.6)	248(19.1)	3550(273.1)	(93.2%)

A. Analysis Variables

Table V shows the instance characteristics as the variables for calculating *correlation coefficient* using *Python* and its three libraries.

B. Analysis Process

For this analysis, we utilized *Python* and its statistical analysis libraries to calculate *correlation coefficients*, and generated scatter plots and trend-lines. This approach allows us to automate the computational process, ensuring the accurate and efficient assessment of the impact of the *blank element selection algorithm*.

To initiate the analysis, we adopted the *pandas* library for data processing and the *seaborn* and *matplotlib* libraries for data visualizations. The obtained scatter plot with trend-lines can visualize the relationship between two variables, which can add the insight of them by showing the overall trend of data points. *Correlation coefficient* calculated by the *pandas* library quantify the direction of the linear relationship between the variables.

The process of calculating the *correlation coefficient* and generating the scatter plot with the trend-line in a *Python* program is given as follows:

1) Import libraries: We import the *pandas* library for data manipulations by `import pandas as pd`, the *seaborn* library for statistical data visualizations by `import seaborn as sns`, and the *pyplot* module from the *matplotlib* library for creating plots by `import matplotlib.pyplot as plt`.
2) Load data: We read data from the *Excel* file containing the variables in Table V into the *pandas DataFrame* `df`.
3) Extract relevant columns: We extract selected two columns as the parameters (variables) from `df`.
4) Create scatter plot with trend-line: We use `sns.regplot` to create the scatter plot with the regression line (trend-line) fitted to the data points.
5) Calculate correlation coefficient: We use the *corr* function in the *pandas* library to calculate the *correlation coefficient* between the two parameters.
6) Display correlation coefficient: We display the calculated *correlation coefficient* on the plot.
7) Add labels and title: We add the labels to the x and y axes, and the plot title.
8) Display plots: Finally, we display the generated scatter plot with the trend-line and the *correlation coefficient*.

C. Correlation Analysis with Instance Characteristics

In order to provide insight into the relationship between instance difficulty and student answer correctness, including variables such as the number of lines of code, the number of blanks in the instances, and the time taken by students to submit, we conducted a comprehensive analysis using *correlation coefficients*.

1) Number of Blanks and Number of Code Lines: Regarding to the relationship between the number of blanks and the number of code lines in Figure 23a, the *correlation coefficient* is 0.941, which suggests the *strong positive* correlation. This implies that as the number of code lines increases, the number of blanks increases linearly.

2) Number of Submission Times and Number of Blanks: Regarding to the relationship between the number of submission times and the number of blanks in Figure 23b, the *correlation coefficient* is 0.579, which suggests the *positive* correlation. Instances with more blanks require more submission attempts.

3) Number of Submission Times and Number of Code Lines: Regarding to the relationship between the number of submission times and the number of code lines in Figure 23c, the *correlation coefficient* is 0.421, which suggests the *positive* correlation. Instances with more code lines require more submission attempts.

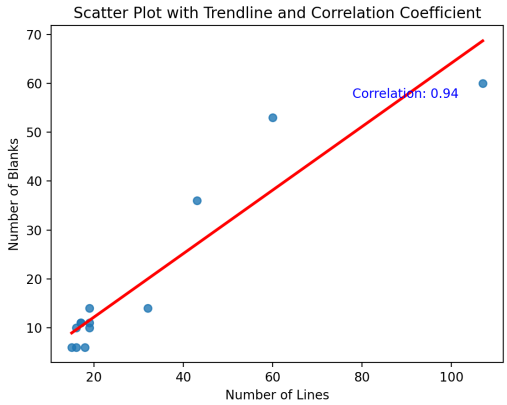
4) Correct Answer Rate and Number of Code Lines: Regarding to the relationship between the correct answer rate and the number of code lines in Figure 23d, the *correlation coefficient* is -0.533, which suggests the *negative* correlation. Instances with more code lines are resulted in lower correct answer rates.

5) Correct Answer Rate and Number of Blanks: Regarding to the relationship between the correct answer rate and the number of blanks in Figure 23e, the *correlation coefficient* is -0.665, which suggests the *negative* correlation. Instances with more code blanks are resulted in lower correct answer rates.

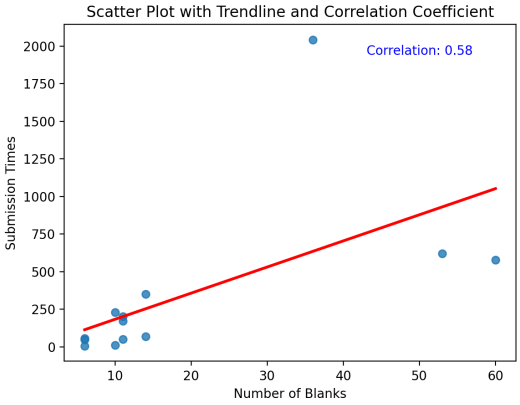
6) Number of Submission Times and Correct Answer Rate: Regarding to the relationship between the number of submission times and the correct answer rate in Figure 23f, the *correlation coefficient* is -0.781, which suggests the *strong negative* correlation. Instances with lower correct answer rates usually require more submission attempts.

X. Conclusion

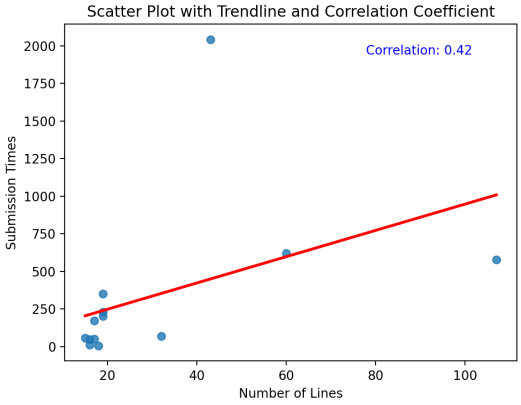
This paper presented the *blank element selection algorithm* to generate a new EFP instance from a given source



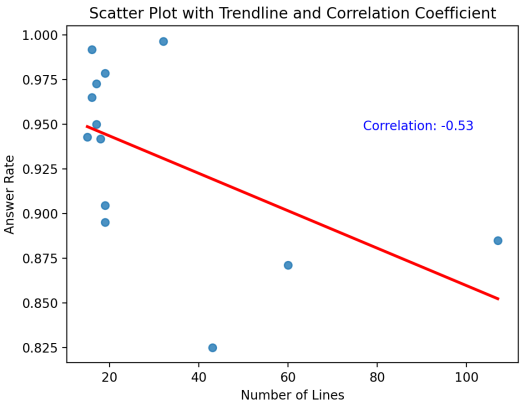
(a) Correlation Coefficient between Number of Blanks and Code Lines



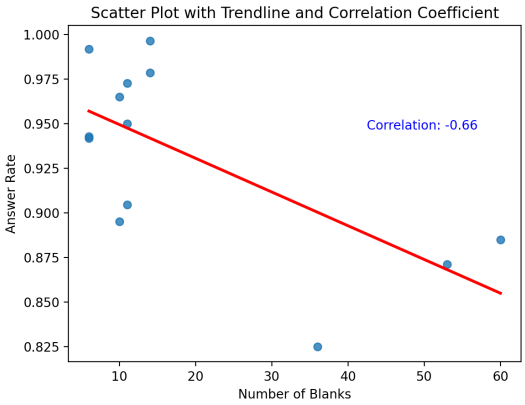
(b) Correlation Coefficient between Submission Times and Number of Blanks



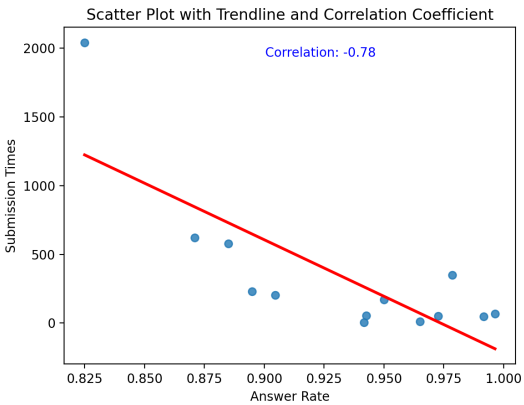
(c) Correlation Coefficient between Submission Times and Number of Code Lines



(d) Correlation Coefficient between Answer Rate and Number of Code Lines



(e) Correlation Coefficient between Answer Rate and Number of Blanks



(f) Correlation Coefficient between Submission Times and Answer Rate

Fig. 23: Scatter Plots with Trendlines for Instance Characteristics.

code for *client-side web programming*. Seven rules were defined for blank element selections from a source code, and the algorithm procedure was implemented in *Python* using *BeautifulSoup* and *regular expressions*. For evaluations, the proposed algorithm was first applied to the 47 source codes that were used for manual generations, where the better instances with more blanks were obtained. Thus, the validity of the algorithm was confirmed. Then, 10 new basic EFP instances were generated by the algorithm and assigned to 40 students in two universities who have not studied *web-client programming* formally. The solution results confirm the effectiveness. Besides, three game-based EFP instances were generated by the algorithm and assigned to 20 students, to diversify the assessment by introducing complex and interactive instances. The results demonstrate the versatility of the algorithm. Furthermore, the relationships between the number of code lines, the number of generated blanks, the submission times and the correct answer rates by the students were examined. The results indicate that EFP instances with longer codes and more blanks tended to lead to lower correct rates and higher submission times. In future works, we will make guide documents for hard concepts in *web-client programming*, improve the algorithm implementation to properly select blanks for the same word that appears several times in

the source code by considering the importance or difficulty as a blank, continue to use the algorithm to generate EFP instances for other topics, and assign them to students for validity verifications.

ACKNOWLEDGMENT

The authors would like to thank to the students in Okayama University and State Polytechnic of Malang to answer the problems and give us comments. They are inevitable to complete this paper.

REFERENCES

[1] What is JavaScript, https://developer.mozilla.org/en-US/docs/Learn/ JavaScript/First_steps/What_is_JavaScript.

[2] S. T. Aung, N. Funabiki, Y. W. Syaifudin, H. H. S. Kyaw, S. L. Aung, N. K. Dim, and W. C. Kao, "A proposal of grammar-concept understanding problem in Java programming learning assistant system," J. Adv. Inform. Tech, vol. 12, no. 4, pp1-10, 2021.

[3] 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," Inf. Eng. Express, vol. 1, no. 3, pp9-18, 2015.

[4] S. H. M. Shwe, N. Funabiki, Y. W. Syaifudin, E. E. Htet, H. H. S. Kyaw, P. P. Tar, N. W. Min, T. Myint, H. A. Thant, and W. C. Kao, "Value trace problems with assisting references for Python programming self-study," Int. J. Web Inform. Syst, vol. 5, no. 2, pp100-110, 2021.

[5] X. Lu, N. Funabiki, H. H. S. Kyaw, E. E. Htet, S. L. Aung, and N. K. Dim, "Value trace problems for code reading study in C programming," Adv. Sci. Tech. Eng. Syst. J. (ASTESJ), vol. 7, no. 1, pp14-26, 2022.

[6] K. H. Wai, N. Funabiki, K. T. Mon, S. H. M. Shwe, H. H. S. Kyaw, and K. S. Lin, "A proposal of code modification problem for web client programming using JavaScript," Proc. CANDAR, pp196-202, 2021.

[7] N. Funabiki, Tana, K. K. Zaw, N. Ishihara, and W. C. Kao, "A graph-based blank element selection algorithm for fill-in-blank problems in Java programming learning assistant system," IAENG Int. J. Comput. Sci., vol. 44, no. 2, pp247-260, 2017.

[8] H. H. S. Kyaw, N. Funabiki, S. L. Aung, N. K. Dim, and W. C. Kao, "A study of element fill-in-blank problems for C programming learning assistant system," Int. J. Inform. Edu. Tech., vol. 11, no. 6, pp255-261, 2021.

[9] H. H. S. Kyaw, S. S. Wint, N. Funabiki, and W. C. Kao, "A code completion problem in Java programming learning assistant system," IAENG Int. J. Comput. Sci., vol.47, no. 3, pp350-359, 2020.

[10] N. Funabiki, Y. Matsushima, T. Nakanishi, and N. Amano, "A Java programming learning assistant system using test-driven development method," IAENG Int. J. Comput. Sci., vol. 40, no.1, pp38-46, 2013.

[11] H. Qi, N. Funabiki, K. H. Wai, X. Lu, H. H. S. Kyaw, W. C. Kao, "An implementation of element fill-in-blank problems for code understanding study of JavaScript-based web-client programming," International Journal of Information and Education Technology (IJIET), vol. 12, no. 11, pp1179-1184, 2022.

[12] H. Qi, N. Funabiki, K. H. Wai, M. Z. Htun, K. T. Mon, and W. C. Kao, "A study of element fill-in-blank problems for applicative grammar topics in JavaScript-based web-client programming," IPSJ SIG Tech. Rep, vol. 2022-CE-166, no.2, 2022.

[13] H. Qi, N. Funabiki, K. H. Wai, M. Z. Htun, V. Flasma, and Y. W. Syaifudin, "A study of blank element selection rules for element fill-in-blank problem in JavaScript-based web-client programming," Proc. ICEMSIT, 2022.

[14] BeautifulSoup in Python, https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[15] Regular Expression in Python, https://docs.python.org/3/library/re.html.

[16] E. Lahtinen, K. Ala-Mutka, and H. M. Järvinen, "A study of the difficulties of novice programmers," Proc. ITiCSE, vol. 3, no. 2, pp14-18, 2005.

[17] A. Knutas, D. Savchenko, T. Hynninen, and N. Grönberg, "Constructive alignment of web programming assignments and automated assessment with unit testing," Proc. Koli Calling, 2019.

[18] S. Kar, M. M. Islam, and M. Rahaman, "State-of-the-art reformation of web programming course curriculum in digital bangladesh," Int. J. Adv. Comp. Sci. Appl, vol. 11, no. 3, pp193-201, 2020.

[19] I. W. Arawjo, C. Y. Myers, A. C. Andersen, E. Guimbretière, "Teaching programming with gamified semantics," Proc. of the 2017 CHI conference on human factors in computing systems, pp4911-4923, 2017.

[20] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in Python," Proc. International Symposium on Software Testing and Analysis, pp282-293, 2016.

[21] E. Larson, "Automatic checking of regular expressions," International Working Conference on Source Code Analysis and Manipulation, pp225-234, 2018.

[22] I. Onyenwe, O. Ebele Gr, C. A. Nwafor, O. Agbata, "Developing products update-alert system for e-commerce websites users using HTML data and web scraping technique," arXiv preprint arXiv:2109.00656, 2021.

[23] S. Thivaharan, G. Srivatsun, S. Sarathambekai, "A survey on python libraries used for social media content scraping," Proc. International Conference on Smart Electronics and Communication, pp361-366, 2020.

[24] I. Darmawan, M. Maulana, R. Gunawan, N. Widiyasono, "Evaluating web scraping performance using XPath, CSS Selector, regular expression, and HTML DOM with multiprocessing technical applications," JOIV, vol. 6, no. 4, pp904-910, 2022.

[25] M. Khder, "Web scraping or web crawling: state of art, techniques, approaches and application," International Journal of Advances in Soft Computing and Its Applications, vol. 13, no. 3, pp145-168, 2021.

[26] Fujita, M. S. L., Katahira, I., Tolare, J. B, "Institutional Repository Keyword Analysis with Web Crawler," Central European Journal of Educational Research, vol. 4, no. 2, pp54-59, 2022.

[27] Singrodia, Vidhi, A. Mitra, S. Paul, "A review on web scrapping and its applications," ICCCI, 2019.

[28] Andersson, Pontus, "Developing a Python based web scraper: A study on the development of a web scraper for TimeEdit," 2021.

[29] Oprea, S. Vasilica, A. Bâra, "Why Is More Efficient to Combine BeautifulSoup and Selenium in Scraping For Data Under Energy Crisis," Ovidius University Annals, Economic Sciences Series, vol. 22, no. 2, pp146-152, 2022.

[30] Sahoo, K., Samal, A. K., Pramanik, J., and Pani, S. K., "Exploratory data analysis using Python," International Journal of Innovative Technology and Exploring Engineering, vol. 8, no. 12, pp4727-4735, 2019.

[31] Gupta, B., Negi, M., Vishwakarma, K., Rawat, G., Badhani, P., and Tech, B., "Study of Twitter sentiment analysis using machine learning algorithms on Python," International Journal of Computer Applications, vol. 165, no. 9, pp29-34, 2017.

[32] Sial, A. H., Rashdi, S. Y. S., and Khan, A. H., "Comparative analysis of data visualization libraries Matplotlib and Seaborn in Python," International Journal, vol. 10, no. 1, 2021.

[33] Stančin, I., and Jović, A., "An overview and comparison of free Python libraries for data mining and big data analysis," IEEE, pp977-982, 2019.