A Judge Method for Programming with Graphical User Interface

Yu-Ju Liao[a], Chang-Biau Yang[a] and Kuo-Tsung Tseng[b] *

[a]Department of Computer Science and Engineering
National Sun Yat-sen University, Kaohsiung, Taiwan
[b]Department of Shipping and Transportation Management
National Kaohsiung University of Science and Technology, Kaohsiung, Taiwan

Abstract

MIT App Inventor 2 (AI2) is a web-based development environment for Android and iOS applications (Apps). Its graphical user interface (GUI) makes it easily be used, but the graphical components lead to the difficulty of automatic judging. This paper tries to design a method for automatically judging a given screenshot produced by an AI2 program or other GUI programming tools. For given a template screenshot and a testing screenshot, we first detect the components in the two screenshots and compare the similarity between these components. Then, we calculate the component layout similarity of the two screenshots with the 2D LCS (2-dimensional largest common substructure) algorithm. The experimental results show that the screenshots of the application can be detected by the technology of computer vision, and hence we can judge the similarity between two application programs of GUI.

Keywords: graphical user interface, MIT App Inventor 2, component detection, text recognition, 2-dimensional largest common substructure

1 Introduction

With the widespread utilization of information and communication technology in our lives, logical thinking has become one of the fundamental abilities that students should have. Educational institutions are also launching more and more courses to help students get in touch with and become familiar with computer skills as early as possible. Many tools are used to train students for their logical thinking. Among them, the MIT App Inventor 2 (AI2) [5] is popular tool with the graphical user interface (GUI) and a simple programming language for developing Android and iOS Apps. It was originally developed by Google and it is currently maintained and managed by the Massachusetts Institute of Technology (MIT).

The visual features of AI2 allow students to get started easily, but it also increases the difficulty of automatic judge. The CodeMaster [1, 22] can judge the components and their characteristics used in the project by analyzing the AI2 source code. The CodeMaster first decompresses the `.aia` files and then converts the result into a string. Next, the string is split according to the grammar to analyze the GUI components and dynamic events used in the project. Finally, based on these information, the submitted cases are scored for the complexity and layouts [22]. There were some studies based on the results analyzed by the CodeMaster to evaluate the applications [14] or to discuss the accuracy of the assessment [13].

However, among the current App Inventor's judge methods [22], there is a lack of methods that can judge based on a single standard answer. That is to give a standard answer, and then to ask students to do the same or something similar.

In this paper, we would like to present a novel method to analyze the App screenshot by computer vision techniques, and then to determine whether the screenshot meets the requirements. Our method is divided into two stages for comparing the similarity of two screenshots. The first stage is to detect the components in both screenshots, and the second stage compares the overall layouts of the two screenshots to judge whether the two screenshots are similar. Due to lack of generic datasets at present, we generate screen-

*Corresponding author. E-mail: cbyang@cse.nsysu.edu.tw (Chang-Biau Yang).

Table 1: Related researches of the judge method for App Inventor programs.

Year	Author	Keyword
2015	Maiorana *et al.* [17]	Blockly
2018	Wangenheim *et al.* [22]	CodeMaster
2018	Li *et al.* [18]	TF-IDF
2019	Solecki *et al.* [14]	UI design

shots of AI2 projects by capturing the appearance of components in AI2 and randomly produce some texts on these components.

In the first stage, we train the YOLOv4 model [8] to recognize components, and the recognition rate can reach 100% under the condition of IoU less than 0.6. Moreover, we use the Keras-OCR [4] with the template matching to achieve a text recognition rate of 98% under certain conditions. In the second stage, we invoke the algorithm for 2D LCS [10] to determine the similarity of the component layouts in the two screenshots. The similarity scoring can be flexibly adjusted according to needs, since the 2D LCS algorithm can adopt various similarity definitions.

The structure of this paper is given as follows. In Section 2, we will introduce some knowledge related to judging methods for GUI programming. In Section 3, we will give definitions of our judge rule and our method. Section 4 presents our experimental results. In Section 5, we will give the conclusion and discuss some possible future works.

2 Preliminaries

2.1 Judge Methods for App Inventor Programs

MIT App Inventor 2 (AI2) [5] is a web-based platform for developing the Android and iOS applications. Table 1 lists the researches about the judge methods for AI2 programs. CodeMaster [1, 22] asks the user to upload the original files of the App Inventor project, and it then analyzes the program in the files. The program is divided into several segments (strings) according to the grammar, and statistics are made for each segment that appears. Finally, according to these statistics, each grading segment is scored. Their grading segments have items such as Logic, Loop, and Screen.

Chang *et al.* proposed an algorithm to calculate the similarity of component layouts in two images

Figure 1: An example for GUI component recognition.

[11]. Their method first converts the component positions to a 2D string, then the similarity is got from the 2D string comparison.

2.2 GUI Component Recognition

The *GUI component recognition* (UI element detection) [24], proposed by Xie *et al.* in 2020, uses computer vision as a tool to analyze the screenshot to detect the positions of GUI components. As shown in Figure 1, the main purpose of GUI component recognition is to determine the positions of components (objects) in the screen, or even to determine the category of each component. Detecting GUI elements or components can be roughly divided into two stages, one is the non-text component recognition and the other is the text-component recognition [24]. The UIED method is opposite to the order of Canny edge detection [25] by merging large components from small edges.

In the stage of text component recognition, some tools can detect the part of the text, such as the pre-trained model EAST [26]. Another commonly used method is the *optical character recognition* (OCR) technology. There are also many tools that convert text into speech by reading images [16, 21], such as the OCR model provided and maintained by Google. Hence, people can obtain the position and context of the text by upload the images to Google platform [2]. But it requires sending the image to the cloud server for processing. In some situations, we prefer methods that can be used without a network. Thus, in this paper, we choose the Tesseract OCR [21] and Keras-OCR [4] for text processing.

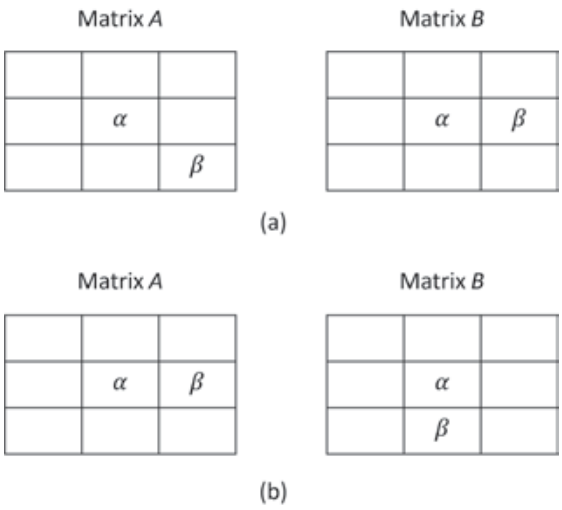

Figure 2: Examples of the P(ENL) problem. (a) The corner relation. (b) The side relation.

2.3 The Two-Dimensional Largest Common Substructure Problem

Amir *et al.* [7] defined the problem of *two-dimensional largest common substructure* (2D LCS). The largest common substructure of two sets of 2D-index elements is defined as that after deleting zero or more elements in the two sets, the relative positions of every two elements in the two remaining subsets meet some specific position similarity requirements and the number of the remaining 2D-index elements is maximal.

In 2020, a general definitions of the 2D LCS was given by Chan *et al.* [10], and they proposed a solution by using the integer linear programming (ILP) [15, 19, 20] to solve the 2D LCS problem [10, 12]. They defined several different criteria and demonstrated that four of them are legitimate definitions with symmetry, called by P(ENL), P(ENE), P(LOL), and P(LOE). They also proved that these four problems are all NP-hard.

We present the definition of P(ENL) [10] here. An example is illustrated in Figure 2.

Definition 1. (P(ENL) matching rules in 2D LCS) [10] *Suppose both 2D matrixes A and B have two components α and β. It is assumed that $A[i_1, j_1]$ matches $B[h_1, k_1]$ for α and $A[i_2, j_2]$ matches $B[h_2, k_2]$ for β. The matching rules for P(ENL) are given as follows.*
$i_1 < i_2, j_1 < j_2 \ \rightarrow \ h_1 \leq h_2$ *and* $k_1 \leq k_2$
$i_1 < i_2, j_1 > j_2 \ \rightarrow \ h_1 \leq h_2$ *and* $h_1 \geq h_2$

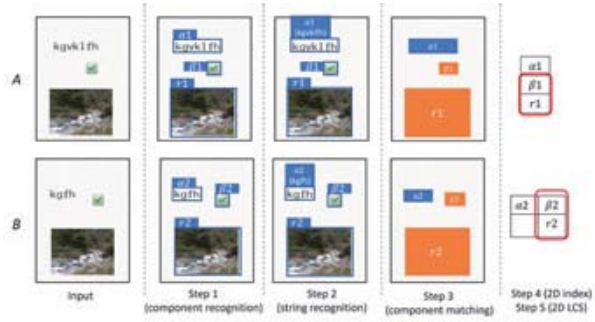

Figure 3: An illustration for judging MIT App Inventor 2 Programs.

$i_1 < i_2, j_1 = j_2 \ \rightarrow \ h_1 < h_2$
$i_1 = i_2, j_1 < j_2 \ \rightarrow \ h_1 < h_2$

3 Our Judge Method

3.1 The Problem Definition

Definition 2. (similarity of two components) *If the similarity of two components is higher than a threshold, they are considered as two identical components (matching components). The* similarity *of two components is divided into two parts. The first part is the appearance attributes, including shape, color and size. The second part is the similarity of the text strings within the two components.*

Definition 3. (Judgement of GUI programs, layout similarity) *The judgement of GUI programs is achieved by measuring the layout similarity of the template image and the testing image. The* layout similarity *of two images is the maximal subsets of components in both images that meet the relative position relationship between each pair of matching components in the two subsets.*

For identifying the matching components in both images, we can use a set of 2D indices to represent relative positions of these components. That is, we transform an exact coordinate into a 2D index. So it can be regarded as a *two-dimensional common substructure* (2D LCS) problem [10].

3.2 The Main Judgement Method for GUI Programs

Figure 3 shows an example of our judging method, described as follows.

Algorithm: The judgement of a graphical user interface (GUI) program.

Input: Two screenshot images of GUI programs.

Output: The maximal subset of matching components in both images.

Step 1 (Components recognition): Find out all components in both image and locate the coordinates of these components by using the YOLOv4 [8], proposed by Bochkovskiy *et al.* We classify common basic components according to their characteristics, such as `button` or `string`. The components that do not belong to any category are categorized as other components.

Step 2 (GUI text recognition): Use the Keras-OCR package [4] to retrieve the texts within the components. In addition, we increase the recognition accuracy with the template matching method (described later).

Step 3 (Component matching): Find out the corresponding (matching) components in the two images. The similarity of two components are measured by the component appearance similarity (including shape, size and color) and the text similarity.

Step 4 (2D index transformation): The component position reported by YOLOv4 is its upper-left and lower-right coordinates. We discrete the coordinates of each component to a simple 2D index, which can easily represent the relative positions among all components.

Step 5 (Two-dimensional largest common substructure (2D LCS) [10]): Since each component has its own 2D index, we can apply the 2D LCS algorithm [10] to measuring the layout similarity of two images. Chan also provided some sample programs to implement the 2D LCS algorithm by the *integer linear programming* (ILP) [9], using the *Gurobi optimization* [3].

3.3 GUI Text Recognition

After identifying the components in the image, we use the *optical character recognition* (OCR) technology to recognize the text within each component. The test contains only English letters. Here, we use two methods of OCR, Keras-OCR [4] and Tesseract v5 [21], for text recognition.

Since the accuracy of text recognition is still not high enough, we try to improve it after the recognition obtained by Keras-OCR. Only a small portion of the string is wrong, most of the mistakes are from incorrect recognition. The more common mistakes occur in recognizing `q` as `g`, recognizing `l` as `i`, or extra letters recognition, such as `l` and `i`. We use the template matching method for correcting the mistakes, described as follows.

Algorithm: Text correction with the template matching method.

Input: A set T of template English letters, a text component image I (containing only English letters) and its string S detected by Keras-OCR.

Output: The corrected text of S.

Step 1 : Calculate the number n of characters contained in I by the fixed width of each letter. If $n = |S|$, then $|S|$ is considered to be correct, and go to Step 3.

Step 2 : If there is `l` (Lima) in S, but there is no `l` in I detected by the template matching, delete this `l` (Lima). If there is `t` in S, but there is no `t` in I detected by the template matching, delete this `t`.

Step 3 : If there is `1` (one) in S, replace `1` (one) with `l` (Lima). If there is `0` (zero) in S, replace `0` (zero) with `o`.

Step 4 : If there is `g` (or `a`) in S, but at this position, `q` in I is obtained by the template matching, then replace `g` (or `a`) with `q`. If the detected letter by the template matching is neither `g` (or `a`) nor `q`, and $|S|$ is too large, then delete this `g` (or `a`). If the detected by the template matching is neither `g` (or `a`) nor `q`, but $|S|$ is the correct length, then keep this as `g` (or `a`).

Step 5 : If there is `i` in S, but at this position, `l`, `f` or `j` in I is obtained by the template matching, then replace this `i` by `l`, `f` or `j`, respectively. If no `i`, `l`, `f` or `j` in I is obtained by the template matching, then delete this `i`.

Step 6 : The algorithm ends.

3.4 2D Index Transformation

In order to apply the 2D LCS algorithm [10, 12], we need to transform the coordinates of components to a set of 2D indexes. We transform the horizontal and vertical directions independently. We propose two cutting methods. The first is the

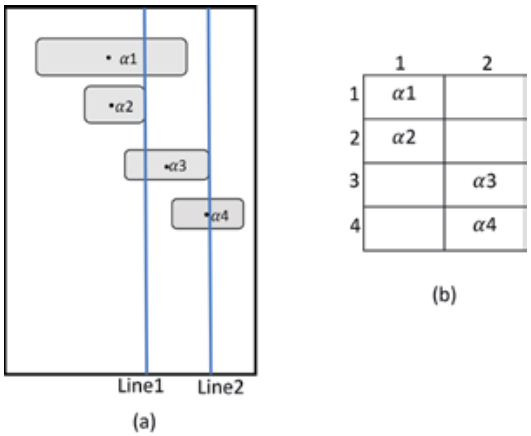

Figure 4: An example of the edge cutting method. (a) The original image. (b) The transformed 2D indexes (1,1), (2,1), (3,2) and (4,2).

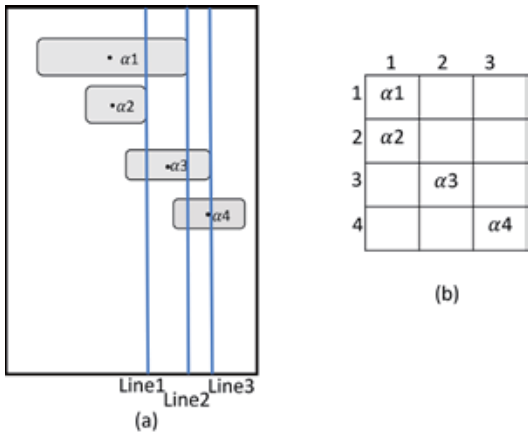

Figure 5: An example of the edge survival cutting method for the transformation of 2D indexes. (a) The original image. (b) The transformed 2D indexes (1,1), (2,1), (3,2) and (4,3).

edge cutting method, which cuts at the end of each component and uses the center point of the component as its position, as shown in Figure 4. Let A' represent remaining components in this image. First, we initialize the column index $c = 1$. The minimum rightmost coordinate of all components corresponds to component α_2, with label Line 1. The coordinates of the center points of components α_1 and α_2 are less than Line 1. Thus we assign the column index of components α_1 and α_2 as 1 and remove them from A'. Next, consider the remaining components α_3 and α_4 in A'. The minimum rightmost coordinate of them belongs to α_3. The center coordinates of components o_3 and α_4 are less than Line 2. Thus we assign the column index of α_3 and α_4 as $c = 2$. In addition, the row-index of components can also be obtained in the similar way.

The second is the *edge survival* method, in which each rightmost coordinate is a possible cutting line even if its component has been indexed previously, as shown in Figure 5.

4 Experimental Results

4.1 The Experimental Dataset

Each group of experimental data is composed of a template image, and 25 testing images, generated by slightly modifying the template. We generate 1000 groups of images for experiments.The generation of a template image is given as follows.

Step 1: Capture the appearances of components of AI2 programs. AI2 has eight basic components, including `slider`, `string`, `map`, `checkbox`, `list`, `textbox`, `button` and `image`.

Step 2: Randomly choose n ($5 \leq n \leq 13$) components for the image.

Step 3: Randomly generate strings of letters with lengths three to ten for a `button` or `string` component.

Step 4: Place these components (without overlapping) on the open area randomly.

4.2 Component Recognition

The YOLOv4 has very good ability in component recognition. This experiment contains the 1000 groups of images, where there are 26 images in each group. Thus, 26000 images, with more than 200,000 components totally, are used for this experiment. We use the *intersection over union* (IoU) to represent the accuracy. Table 2 shows the accuracies of component recognition with YOLOv4.

We compare the recognition results of YOLOv4 with the network tool UIED [6]. Taking Figure 6 as an example, it can be seen that the UIED detection method may produce errors for some components that are close to each other.

4.3 Text Recognition

For text recognition within `button` or `string` components, we use the Tesseract [21] and Keras-OCR [4]. We take text components of string

Table 2: The accuracies of component recognition obtained from YOLOv4.

Component	IoU = 0.6	IoU = 0.7	IoU = 0.8
slider	99.9%	99.9%	99.1%
string	100%	100%	100%
map	100%	100%	100%
checkbox	100%	100%	99.7%
list	100%	100%	99.9%
textbox	100%	100%	100%
button	98.5%	98.5%	98.5%
image	99.9%	99.9%	99.4%
Average	99.7%	99.7%	99.4%

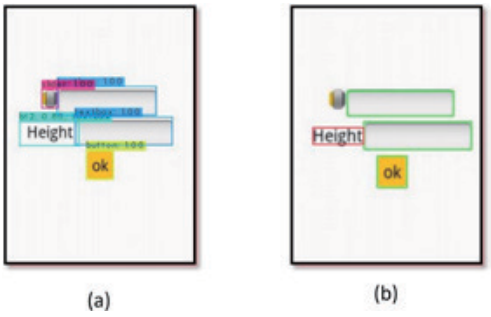

Figure 6: Examples of component recognition. (a) YOLOv4 method. (b) UIED method.

lengths from 3 to 10, and each text contains only lowercase English letters. We use the Tesseract and Keras-OCR to perform experiments with three different fonts, including Sans Serif, Serif, and Monospaces. There are more than 75000 `button` components and `string` components in the Sans Serif and Serif font, and totally about 42000 `button` components and 34000 `string` components in the Monospace font in 26000 images. We compare the recognized text with the correct text for three situations, including exact correctness (no error), errors of at most one character (within one error), and errors of at most two characters (within two errors). The experimental results are shown in Table 3, where Keras-OCR has better performance than Tesseract in all aspects

To further improve the accuracy of the 34000 `string` components, we analyze the situations of allowing different errors. Since the Monospace font has the same width for each letter, we chose this font for improvement. Table 4 shows the proportions for cases wrong recognized to another letter in font Monospace.

We find that the Monospace font has a lot of repeated wrong cases, where 86% of the errors stand for recognizing to another letters, 11% stand for recognizing too many letters, and the remaining

Table 3: An accuracy comparison for text recognition, where the text length is at most 10. Note that we use the corresponding font in each corresponding row.

Method	Font	No error	Within 1 error	Within 1 error
Tesseract	Sans Serif	79%	96%	99%
Tesseract	Serif	70%	91%	98%
Tesseract	Monospace	74%	95%	99%
Keras-OCR	Sans Serif	81%	97%	99%
Keras-OCR	Serif	85%	98%	99%
Keras-OCR	Monospace	84%	97%	99%
Keras-OCR + template match	Monospace	97%	99%	99%

Table 4: The proportions of various cases in about 1000 `string` components for wrong recognition to another letter in the Monospace font.

Proportion	Recognized letter	Correct letter
68%	g	q
15%	i	f, j, l(Lima)
10%	1(one)	l(Lima)
4%	a	q
1%	0(zero)	o(Oscar)
2%	others	

3% stand for recognizing too few letters. As a result, the accuracy can be improved to 97% in the 34000 `string` components.

4.4 2D Layout Similarity

In each group, there are 25 pairs of images, where the template image and each testing image forms one pair. Thus, we prepare 25000 pairs of images (in 1000 groups) for the experiments. In each pair of images, the maximal subset of matching components can be regarded as the largest common substructure.

The experimental results are shown in Tables 5 and 6 for the edge cutting method and the edge survival method, respectively, combined with 2D LCS. The edge survival method should have more cutting lines than the edge cutting method, since in the edge survival method, the rightmost boundary of a component is still kept as a possible cutting line after the component index has been decided.

It takes about an hour for judging every 100 groups of images (2500 pairs). That is, the judge job of about 40 pairs of images can be performed in one minute. With such a speed, the instant judg-

Table 5: The distribution for 2D layout similarities of AI2 programs by using the edge cutting method with various definitions of 2D LCS.

Similarity	0-49%	50-59%	60-69%	70-79%	80-89%	90-99%	100%
P(ENL)	0.011	0.050	0.152	0.186	0.363	0.033	0.205
P(ENE)	0.011	0.048	0.151	0.185	0.364	0.033	0.208
P(LOL)	0.007	0.024	0.092	0.129	0.372	0.067	0.309
P(LOE)	0.007	0.024	0.090	0.126	0.370	0.068	0.315

Table 6: The distribution for 2D layout similarities of AI2 programs by using the edge survival method with various definitions of 2D LCS.

Similarity	0-49%	50-59%	60-69%	70-79%	80-89%	90-99%	100%
P(ENL)	0.013	0.058	0.169	0.197	0.351	0.023	0.189
P(ENE)	0.013	0.057	0.168	0.196	0.352	0.023	0.191
P(LOL)	0.007	0.024	0.091	0.126	0.371	0.068	0.313
P(LOE)	0.007	0.023	0.089	0.124	0.370	0.069	0.318

ment on a small testing site (such as a classroom) can be achieved.

5 Conclusion

This paper proposes a judge method from the perspective of image processing to analyze the appearance of components. We apply the 2D LCS algorithm combined with image recognition to measure the layout similarity of application screenshots. This approach allows more diverse judge, and it is also a basis for scoring the similarity between two images.

Currently, we regard the scores of different components to be the same. In the future, it is possible to give different weights for different components. It is worthy of considering other component cutting methods, such as cutting the same component into multiple equal parts for comparison. These applications may make the scoring method more flexible and increase its usability.

In addition, in the field of very-large-scale integration (VLSI), the circuit layout is represented with the tree structure [23]. This notation may be applied to the representation of the 2D component layout in application. In the text part, it requires the recognition accuracy of text to reach 100%, but the current computer vision technology cannot achieve this accuracy. Therefore, in the future, we may have to grab the information of the text inside the AI2 program to ensure the correct interpretation of the text.

References

[1] "CodeMaster." `http://apps.computacaonaescola.ufsc.br:8080.`

[2] "Google Platform." `https://cloud.google.com/`.

[3] "Gurobi Optimization, Inc.." `http://www.gurobi.com`.

[4] "Keras-OCR." `https://github.com/faustomorales/keras-ocr`.

[5] "MIT App Inventor." `https://appinventor.mit.edu/`.

[6] "UIED." http://uied.online.

[7] A. Amir, T. Hartmana, O. Kapaha, B. R. Shaloma, and D. Tsur, "Generalized LCS," *Theoretical Computer Science*, Vol. 409, pp. 438–449, 2008.

[8] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Optimal Speed and Accuracy of Object Detection," *ArXiv*, Vol. abs/2004.10934, 2020.

[9] H.-T. Chan, *The Definitions and Computation of the Two Dimensional Largest Common Substructure Problems, Master's Thesis.* National Sun Yat-sen University, Kaohsiung, Taiwan, 2016.

[10] H.-T. Chan, H.-T. Chiu, C.-B. Yang, and Y.-H. Peng, "The generalized definitions of the

two-dimensional largest common substructure problems," *Algorithmica*, Vol. 82, No. 7, pp. 2039–2062, 2020.

[11] S.-K. Chang, Q.-Y. Shi, and C.-W. Yan, "Iconic indexing by 2-D strings," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-9, No. 3, pp. 413–428, 1987.

[12] S.-Z. Chiou, C.-B. Yang, and Y.-H. Peng, "The NP-hardness and APX-hardness of the two-dimensional largest common substructure problems," *Proceedings of the 34th Workshop on Combinatorial Mathematics and Computation Theory*, Taichung, Taiwan, 2017.

[13] N. da Cruz Alves, C. G. von Wangenheim, J. C. R. Hauck, and A. F. Borgatti, "A large-scale evaluation of a rubric for the automatic assessment of algorithms and programming concepts," *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, New York, NY, USA, pp. 556–562, Association for Computing Machinery, 2020.

[14] I. da Silva Solecki, J. V. A. Porto, K. A. Justen, N. da Cruz Alves, C. G. von Wangenheim, A. F. Borgatto, and J. C. R. Hauck, "CodeMaster UI Design - App Inventor: A Rubric for the Assessment of the Interface Design of Android Apps Developed with App Inventor," *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems*, New York, NY, USA, Association for Computing Machinery, 2019.

[15] G. B. Dantzig, "Reminiscences about the origins of linear programming," *Operations Research Letters*, Vol. 1, pp. 43–48, 1982.

[16] Y. Du, C. Li, R. Guo, X. Yin, W. Liu, J. Zhou, Y. Bai, Z. Yu, Y. Yang, Q. Dang, and H. Wang, "PP-OCR: A Practical Ultra Lightweight OCR System," *ArXiv*, Vol. 2009.09941, 2020.

[17] R. M. Francesco Maiorana, D. Giordano, "Quizly: A live coding assessment platform for App Inventor," *Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, Atlanta, GA, USA, pp. 25–30, 10 2015.

[18] Y. Li, Y. Pan, W. Liu, and X. Zhang, "An Automated Evaluation System for App Inventor Apps," *Proceedings of the 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Athens, Greece, pp. 230–235, 2018.

[19] Y. Pochet and L. A. Wolsey, *Production Planning by Mixed Integer Programming*. Springer-Verlag New York, 2006.

[20] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley and Sons, Inc. New York, 1986.

[21] R. Smith, "An Overview of the Tesseract OCR Engine," *Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 2, pp. 629–633, 2007.

[22] C. G. von Wangenheim, J. C. R. Hauck, M. F. Demetrio, R. Pelle, N. da Cruz Alves, L. F. Azevedo, and H. Barbosa, "CodeMaster – Automatic Assessment and Grading of App Inventor and Snap! Programs," *Informatics in Education - An International Journal*, Vol. 17, No. 1, pp. 117–150, 2018.

[23] N. H. Weste and D. Harris, *CMOS VLSI design: a circuits and systems perspective*. Pearson Education India, 2015.

[24] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "UIED: A Hybrid Tool for GUI Element Detection," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, p. 1655–1659, Association for Computing Machinery, 2020.

[25] Z. Xu, X. Baojie, and W. Guoxin, "Canny edge detection based on Open CV," *Proceedings of the 13th IEEE International Conference on Electronic Measurement Instruments (ICEMI)*, Yangzhou, China, pp. 53–56, 2017.

[26] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, "East: An efficient and accurate scene text detector," *ArXiv*, 2017.