Evaluating Control-Flow Graph Similarity for Grading Programming Exercises

Kevin Sendjaja
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: 13517023@std.stei.itb.ac.id

Satrio Adi Rukmono
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: sar@itb.ac.id

Riza Satria Perdana
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: riza@informatika.org

***Abstract*—Programming has become a fundamental skill in the current digital era. A formal programming course relies on an autograder to score student works. However, the usual black-box method only compares the output instead of adequately examining the code structure. As such, another method is required to measure the structure of the student submission code to give fairer scores. In this paper, an experiment is conducted using a control-flow graph (CFG) comparison algorithm to measure the similarity between student submission code and reference code from the instructor, followed by an analysis of the results obtained from the experiment. The comparison was made using the Hu Algorithm [1], based on previous CFG similarity measurements presented by Chan and Collberg [2]. Through the experiment and analysis process, it is concluded that the CFG comparison method implemented in this research is better applied to boost students who got low scores due to minor mistakes rather than be applied to the entire student submissions as the primary scoring algorithm.**

***Index Terms*—control-flow graph, cost matrix, white-box**

I. Introduction

In this digital era, programming skill is indispensable in many fields. People from all kinds of backgrounds are expected to keep up with the current technological developments. Today, anyone can hone their programming skills, either through formal education or by self-studying.

Most programming classes utilise an automated grading system using the black-box method, which compares the output from submitted code when given a particular input with the expected output. This method has already proven to be quite effective to grade programming assessments. However, it is not entirely uncommon that a minor mistake, such as writing the output in a slightly different format, causes a student to receive a bad score even when the logic of the program is mostly correct. Thus, while black-box autograders are suitable for competitions, where coders are already expected to create well-written codes, they are not entirely suitable for the learning environment in which we want to assess the learner's skills. This situation pushes the need for an additional grading method to give a fairer score for each student's work.

This paper aims to analyse the application of static analysis using control-flow graphs (CFG) to grade programming assessments. Our grading process compares the CFG from the student's submission with the CFG from the instructor's reference code. It uses a graph comparison algorithm to measure the similarity. In particular, we try to find out how this automated grader performs compared to manual evaluation and the traditional black-box grader with randomised test cases.

The rest of this paper is organised as follows. In Section II, we present the foundational bases to our work. Then, we propose our automated grading system in Section III. We explain our experiment next in Section IV and discuss the result in Section V. Finally, we draw our conclusion and suggest future work in Section VI.

II. Foundational Bases

A. Control-Flow Graph

Control flow analysis refers to the technique of analysing the control flow of a program [3]. A control flow can be described as the execution sequence of the instructions within the program. An instruction can be defined as lines of code that execute a particular task or function calls. This term was used by Francis E. Allen in his journal back in 1970 [4]. The relations between the control flow can be drawn in the form of a directed graph, referred to as Control-Flow Graph (CFG).

A node represents a basic block within a CFG, i.e., a series of instructions with exactly one entry point and one exit point. Meanwhile, an edge represents the path that is passed by the control flow. An example of a CFG can be seen in Fig 1.

CFGs have been used for various use cases. For example, Harrold et al. [5] used this approach as an alternative for representing and analyzing software. Kruegel et al. [6] applied the same technique to detect network worms. Vujosevic-Janicic et al. [7] also used this approach to analyze assessments submissions.

B. Control Flow Graph Similarity Algorithms

There is no "standard" metric for measuring the similarity between two CFGs. As such, many researchers defined their algorithms over the years. Chan and Collberg [2] categorized and compared some of the algorithms as follows.

a) k-subgraph mining: This algorithm was presented by Kruegel et al. [6], and its primary focus is to measure the similarity of CFGs based on the similarity of their subgraphs. k-subgraph refers to a subgraph of each node within the graph which spanned from the current root node such that the out-degree of each node ≤ 2, and the in-degree ≤ 1. Each of these

978-1-6654-9453-3/21/.00

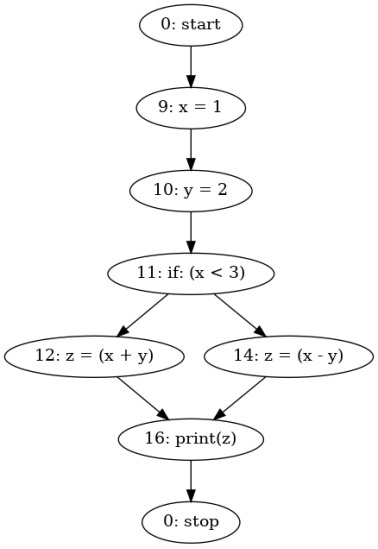

Fig. 1: Example of a CFG

k-subgraphs will then be plotted into a unique integer, which serves as its' fingerprint. These fingerprints are later used to determine whether two graphs are similar or not.

b) Edit distance/cost-based algorithm: Hu et al. [1] proposed an algorithm to measure graph similarity by determining the minimum number of transformations required to transform a graph into another. The algorithm was first designed to match software to currently known malware. A cost matrix is used to calculate the total cost needed. Once the cost matrix is defined, the minimum cost is found using the Hungarian Algorithm. The result is the total cost or edit distance between the two graphs, which could be derived into a similarity score.

c) Neighbor matching: Vujosevic-Janicic et al. [7] designed this algorithm for grading student assessments. The basic idea is to create a similarity matrix between two CFGs through an iterative process. The values are calculated through the similarities between neighbouring nodes. Once the iteration process has stopped, the similarity between the two graphs can be obtained using optimal matching on the similarity matrix.

C. Autograder

Autograder refers to software used to automatically execute submitted programs and give a proper grading for each submission. The main reason for its usage is due to a large number of students and a large number of assignments, which would cost a lot of time and energy if the lecturer has to grade each one manually. The grading process of an autograder can be seen in Fig 2

A programming instructor must provide a series of test cases used when the submitted code is executed. A grader will first compile the submitted code to make sure that the program can be executed. Then, the inputs from the test cases will be supplied to the program, and the output will be matched with the expected output from the test cases to get the final grade.

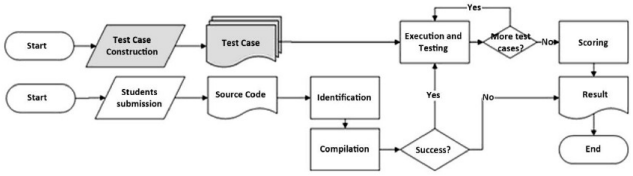

Fig. 2: Grading process using the black-box method
Source: (Danutama, K. Liem, I, 2014) [8]

Such a method is called the black-box method, where grades are determined by comparing program outputs.

In competitive programming, grading is done only by whether the submission can generate the correct outputs, given specific inputs. However, when it comes to a learning process, students are not only expected to submit the correct code. They are also expected to pay attention to code readability, follow coding conventions, and other aspects that are related to taught materials. This kind of grading is referred to as white-box grading, where the process involves analyzing the contents of the code. White-box methods are relatively more challenging than black-box due to the extra efforts required to analyze the entire code.

III. Proposed Solution

A. Solution Design

The proposed solution involves comparing the difference between the CFG of the student's submission to the CFG of the reference code given by the lecturer. This approach would allow the program to analyse the correctness of the code based on the similarity of the code structure against the expected code, providing a more thorough assessment compared to the usual black box method. First, an assessment tool is implemented, utilising CFG comparison algorithms on submission codes and reference codes. Some cases where the white-box and black-box scores have significant differences will then be graded manually by an expert. The expected result is that the white-box score from the CFG comparison algorithm can be as similar as possible to the scores from manual grading.

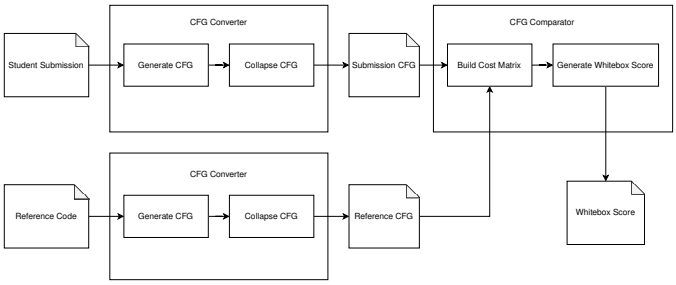

Fig. 3: Diagram of the proposed grading system

Fig. 3 describes the design of the proposed system. The system will run several submission codes at once, and the scores will be saved in an output file for evaluation. It is possible to use more than one reference code for each problem

since there may be more than one correct program for a problem. When more than one reference code is provided, each submission will be compared to every reference code, with the highest score regarded as the final grade.

B. CFG Converter

The CFG Converter module converts both submission and reference codes into their CFG representations, later used in the grading process. Before grading, both CFGs go through the "collapse" process, which combines all nodes located within a single linear flow. The steps of the collapse process are as follows.

1) Create a list that contains all nodes that will be removed by the end of the process.
2) Iterate over each node within the graph.
3) When the current node only has one outgoing edge and is not within the list, mark the node as start node.
4) If the direct successor only has one incoming edge and one outgoing edge, the direct successor will become the current node in the next iteration. The current node is then added to the list, and the content of the current node will be added to the start node.
5) Step 4 is repeated until a node that does not satisfy the conditions is found. The current node will then be marked as a stop node. When the immediate successor of the current node has no outgoing edge, that immediate successor will be marked as a stop node.
6) If the start node is different from the stop node, add an edge from the start node into all nodes with an incoming edge from the stop node.
7) Repeat steps 2 to 6 until the iteration has covered all nodes in the graph.
8) For each node in the list, remove all edges connected to those nodes from the graph.
9) Remove all nodes in the list from the graph.

C. CFG Comparator

The CFG Comparator module compares the CFG representations and generates the final grade. A score between 0 to 100 represents the grade. We chose to implement the Hu algorithm [1] to compare the graphs, considering its good accuracy and execution time based on the research conducted by Chan and Collberg [2].

The graph representations will be used to create the cost matrix. The process of creating the cost matrix are as follows:

1) For graphs G1 and G2, with V1 and V2 are set of nodes for each graph, create a zero matrix with (|V1| + |V2|) * (|V1| + |V2|) dimension.
2) For the upper left submatrix with |V1| * |V2| dimension, replace the elements using Equation 1. The cost of transforming the node contents is not included in this solution.

$$\begin{aligned} a_{ij} = relabeling\ cost + (|(ON)_i| + |(ON)_j| \\ -2 * |(ON)_i \cap (ON)_j|) + (|(ON)_i| + |(ON)_j| \\ -2 * |(ON)_i \cap (ON)_j|) \end{aligned} \quad (1)$$

where ON and IN represent the number of nodes connected by out-going/in-going edges from each respective nodes.

3) For the upper right and lower left submatrices, with |V1| * |V1| and |V2| * |V2| dimensions respectively, replace the elements using Equation 2. When $i \neq j$, replace the element with 999.

$$a_{ij} = 1 + |IE_{(i/j)}| + |OE_{(i/j)}| \quad (2)$$

where IE and OE represent the number of in-going and out-going edges from each respective nodes.

$$\begin{pmatrix} \textcircled{0} & 1 & 2 & 2 & 5 & 3 & \infty & \infty & \infty \\ 2 & \textcircled{1} & 2 & 2 & 3 & \infty & 5 & \infty & \infty \\ 2 & 1 & 0 & \textcircled{0} & 3 & \infty & \infty & 3 & \infty \\ 4 & 3 & 2 & 2 & \textcircled{1} & \infty & \infty & \infty & 3 \\ 3 & \infty & \infty & \infty & \infty & \textcircled{0} & 0 & 0 & 0 \\ \infty & 4 & \infty & \infty & \infty & 0 & \textcircled{0} & 0 & 0 \\ \infty & \infty & \textcircled{3} & \infty & \infty & 0 & 0 & 0 & 0 \\ \infty & \infty & \infty & 3 & \infty & 0 & 0 & \textcircled{0} & 0 \\ \infty & \infty & \infty & \infty & 4 & 0 & 0 & 0 & \textcircled{0} \end{pmatrix}$$

Fig. 4: Minimum cost selection using Hungarian Algorithm Source: (Chan, P. P. F., & Collberg, C., 2014) [2]

Since the cost matrix was initialized with 0, no changes are required for the lower right submatrix. The total cost will then be calculated from the cost matrix using the Hungarian Algorithm, which is also used by the original Hu algorithm [1]. The final grade will then be calculated from the total cost as the white-box score. Fig. 4. visualizes the application of the Hungarian Algorithm on the cost matrix.

IV. Experiment

A. Experiment Setup

The software for the experiment is implemented using Python version 3.8.5. The conversion process from code into its CFG representation is done using the Python library pycfg. The graph representation is saved in the form of AGraph class from PyGraphViz. This representation then goes through the collapse process to avoid redundant nodes.

The cost matrix will be created from the collapsed graphs, both submission and reference graphs. The matrix is constructed using the steps from Section III.A. The Python library Munkres is used for applying the Hungarian Algorithm on the cost matrix. Finally, the white-box score is obtained through equation 3. V and E symbolizes the set of nodes and edges for each graph.

$$final\ score = \frac{total\ cost}{|V_1| + |E_1| + |V_2| + |E_2|} \quad (3)$$

A black-box grader is also implemented for comparison purposes. It runs the submission code using provided test cases and compares the output with the expected values. This grader provides a score on a scale of 0 to 100. Should an error is

detected, the submission will automatically be given a score of 0.

B. Test Data

The experiment data are obtained from student submission codes from a freshman-level programming course. The test uses two problem sets that were given to every student. These problem sets are chosen to examine various algorithmic fundamentals, including conditionals, loops, and functions. Due to test data limitations, recursive functions are not tested. 4 different reference codes are provided for each problem. The problem set descriptions are as follows.

1) *square*: Given an integer n and 2 characters $c1$ and $c2$, construct a string in the shape of a square sized n using $c1$ as border and $c2$ as fill. If $n \leq 0$ or $c1 = c2$, print an error message instead.
2) *count_vowels*: Given a string, write a function to write the string into a file, then read the file and write the number of vowels within the string. The final character in the string is always a dot (".").

C. Experiment Process

The experiment follows the steps described below.

1) The software runs the CFG algorithm on a series of submission and reference codes. If two or more reference codes are given, it compares each submission to each reference, and the highest score is regarded as the white-box score for the submission.
2) The software also runs the black-box grading for each submission using provided test cases.
3) The software writes down the results and the identity for each submission in a spreadsheet. The difference between white-box and black-box scores are also documented.
4) Several cases with notable differences between white-box and black-box scores are analysed manually by an expert. Due to the limitations in time and resources, manual grading is only applied to a small portion of the entire submissions.

A programming lecturer performs the manual grading without seeing the white-box scores before to guarantee objectivity. The identity for each submission not be shown for the sake of privacy.

D. Experiment Results

The experiment was conducted on 434 submission codes for the $square$ problem set and 444 submission codes for $count\_vowels$. The results can be seen from Fig. 5 and Fig.6.

The test result shows 12 submissions with significant differences between white-box and black-box scores from each problem. These submissions are then manually analyzed and graded. The score comparison can be seen on Table I and Table II.

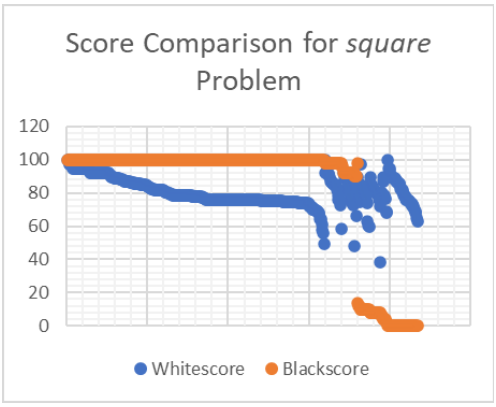

Fig. 5: Score comparison visualization for $square$ problem

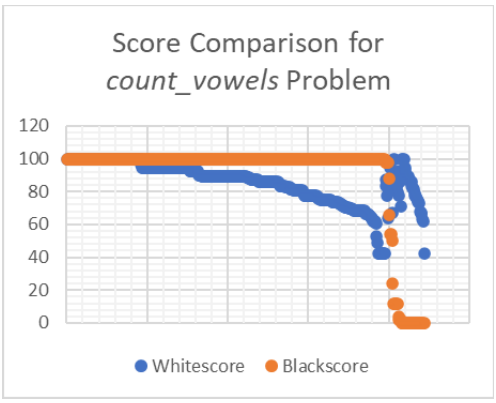

Fig. 6: Score comparison visualization for $count\_vowels$ problem

V. Discussion

From Fig. 5 and Fig. 6, it can be seen that the white-box score is more evenly spread out compared to the black-box score, with a relatively higher minimum score as well. However, Table I and Table II show that the white-box scores still differ significantly from manual grading in most cases.

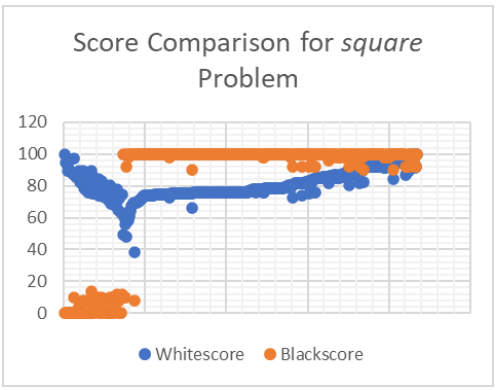

Fig. 7: Score comparison visualization for $square$ problem sorted by difference

TABLE I: Score Comparisons for *square* Problem

Case No	Whitescore	Blackscore	Manual Score
1	89.74	14.00	50.00
2	49.09	100.00	93.75
3	86.84	4.00	87.50
4	59.41	10.00	87.50
5	38.46	8.00	25.00
6	68.18	2.00	25.00
7	94.37	0.00	75.00
8	97.30	10.00	87.50
9	62.77	0.00	62.50
10	47.79	92.00	87.50
11	85.29	8.00	50.00
12	58.59	98.00	93.75

TABLE II: Score Comparisons for *count_vowels* Problem

Case No	Whitescore	Blackscore	Manual Score
1	42.11	100.00	100.00
2	87.50	98.00	87.50
3	94.59	54.00	62.50
4	67.12	24.00	62.50
5	64.00	98.00	75.00
6	100.00	0.00	37.50
7	100.00	12.00	37.50
8	77.78	4.00	62.50
9	42.11	0.00	12.50
10	71.11	2.00	25.00
11	89.19	50.00	62.50
12	92.31	2.00	37.50

When the individual results are sorted by their differences from the largest to smallest (Fig. 7 and 8), it can be seen that the white-box and black-box scores display similar variation patterns in several places. Moreover, the patterns are almost identical when both scores are high, albeit white-box scores are relatively lower. This phenomenon indicates that the CFG comparison algorithm results do not go against the usual black-box method.

The algorithm generally provides better scores when the black-box score is very low or 0, which may be concluded that the white-box score can give a more proper score when the submitted code contains an error that causes a low black-box score. However, it should also be noted that when the black-box score is very high or perfect, the algorithm tends to give lower grades than expected.

The scores from Table I and Table II are used to produce a multilinear regression model that can be seen on equation 4. y signifies the manual score, while x1 and x2 represent white-box and black-box scores, respectively. The model is not absolute due to the limitation in the number of data, but it is enough to estimate that the white-box score has less impact on the manual score than the black-box score.

$$y = 30.4356 + 0.2243 * x1 + 0.4569 * x2 \quad (4)$$

A simulated score is generated through equation 4 to simulate the grade from manual grading for all submissions. Fig. 9 and Fig. 10 shows the score comparison between white-box, black-box, and simulated scores for both problems. It can be seen that before the black-box score reaches 100, the white-box score and simulated score tend to change direction at the x-axis, albeit with a difference in the y-axis and how significant does the direction change. Once the black-box score is consistent at 100, both the white-box score and simulated score stop changing directions and consistently move toward 100. This result supports the theory that white-box scores generally are closer to manual scores when the black-box scores are low.

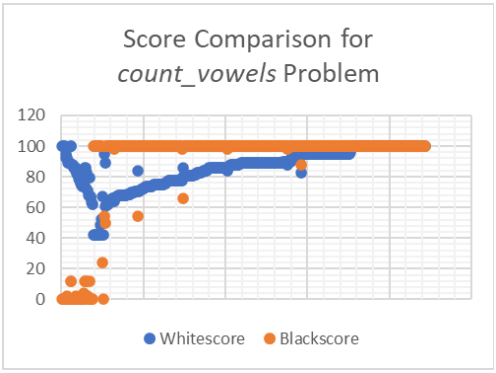

Fig. 8: Score comparison visualization for *count_vowels* problem sorted by difference

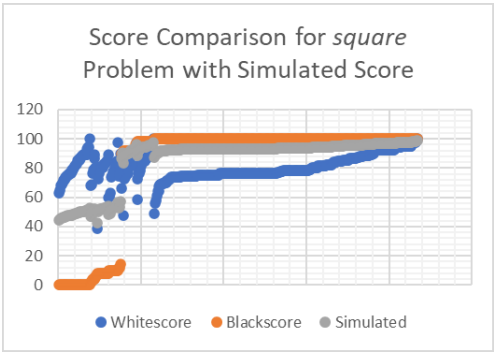

Fig. 9: Score comparison visualization for *square* problem with simulated score

Fig. 11 and Fig. 12 shows the distribution of all three scores in the form of boxplot. Both graphs show that the white-box scores generally have a larger range and smaller number of outliers. This shows that the white-box scores are more evenly distributed compared to the other scores. However, one thing that should be noted is that the lower range of the white-box score is longer than both black-box and simulated scores. If the outliers are ignored, then it can be assumed that there will be many cases when the white-box score is lower than the manual score.

To see how far the white-box score deviates from the manual score, we calculated the Mean Absolute Error (MAE), with the white-box score as the predicted value and the manual score as the actual value. MAE is calculated for each problem, using manual grading when available and simulated scores

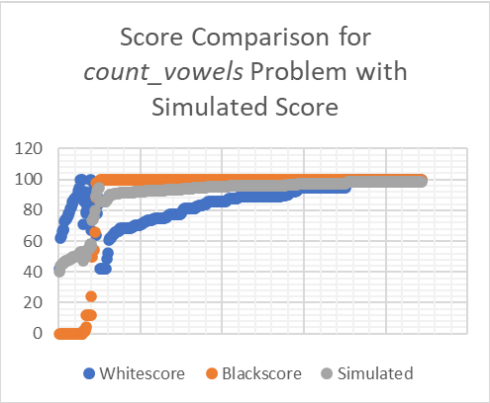

Fig. 10: Score comparison visualization for $count\_vowels$ problem with simulated score

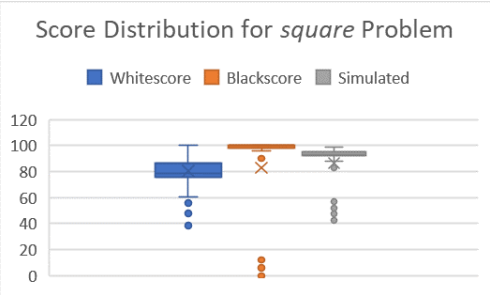

Fig. 11: Score distribution for $square$ problem

otherwise. The scores can be seen in Table III. The scores suggest that the white-box score is still not accurate enough to simulate the manual score given by a lecturer or an expert.

Further analysis on cases where white-box scores give higher results than black-box and manual scores reveals that the code's mistakes are located within the collapsed nodes. The collapse process is meant to simplify the CFG so that slight writing style differences between the submission and reference codes would not reduce the score. However, in cases where there are actual coding mistakes within the code that follow a single linear flow, the collapse process would cause the mistake to be skipped, resulting in a higher score than it should be. As such, further improvements are required so that the collapse process can be used effectively.

VI. Conclusion

The test software built for this project implements the CFG comparison algorithm to measure the similarity between student submissions and reference codes from the lecturer. Through experiment and analysis, it was found that the white-box score from the graph comparison algorithm is relatively good when the submission code contains minor mistakes but is structurally correct. However, since the white-box score is relatively low when the black-box score is high, as well as the fact that the white-box score is not accurate enough to simulate the manual grading, it can be concluded that this current CFG comparison scoring implementation is better suited as a method to boost student's scores that are low due to slight mistakes while writing the code, rather than be applied to all student submissions.

TABLE III: MAE Score Comparison Table

	12 Submissions	All Submissions
$square$	25.7825	16.0459
$count\_vowels$	33.5917	11.7257

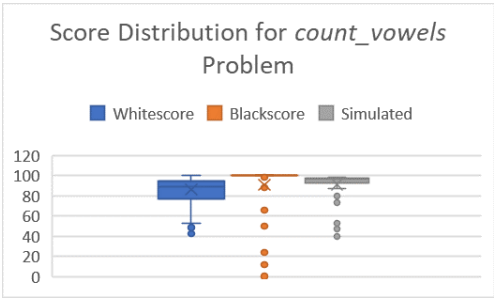

Fig. 12: Score distribution for $count\_vowels$ problem

Future work

While the research covered by this paper only includes CFG comparison by the graphs' structures, further comparison by comparing the contents of each node would allow a more accurate score generation. A weighted score could also be implemented so that critical nodes would significantly affect the final score. If the algorithm has been improved until accurate enough, it may also be implemented within a Learning Management System (LMS) for automatic grading purposes.

Acknowledgment

The research covered in this paper is a part of a more extensive work regarding automatic grading involving our peers Irfan Sofyana Putra, M. Rifky I. Bariansyah, and Bram Musuko Panjaitan.

References

[1] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009, p. 611–620.

[2] P. P. Chan and C. Collberg, "A method to evaluate cfg comparison algorithms," in *2014 14th International Conference on Quality Software*, 2014, pp. 95–104.

[3] R. Dévai, J. Jász, C. Nagy, and R. Ferenc, "Designing and implementing control flow graph for magic 4th generation language," 2013.

[4] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*, 1970, pp. 1–19.

[5] H. M. E. *et al.*, "Ras representation & analysis software," 2003.

[6] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," *RAID*, pp. 207–226, 2006.

[7] M. Vujosevic Janicic, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments," *Information and Software Technology*, vol. 55, p. 1004–1016, 2013.

[8] J. Fernando and M. Liem, "Components and architectural design of an autograder system family," vol. 8, pp. 69–79, 2014.