COMPLEXITOR: AN EDUCATIONAL TOOL FOR LEARNING ALGORITHM TIME COMPLEXITY IN PRACTICAL MANNER

Elvina[1] and Oscar Karnalim[2]

[1, 2,] Faculty of Information Technology, Maranatha Christian University
Jln. Prof. Drg. Surya Sumantri No. 65, Bandung, Jawa Barat, 40164, Indonesia
[1]elvinazhang@gmail.com; [2]oscar.karnalim@it.maranatha.edu

Received: 6th December 2016/ **Revised:** 9th February 2017/ **Accepted:** 10th February 2017

Abstract **-** Based on the informal survey, learning algorithm time complexity in a theoretical manner can be rather difficult to understand. Therefore, this research proposed Complexitor, an educational tool for learning algorithm time complexity in a practical manner. Students could learn how to determine algorithm time complexity through the actual execution of algorithm implementation. They were only required to provide algorithm implementation (i.e. source code written on a particular programming language) and test cases to learn time complexity. After input was given, Complexitor generated execution sequence based on test cases and determine its time complexity through Pearson correlation. An algorithm time complexity with the highest correlation value toward execution sequence was assigned as its result. Based on the evaluation, it can be concluded this mechanism is quite effective for determining time complexity as long as the distribution of given input set is balanced.

Keywords: Complexitor, educational tool, learning algorithm, time complexity

I. INTRODUCTION

The algorithm is the core topic of Computer Science (CS) field. However, since not all undergraduate CS students can understand it properly, several CS educational tools are developed. These tools target various level of algorithm understanding. It starts from technical implementation (e.g. program creation) to abstractive form (i.e. algorithmic steps).

To help students in technical implementation, various educational tools are used to encourage students to implement their user-defined algorithm into source code (Guo, 2013; Laakso, Kaila, & Salakoski, 2008; Cisar, Pinter, Radosav, & Cisar, 2010). However, these tools have several unique features which distinguish them from standard Integrated Development Environments (IDEs). Guo (2013) proposed Python Tutor, an embeddable Web-based program visualization which aid ed students by providing reversible source code tracking. Students can see how their algorithm works in sequential order and verify variable contents during the process. Similar with Python Tutor, Jeliot 3 (Cisar, Pinter, Radosav, & Cisar, 2010) and Ville (Laakso, Kaila, & Salakoski, 2008) also incorporate program code tracking. However, there searches are featured with several additional features such as pop-up question and program comprehension.

To avoid syntactic difficulties, several researches incorporate unique yet effective mechanisms. Radosevic, Orehovacki, and Lovrencic (2009) incorporated a ”traffic-light” system which limited the number of source code modification before compiling. Students are forced to compile their code every N modification, and they can only continue to write code if their code is successfully compiled. In such manner, the numbers of syntactic errors handled by students for each code compilation are limited. Students will never face over whelming errors caused by writing large source code at once without compiling. On the other hand, Carlisle, Wilson, Humphries, and Hadfield (2005) and Watts (2004) used flowchart-like representation for constructing algorithm. Students are not encouraged to write source code directly. Instead, they construct algorithm flowchart converted into source code. This mechanism may avoid syntactic errors which are majorly caused by permitting students to write code syntaxes directly. Areias and Mendes (2007) also applied flowchart-like representation for constructing algorithm, but they focused on teaching weak students by incorporating problem-specific questions. These questions were statically defined for each problem and utilized to guide students to solve particular problem.

Even though learning algorithm implementation is important, it can only be conducted when students have already understood its algorithmic steps. Thus, several researches are focused on teaching algorithmic steps instead of technical implementation. CeeBot4 (http://www.ceebot.com/ceebot/4/4-e.php), Karel Robot (Buck & Stucki, 2001), and Alice (Cooper, Dann, & Pausch, 2000) are several examples in this category. These researches teach students to solve problem algorithmically through interactive environments (e.g. 3D visualization and real-world object). Then,students can arrange their algorithm based on provided instructions and see how their composed algorithm researches.

Meanwhile, there are researches focused on teaching how standard algorithms work instead of aiding students to construct their algorithm like VisuAlgo which incorporates animation and visualization to teach standard algorithms (Halim, 2011; Ling, 2014; Halim, Koh, Loh, & Halim, 2012). Students can learn how an algorithm works based on a particular input and see variable state for each given instructions through visualization. Similar to VisuAlgo, AP-ASD1 (Christiawan & Karnalim, 2016) also adds animation and visualization. However, it is more focused on covering course materials. It is fully-synchronized with Basic Algorithm and Data Structure course on Faculty of Information Technology in Maranatha Christian University, Indonesia.

To enhance students' understanding further, several researches do not only focus on teaching how standard algorithms work. Instead, they also focus on why a particular algorithm is better than others. Velázquez-Iturbide and Pérez-Carrasco (2009) developed GreedEx, an educational tool focused on learning greedy algorithm. Using GreedEx, students can explore how several greedy algorithms work like comparing their respective output, and determine which greedy algorithm is the best approach to solving the

given problem. In addition, their research is also extended by incorporating Computer-Supportive Collaborative System (CSCL) and named GreedExCol (Debdi, Paredes-Velasco, & Velázquez-Iturbide, 2015). On the contrary, Jonathan, Karnalim, and Ayub (2016) developed AP-SA, an educational tool to learn algorithm strategy (i.e. brute force, greedy, back tracking, and dynamic programming). Their research incorporated case-based performance comparison so that students could determine which algorithm was the best solution to the given problem. There are five aspects that are considered on case-based performance comparison. These aspects are optimality, completeness, time complexity, execution time, and output. All of them are expected to represent algorithm characteristic for a particular problem.

Nevertheless, to the researchers' knowledge, there is no educational tool focused on teaching how to calculate time complexity. Thus, this research proposes Complexitor, an educational tool focused on teaching algorithm time complexity through algorithm implementation. This tool is named based on its function and stands for "Time Complexity Calculator". For the initial step, Complexitor is only focused on teaching how to measure time complexity in a practical manner. Students could learn how to calculate time complexity based on actual execution of algorithm implementation. In addition, because algorithm implementation must rely on the particular programming language, Complexitor is also added with a programming language setting mechanism so students could incorporate new programming languages dynamically.

II. METHODS

In general, students can learn algorithm time complexity through Complexitor by the following flowchart in Figure 1. For each target of the algorithm, students must provide algorithm implementation or source code in a particular programming language, and input set. Afterward, two kinds of number sequences are generated which are execution and complexity-defined sequence. The execution sequence is generated based on the number of processed instruction or execution time through actual execution. This step requires programming language setting defined before hand since programming language instruction may be different per programming language. On the other hand, complexity-defined sequences are set by executing standard complexity functions to input size. After both sequences are generated, time complexity for the target algorithm is assigned with the most correlated complexity to execution sequence.

The learning flowchart in Figure 1 is inspired from the second author's teaching method about algorithm time complexity for weaker students. Before learning algorithm complexity in a theoretical manner, students are encouraged to see directly how algorithm time complexity is determined through actual execution. The time required for executing a particular algorithm is calculated manually based on input sets, and its complexity is determined based on its sequence similarity toward a particular complexity function. Based on the informal survey in the academic year of 2014/2015 on Faculty of Information Technology in Maranatha Christian University, Indonesia, this method is quite effective to enhance students' further understanding of time complexity of the algorithm.

To learn how to determine time complexity toward a particular algorithm, students must prepare algorithm implementation and input set. Algorithm implementation must be represented as a single file and should have a main method. It can be written in any programming language as long as its setting has been registered in Complexitor. Meanwhile, input set is seen as multiple test case files which each file should be named with input size (N), and its content should represent input details. The example of testcase files for linear search algorithm can be seen in Table 1. In this example, the researchers assume that implementation of linear search has accepted three lines as its input. The first line represents input size, the second line is input sequence, and the last line shows searched value. All testcases provided in Table 1 are intended to represent the worst case scenario like the searched value is not found.

Table 1 The Sample of Test Case Files of Linear Search Algorithms

Filename (N)	File Content
1	1 1 -1
3	3 1 2 3 -1
5	5 1 2 3 4 5 -1
10	10 1 2 3 4 5 6 7 8 9 10 -1

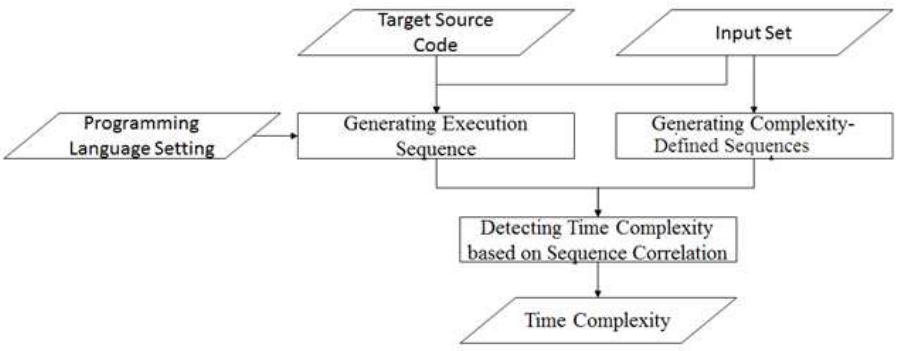

Figure 1 Learning Flowchart of Complexitor

From the learning perspective, students are asked to provide algorithm implementation and input set based on the following reasons. First, most of the students miscalculate time complexity since they do not know how to implement it as a program. They naively assume that each algorithm instruction has its one-to-one correlation with program instruction. Thus, by providing algorithm implementation, students are expected to understand more about their target algorithm in detail. Second, in most cases, determining time complexity is split into three domains which are best, worst, and average cases. By providing input set, students can determine which time complexity they want to calculate. For example, if students want to calculate the worst case for linear search algorithm, they should provide input set where each searched value is not found.

To design Complexitor as language-independent as possible, the researchers separate all language-dependent instructions and store them in programming language setting. Hence, Complexitor can incorporate various programming languages as long as their settings are provided. Language-dependent instructions are utilized to perform several tasks to generate execution sequence. These tasks include compiling source code, running executable file, and calculating the number of processed instructions. The first two tasks are for performing the actual execution. Both of them require students to arrange command prompt instructions for performing the tasks using a particular programming language like java and javac instruction in Java programming language. On the contrary, the latter one is to generate execution sequence based on standard counter mechanism. Students must describe how to initialize, increment, and print the counter in a particular programming language. Then, Complexitor will embed these instructions in the loaded source code, execute, and extracts its result by overriding command prompt instructions. Moreover, by default, Complexitor provides three programming language settings which cover C++, Java, and Python. These languages are selected due to their popularity in undergraduate students in Faculty of Information Technology in Maranatha Christian University, Indonesia.

For a broader view of programming language setting, the setting of C++ programming language in Complexitor can be seen in Table 2. It is compiled by using GNU Compiler (g++) and its counter mechanism relies on long long int type. In addition, because compiling and running instructions require file name and path to perform their respective tasks, Complexitor provides three variables for the convenience of access. These variables are a file name with extension, file name without extension, and file path. They are named as @filenamewithextension, @filenamewithoutextension, and @filepath respectively.

Table 2 C++ Programming Language Setting

Instruction Type	Value
Compile	g++ @filepath\@filenamewithextention -o @filepath\@filenamewithoutextention
Run	@filepath\@filenamewithoutextention.exe
Initialize counter	long long int count = 0;
Increment counter	count++;
Print counter	cout<<"\n"<<count;

The execution sequence is based on actual execution of algorithm implementation toward the given input set. It is represented as an array with length *K* which *K* is the number of test case files on input set. For test case file of *i* on input set, its execution result will be stored in the array at index *i*. By default, the execution result is determined according to the time required for a particular program to process a particular input. It is measured in nanoseconds to keep its sensitivity toward short execution time. However, the time execution is quite unreliable due to hardware and operating system dependency. Thus, the researchers' tool also adds the number of processed instructions to generate execution sequence. Students can select the time execution or the number of processed instructions as the baseline for execution sequence.

The number of processed instructions is calculated by embedding standard counter mechanism on algorithm implementation. A counter is initialized at the beginning of the process, incremented each time by involving processed instructions, and printed as a result at the end of the process. Instead of embedding counter mechanism automatically, Complexitor suggests the students embed them semi-manually. Students must determine where to initialize, increment, and print the counter in algorithm implementation. Therefore, students are encouraged to learn where to start and end calculating time complexity. As been known, not all of the instructions are counted to determine time complexity. Most input and output instructions are frequently excluded. In addition, they are also encouraged to determine which instructions are involved by embedding counter increment after each instruction. Based on the fact that interaction may strengthen students' understanding (Naps *et al.,* 2003). This supplementary interaction is expected to enhance students' further understanding.

The example of embedded algorithm implementation can be seen in Figure 2. The researchers translate the source code into the algorithmic form, so it only displays necessary information for clarity. This algorithm represents linear search algorithm where italic lines represent embedded instructions. The red-marked line shows counter initialization. The green -marked line is a counter increment, and blue-marked line means counter print instruction. In this example, the researchers intend to calculate time complexity only based on loop iteration, so they only increment the counter each time when the loop iteration is executed. In fact, counter increment can be embedded more than once in algorithm implementation. It relies heavily on students' assumption about the instructions in the given implementation of the algorithm.

```
1 |  n = input()
2 |  lst = new ArrayOfInteger()
3 |  for i to n do
4 |    lst[i] = input()
5 |  s = input()
6 |  count = 0
7 |  idx = -1
8 |  for i to n do
9 |    count = count + 1
10|    if lst[i] = s do
11|      idx = i
12|  if idx = -1 do
13|    print("not found")
14|  else
15|    print("found")
16|  print(count)
```

Figure 2 Implementation of Embedded Linear Search Algorithm

Furthermore, Complexity-defined sequences are generated based on standard complexity functions that take input size (N) from each test case as its argument. With an assumption that standard progressive complexity functions are limited to $T(\log N)$, $T(N)$, $T(N \log N)$, $T(N^2)$, $T(N^3)$, $T(2^N)$, $T(3^N)$, and $T(N!)$, this phase will generate 8 complexity-defined sequences or one sequence for each function. The representation of Complexity-defined sequence is quite similar to the execution sequence except it is generated by using complexity function instead of actual execution. For each test case file of i on the input set with N as its input size, the result of complexity function $f(N)$ will be stored in the array at index i. For example, if it is $i = 4$, $f(N) = 2^N$, and N=3, its function result will be $2^3 = 8$ and be placed at index 4 in array.

Then, time complexity is determined by selecting the most correlated complexity toward execution sequence. For each complexity-defined sequence of i, its correlation toward execution sequence of e will be measured based on Pearson correlation (Pearson, 1895). Pearson correlation is to measure linear dependence between two sequences (i and e) and yields -1 to 1 inclusively. -1 is the total of negative linear correlation whereas 1 shows the total of positive linear correlation. After all complexity correlations are calculated, time complexity with the highest correlation score is selected as a result.

However, this approach can only be applied on progressive time complexity due to Pearson correlation characteristic. Thus, constant time complexity is predicted before hand based on Mean Average Deviation (MAD) threshold. An execution sequence is considered to result in constant time complexity (*iff*) which the MAD is lower than K times its mean. Otherwise, the complexity will be determined according to Pearson correlation. The researchers do not incorporate gradient mechanism because not all points on both sequences yield a linear form, especially on time execution sequence. K is a parameter which may be changed, and the value is purely based on manual observation about data distribution. This MAD threshold rule is applied with an assumption that constant time complexity should generate small variance among the data.

III. RESULTS AND DISCUSSIONS

Based on the fact that the proposed approach for detecting time complexity is rather unique, the researchers intend to evaluate its effectiveness toward various time complexities and programming languages. To do that, the researchers generate a dataset which represents various time complexity functions implemented in various programming languages. In this dataset, nine standard time complexity functions are incorporated into nine algorithms which details can be seen in Table 3. Each algorithm is featured with 10 test cases where each of them is schemed to perform their respective complexity.

Each algorithm in Table 3 is implemented in three programming languages which differ in runtime execution due to their compiler design. These programming languages are C++, Java, and Python. C++ takes the fastest runtime execution since the executable file is represented as the native codes. Meanwhile, Java takes longer runtime execution than C++ since the executable file must be converted to native codes by using runtime environment. Then, Python takes the longest runtime execution because the source code is translated directly into native codes at the runtime. In short, the dataset consists of 27 cases which are from 9-time complexity functions times 3 programming languages. These cases are expected to represent time complexity and programming language variance.

The evaluation is split into three folds which are: (1) evaluating the effectiveness of sequence correlation for determining time complexity; (2) evaluating the effectiveness of the approach toward various programming languages; and 3) comparing the effectiveness of the two baselines for detecting time complexity. All evaluation is conducted in accordance with several conditions. First, the constant time complexity threshold is assigned as 0,3. In the other word, a sequence is considered to has constant time complexity (*iff*) as its MAD is lower than 130% of its mean. Second, the calculating time complexity based on the number of processed instructions, input and output instructions are automatically ignored.

When evaluating the effectiveness of sequence correlation toward various time complexities, the researchers do not incorporate all cases in the dataset. Instead, they only incorporate cases from a particular programming language like C++ in this evaluation and calculate the time complexity based on the number of processed instructions. It can evaluate the effectiveness of sequence correlation proposed in the approach since it is the only factor affecting the result. They only incorporate one programming language since the number of processed instructions for each algorithm is always similar to various programming languages. In addition, the number of processed instructions is preferred to time execution as the baseline for this evaluation since the result is not affected by hardware and operating system.

The correlation distribution generated based on this evaluation can be seen in Figure 3. Horizontal axis illustrates algorithms provided in Table 3 whereas the vertical axis represents Pearson correlation between execution and complexity-defined sequence. In Figure 3, an algorithm is considered as highly correlated to constant time complexity with correlation = 1 (*iff*) and the execution sequence is considered as constant. Otherwise, it will be assigned as -1. In general, the time complexity of each case in the dataset is detected correctly since its designated time complexity function always yields the highest correlation value. In other the word, the sequence correlation is quite effective to determine time complexity due to the high accuracy. Despite its high accuracy, the designated correlation value is only slightly higher than other correlation values. This finding is natural since the evaluation only incorporates 10 test cases per algorithm. The limited test cases mean limited points to determine time complexity. It is rather difficult to differentiate complexities with similar sequence pattern. In fact, it may produce more significant difference when the numbers of incorporated test cases are larger.

When evaluating the effectiveness of the approach toward various programming languages, the researchers incorporate two schemes which are generated based on the detection baseline. There are the number of processed instructions and time execution. For convenience, the scheme that relies on the number of processed instructions will be referred as NI-GES where as scheme based on time execution will be referred as TI-GES.

To generate the more precise result, the evaluation uses approximate matching instead of the strict one. Matching score for each case is not only limited to Boolean values (match = 1 or mismatch = 0). Alternatively, its matching score is weighted based on misclassification

Table 3 Standard Time Complexity Algorithm Dataset

ID	Time Complexity	Algorithm	Actual Time Complexity	Input Set (N)
A	T(1)	Converting Currency	1	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
B	T(log N)	Binary Search	3 log N + 2	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
C	T(N)	Sequential Search	N + 1	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
D	T(N log N)	Quick Sort	2(N log N) + 8 log N + 1	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
E	$T(N^2)$	Insertion Sort	$N^2 + 3N + 3$	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000
F	$T(N^3)$	Matrix Multiplication	$N^3 + N^2 + 1$	1, 2, 5, 10, 20, 30, 50, 60, 80, 100
G	$T(2^N)$	Recursive Fibonacci	$2^N + 1$	1, 2, 3, 5, 8, 10, 12, 20, 25, 30
H	$T(3^N)$	3 Digit Combination	$3^N + 1$	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
I	T(N!)	Permutation Sort	N! + 1	1, 2, 3, 4, 5, 6, 7, 8, 9, 10

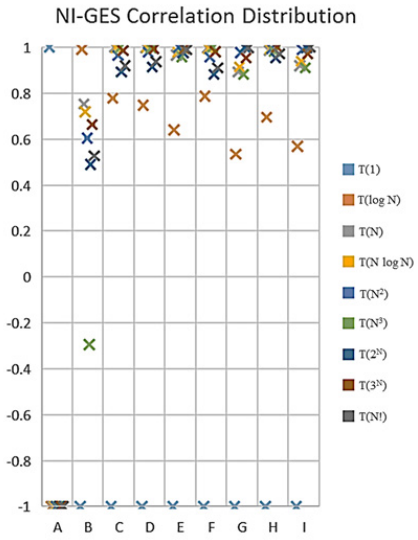

Figure 3 The Correlation Distribution based on Evaluation Dataset

distance. The detail of misclassification-weighted matching score involved in this research can be seen as *r* represents the result of the approach, *e* shows expected result, and distance *(r,e)* is misclassified distance between *r* and *e* toward 9 complexity functions described in Table 3. For example, if it is $r = T(log\ N)$ and $e = T(N^2)$, then the results will be distance $(r,e) = 2$. The more adjacent both *r* and *e* are, the higher the misclassification-weighted matching score is.

$$approx\_match(r, e) = \frac{1}{\text{distance}(r, e) + 1} \quad (1)$$

NI-GES accuracy toward the dataset is 100% according to the result of the first evaluation in Figure 3. Each case per time complexity is detected correctly where as the result is always similar regardless the programming language. On the contrary, TI-GES accuracy is lower and relies heavily on the programming language. The detail of TI-GES accuracy to the evaluation dataset can be seen in Figure 4. Horizontal axis means algorithms provided in Table 3, while the vertical axis is TI-GES accuracy by using misclassified-weighted matching score, and percentage in legend shows the average accuracy per programming language.

Based on Figure 4, Python yields the highest accuracy (73,519%), Java is the intermediate one (62,963%), and C++ has the lowest one (52,778%). When it is associated with time execution, it can be stated that the longer execution time is, the better accuracy it has. Python yields the highest accuracy while taking the longest time execution whereas C++ yields the lowest one while having the shortest execution time. This finding is natural since the program of time execution is heavily depended on hardware and operating system, and taking longer execution time may reduce its drawbacks.

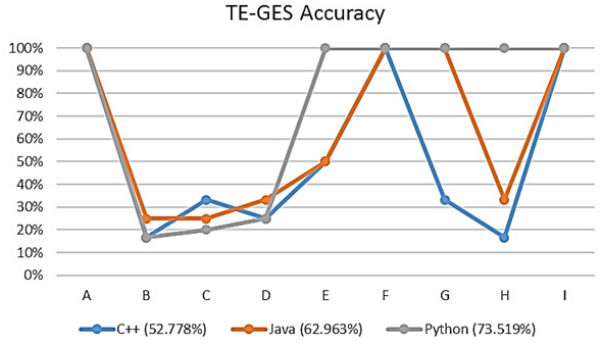

Figure 4 TI-GES Accuracy based on Evaluation Dataset

From the case perspective, all programming languages show the faulty result in the low-progressive time

complexity algorithms (case B to D) and yield the better result in high-progressive ones (case E to I). Case B to D gives faulty result since their generated execution sequences are not patterned correctly due to hardware and operating system dependency. This dependency affects greatly on low-progressive time complexity algorithms since these algorithms only require a short time for executing their program. On the contrary, case E to I has better result since they require a longer time to execute their program. Thus, the impact of hardware and operating system dependency are minimalized. Case A is always detected correctly regardless of programming language since constant time complexity is always checked before hand in the approach.

In general, the number of processed instructions (NI-GES) is more effective to determine time complexity compared to time execution (TI-GES). This finding is based on the respective accuracy in handling the dataset where NI-GES has 100% accuracy, and TI-GES only shows 63,086% in average. When it is discovered further, TI-GES gives lower accuracy due to the dependency on hardware and operating system. It may have the faulty result when time execution required for running an algorithm is low. It means TI-GES cannot determine time complexity correctly when the given algorithm has low time complexity or is implemented in programming languages with the fast running mechanism.

Both NI-GES and TI-GES rely on actual execution. Thus, determining proper test cases for generating execution sequence is important. Test cases should be created carefully, and all of its input should aim for similar time complexity (best, average, or worst case). If the test cases are selected randomly, it may reduce the accuracy since the sequence pattern on generated execution sequence may not show a particular algorithm time complexity.

IV. CONCLUSIONS

In this research, the researchers have proposed Complexitor, an educational tool for learning algorithm time complexity in a practical manner. To determine time complexity, Complexitor requires algorithm implementation and test cases. Algorithm implementation is shown as a single source code whereas test cases are represented as the file text. Time complexity is determined based on Pearson correlation which compares complexity sequence with execution sequence. Complexity sequence is based on the standard complexity functions where the execution sequence is generated with either one of two baselines: the number of processed instructions (NI-GES) and time execution (TI-GES).

According to the evaluation, NI-GES is more effective than TI-GES. It is because TI-GES is heavily affected by hardware and operating system. However, NI-GES requires a considerable amount of time when the algorithm is implemented in a new programming language. Students should provide several programming-related instructions such as counter initialization, counter increment, and counter print.

For further research, the researchers will evaluate the impact of Complexitor qualitatively based on students' necessity. They will conduct a controlled experiment on algorithmic strategy course in their university and evaluate its impact through questionnaire and students' marks. In addition, they also intend to find an optimal threshold to determine the constant time complexity on mathematical manner. As they know, this threshold may be varied regarding the dataset pattern. From the feature perspective, they intend to extend Complexitor with theoretical learning approach so students can learn how to calculate time complexity from both practical and theoretical manner.

REFERENCES

Areias, C., & Mendes, A. (2007). A tool to help students to develop programming skills. In *Proceedings of the 2007 International Conference On Computer Systems and Technologies* (p. 89). ACM.

Buck, D., & Stucki, D. J. (2001). JKarel Robot: A case study in supporting levels of cognitive development in the computer science curriculum. In *The Thirty-Second SIGCSE Technical Symposium on Computer Science Education.* Charlotte.

Carlisle, M. C., Wilson, T. A., Humphries, J. W., & Hadfield, S. M. (2005). RAPTOR: A visual programming environment for teaching algorithmic problem solving. In *The 36th SIGCSE Technical Symposium on Computer science education.* St. Louis.

CeeBot4. (2008). *Learn programming with CeeBot4.* Retrieved November 5th, 2016, from http://www.ceebot.com/ceebot/4/4-e.php

Christiawan, L., & Karnalim, O. (2016). AP-ASD1: An Indonesian desktop-based educational tool for basic data structures. *Jurnal Teknik Informatika dan Sistem Informasi (JuTISI), 2*(1), 21-30.

Cisar, S. M., Pinter, R., Radosav, D., & Čisar, P. (2010). Software visualization: The educational tool to enhance student learning. In *MIPRO, 2010 Proceedings of the 33rd International Convention* (pp. 990-994). IEEE.

Cooper, S., Dann, W., & Pausch, R. (2000). Alice: A 3-D tool for introductory programming concepts. *Journal of Computing in Small Colleges, 15*(5), 107-116.

Debdi, O., Paredes-Velasco, M., & Velázquez-Iturbide, J. Á. (2015). GreedExCol, A CSCL tool for experimenting with greedy algorithms. *Computer Applications in Engineering Education, 23*(5), 790-804.

Guo, P. J. (2013). Online Python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 579-584). ACM.

Halim, S. (2011). *VisuAlgo*. Retrieved May 12th, 2015 from http://visualgo.net/

Halim, S., Koh, Z. C., Loh, V. B., & Halim, F. (2012). Learning algorithms with unified and interactive web-based visualization. *Olympiads in Informatics, 6*, 53-68.

Jonathan, F. C., Karnalim, O., & Ayub, M. (2016). Extending The Effectiveness of Algorithm Visualization with Performance Comparison through Evaluation-integrated Development. In *Seminar Nasional Aplikasi Teknologi Informasi (SNATI)*. Yogyakarta

Laakso, T. R. M. J., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ViLLE tool. *Journal of Information Technology Education*, *7*, 15-32.

Ling, E. (2014). *Teaching algorithms with web-based*

technologies. Singapore: National University of Singapore.

Naps, T. L., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., . . .Velázquez-Iturbide, J. A. (2003). Exploring the role of visualization and engagement in computer science education. In *ITiCSE-WGR '02 Working group reports from ITiCSE on Innovation and technology in computer science education.* New York.

Pearson, K. (1895). Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, *58*, 240-242.

Radosevic, D., Orehovacki, T., & Lovrencic, A. (2009). Verificator: Educational tool for learning programming. *Informatics in Education, 8*(2), 261-280.

Velázquez-Iturbide, J., & Pérez-Carrasco, A. (2009). Active learning of greedy algorithms by means of interactive experimentation. In *ITiCSE '09 Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education.* New York.

Watts, T. (2004). The SFC editor: A graphical tool for algorithm development. *Journal of Computing Sciences in Colleges, 20*(2), 73-85.