PERFORMANCE EVALUATION OF IMPROVED COGNITIVE COMPLEXITY METRIC AND OTHER CODE BASED COMPLEXITY METRICS

Esther Isola [1] ; Stephen Olabiyisi [2] ; Elijah Omidiora [2] ; Rafiu Ganiyu [2]

[1] Osun State University, Osun State Nigeria, Deparment Of Information & Communication Technology
[2] Ladoke Akintola University Of Technology, Oyo State Nigeria, Department Of Computer Science & Engineering

Corresponding Author: Rafiu Ganiyu, raganiyu@lautech.edu.ng

ABSTRACT: Complexity metric is used to estimate various parameters such as software development cost, amount of time needed for implementation and effort required in understanding the software. In this paper, different software complexity models are critically studied and compared. For application, heap sort algorithm is considered. The programs are written in three object oriented languages: C++, C# and Java. Software complexity for each program is found using the four popular Line of Code (LOC), McCabe Cyclomatic Complexity Metric, Halstead Metric and Cognitive model (Improved Cognitive Complexity Metric (ICCM)). The results are compared, according to Halstead Program Difficulty and ICCM, program in C++ has complexity higher than that of program in Java and program in Java has complexity higher than that of program in C#.
KEYWORDS: Software Complexity Metric, Line of Code, Cyclomatic Number, Halstead metric, Cognitive Complexity, Heap sort algorithm.

1. INTRODUCTION

The complexity of software effects on maintenance activities like software testability, reusability, understandability and modifiability. Software complexity is defined as the degree to which a system or component has a design or implementation that is difficult to understand and verify [JK14]. All the factors that make program difficult to understand are responsible for complexity. So it is necessary to find measurements for software to reduce the impacts of the complexity and guarantee the quality at the same time as much as possible. Because of that, the important challenge is how to maintain the software quality in light of the required functionalities. Various metrics of the complexity may be conducted for the software by the developers such as Halstead complexity metric [Hal76], Line of Code or Cyclomatic complexity metric [J+09]. A study was done using 71,917 C/C++ programs to find relations between internal software metrics and metrics of software dependability. It was found that there is a very strong correlation between Lines of Code and Halstead Volume; there is an even stronger correlation between Lines of Code and McCabe‟s Cyclomatic Complexity; and none of the internal software metrics makes it possible to discern correct programs from incorrect ones [Meu07].
The aim of this paper is to critically study and compare four commonly used complexity metrics: LOC complexity, McCabe Cyclomatic Complexity, Halstead and Cognitive metric. It is demonstrated through an example that these metrics provide different complexity measures for the same piece of code, thus making it difficult for the software engineer to choose a suitable complexity metric. The rest of the paper is organized as follow: section 2 presents a review of the software complexity metrics. Section 3 presents complexities of programs written in three object-oriented programming languages using various complexities metrics. A comparison of complexity metrics is presented in section 4 followed by conclusion in section 5.

2. REVIEW OF SOFTWARE COMPLEXITY METRICS

Several methods have been proposed to measure the software complexity. Among the most frequently cited measures are the code based complexity metrics which are: line of code (LOC), McCabe's cyclomatic complexity, Halstead's software metric and Cognitive complexity metrics like Improved Cognitive Complexity Metric (ICCM). These metrics will be discussed briefly in this section.

A. Line of Code (LOC) Complexity

The simplest way to measure the complexity of a program is to count the lines of executable code. There is a strong relationship and connection between complexity and size of code which influences the testability and increases the

implementation and running time [Meu07]. A program with larger LOC value takes more time to be developed. Generally, logical lines of code (LLOC) are more useful as compared to physical lines of code. LOC is a good estimate of the complexity of a program, is easy to implement, and does not require the complex operations and calculations [Jon06]. Moreover, counting lines of code can be transformed from a manual operation to an automated operation. However, it is programmer and language dependent and it does not take into consideration the code functionality [A+16].

B. McCabe's Cyclomatic Complexity Complexity

McCabe defined the cyclomatic number as program complexity. This counts the number of linearly independent paths through a program. First the flow graph of the program is drawn and then the cyclomatic number is found using the following formula [SK10]:

$$M = V(G) = e - n + 2p$$

Where, is the number of edges in the graph, *n* is the number of nodes and *p* is the number of unconnected parts in the graph. It is recommended that no single module has a value of *M* greater than 10. Modules which have a value of greater than 10 are considered as complex modules and require much more testing effort. Those modules should be redesigned to reduce value of *M* [J+09]. Cyclomatic number can be easily computed in the development lifecycle during all phases. It improves the testing process, highlights the best areas of concentration for testing and gives the number of recommended tests for software [SK10]. However, the cyclomatic number presents only a partial view of complexity and can be misleading.

C. Halstead's Software Metric

Halstead model defines a program as a collection of tokens, classified as either operators or operands. He proposed the following formulas to find Program Length, Program Vocabulary, Volume, Difficulty, and Effort [Hal97]:

Program Vocabulary (h) = h1 + h2
Program Length (N) = N1 + N2
Volume (V) = $N \log_2 h$
Potential Volume (v*) = $(2 + h2) \log_2 (2 + h2)$
Program Level (L) = V* / V
Difficulty (D) = V / V*
Effort (E) = E / L
Faults (B) = V / S*

Where *N1* is the number of all operators in the code, *N2* is the number of all operands in the code, *h1* is the number of distinct operators in the code, *h2* is the number of distinct operands in the code, *V** is the minimum volume represented by a built-in function that can perform the task of the entire program and *S** is the mean number of mental discriminations or decisions between errors - a value estimated as 3,000 by Halstead.
Halsted found that complexity increases as vocabulary and length increase. Moreover, complexity increases as volume increases and program level decreases. Modules which do not have program levels close to 1 are too complex. Halstead method is easy to implement, simple to calculate, can be used for any programming language, minimizes rate of errors and maintenance effort. However, there are many shortcomings of this model. It has little or no use as a predictive estimating model. It is based on some unreal and imaginary assumptions which cannot be proven easily. For the large programs it is difficult to count the distinct operators and operands.

D. Cognitive Metric

Cognitive Model proposed Improved Cognitive Complexity Metric (ICCM) to measure the complexity of software [I+16a]; [I+16b]. ICCM is based on number of variables and cognitive weights, for this, every Basic Control Structure is assigned a cognitive weight. CFS is based on cognitive weights. For this, every Basic Control Structure (BC) is assigned a cognitive weight. Either all the BCS's are in a linear layout or some BCS's are embedded in others. For the former, sum of the weights of all *n* BCS's are added and for the later, cognitive weights of inner BCS's are multiplied by the weights of external BCS's [KM06]. Table 1 shows different types of BCSs, the corresponding dedicated weight.

Table 1: Basic Control Structure [KM06]

Category	BCS	CWU
Sequence	Sequence	1
Condition	If-else / Switch For / For-in	2
Loop	While/do…While Functional- call	3
Functional activity	Alert/ prompt throw	2
Exception	try-catch	1

3. METHODOLOGY

Heap sort algorithm is implemented in three object oriented languages which are C++, C# and java. For each program, all the four metrics (LOC, CC,

Halstead and ICCM) are applied and compared. The complexity calculation is demonstrated with Heapsort Algorithm implemented in C# language, given by the following Table 2.

4. RESULTS AND DISCUSSION

The complexity values of the three programs for different complexity measures are summarized in Table 3. It can be seen that heap sort algorithm implemented in C++ language has the highest ICCM and program difficulty values which are 415 and 183 respectively. Hence it is more complex and requires more effort in understanding the programs compared to C# and java language. C# is better compared to the other two programs, in the aspect of program difficulty, CC, LOC and ICCM. This is because the effort required in understanding heap sort algorithm implemented in C# is lesser compared to the other two programs. A graph which covers the comparison between Program difficulty (D), Cyclomatic Complexity (CC), Line of code (LOC) and Improved Cognitive Complexity Metric (ICCM) is plotted in Figure 1. To observe similarities and differences between them a close inspection shows that the metrics are closely related. Figure 2 shows the relative graph between D, CC, LOC and ICCM. It can be observed that LOC, CC, D and ICCM reflects similar trends. In other words, high ICCM values are due to large size, large number of variables and large number of branching structures in a program. For example ICCM has the highest value of 415 for heapsort algorithm implemented in C++ language, which is due to having the maximum line of code 81. Figure 3 shows that if program volume is considered, heapsort algorithm is best implemented in C# and worst implemented in Java language.

Table 2: Complexity Calculation

Complexity Method	C# programs		
LOC	LOC = 25		
McCabe (CC)	CC = 5		
Halstead Method Program Volume (V)= N log2 n Program Difficulty (D) = V/V*	N1 = 131 n1= 19 N2 = 75 n2 = 9 V = 990 D = 79		
Improved Cognitive Complexity Metric (ICCM)	3(ANV) + MNV	CWU	ICCM
1. Functionheapsort (int I)	4	1	4
2. {	0	1	0
3. Var = 1 int I. length	6	1	6
4. End = I = 1	4	1	4
5. Swap	1	1	1
6. Int I = heapify (int I, 1)	8	1	8
7. While (end > 0)	0	3	0
8. {	0	1	0
9. Swap = int [0]	2	1	2
10. Int I [0] = int [end]	6	1	6
11. Int I [end] = swap	6	1	6
12. ….end	0	1	0
13. Int I = siftdown(int 1, 0, end)	10	1	10
14. }	0	1	0
15. Return int I	4	1	4
16. }	0	1	0
17. Function_heapify (int I, 1)	5	1	5
18. {	0	1	0
19. Var start = (1….2)/ 2	2	1	2
20. While (start >= 0)	1	3	3
21. {	0	1	0
22. Int = siftdown (int I, start, I – 1)	8	1	8
23. ---start	1	1	1
24. }	0	1	0
25. Return int I,	4	1	4
	TOTAL:		**75**

Table 3: Complexity values for different complexity measures

Algorithm	Languages	Program Vol (V)	Program Difficulty (D)	Cyclomatic Complexity (CC)	Line of Code (LOC)	ICCM
Heapsort	C#	990	79	5	25	75
Heapsort	C++	1579	183	6	81	415
Heapsort	Java	2098	108	5	71	310

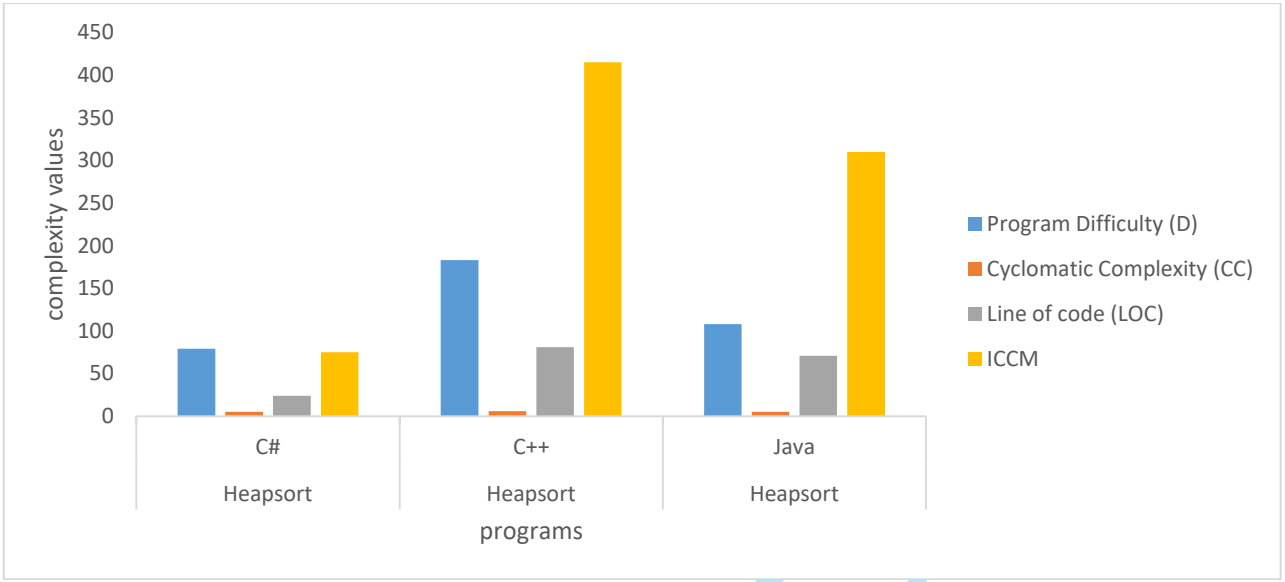

Figure 1: Comparison between ICCM, CC, LOC and D

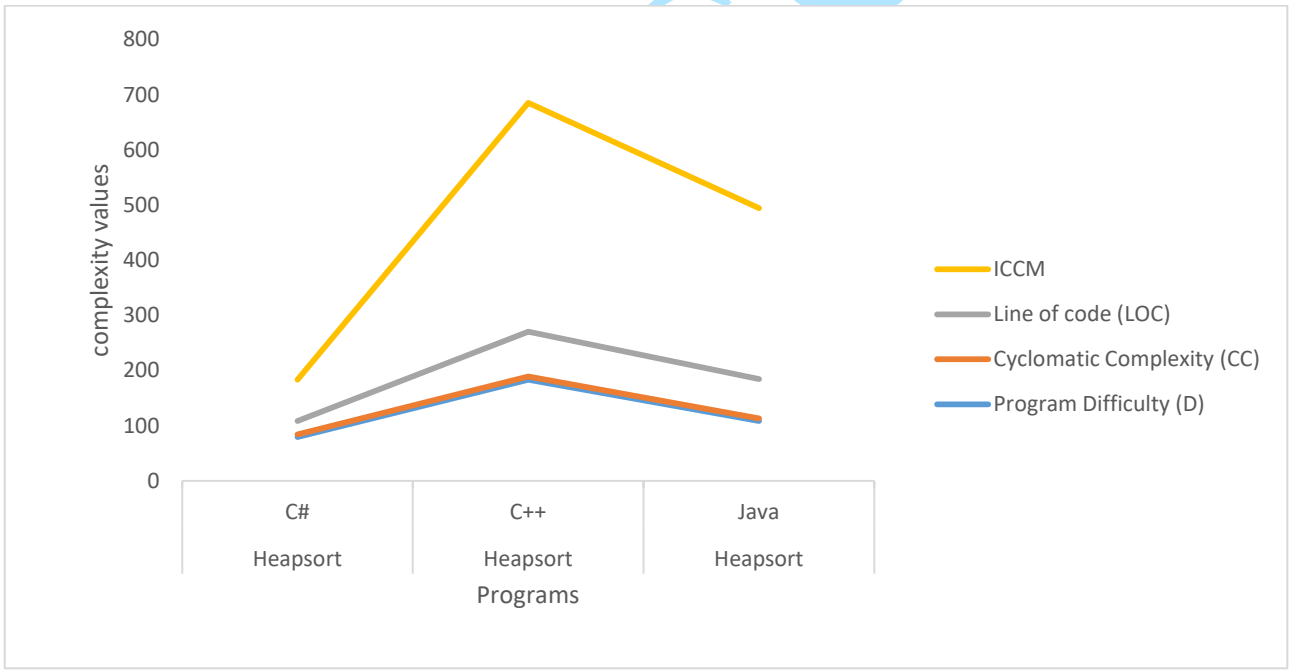

Figure 2: Relative graph between ICCM, CC, LOC and D

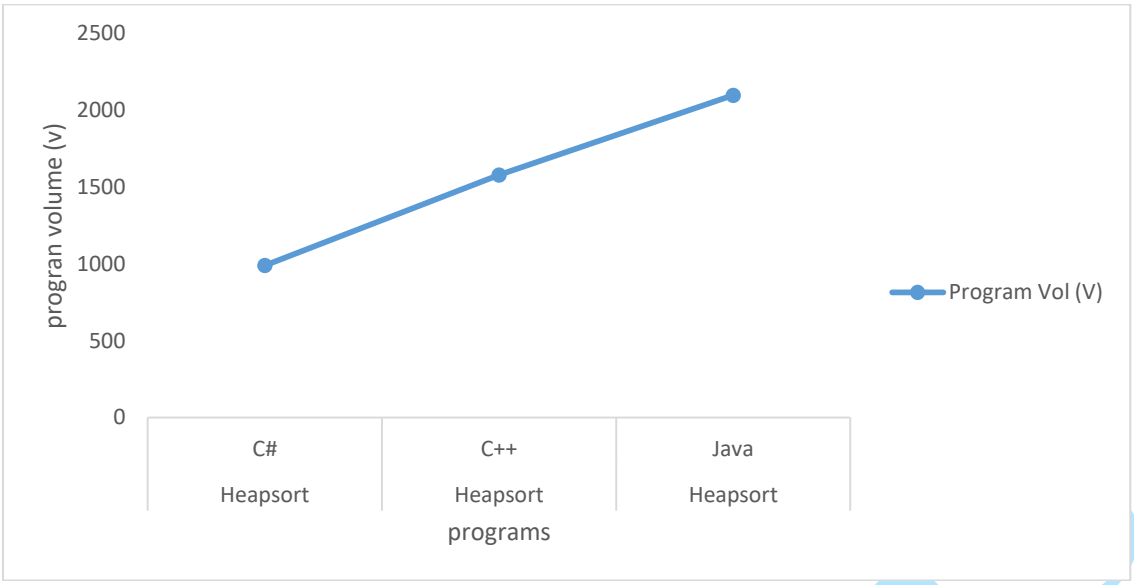

Figure 3: Graph of Program Volume (V) for different implementation of Heapsort Algorithm

CONCLUSION

This research has considered software complexity measure experiment with heapsort algorithm. Heapsort algorithm was studied by computing the Progran Volume (V), the Program Difficulty (D), Cyclomatic Complexity metric (CC), Line of code (CC) and Improved Cognitive Complexity Metric (ICCM) using three different implementation languages (C#, C++ and Java). The Software complexity measurement plays a vital role to reduce the effort required in understanding and maintaining software, and to enhance the effectiveness of testing and software quality. According to Program Difficulty and ICCM, program in C++ has complexity higher than that of program in Java and program in Java has complexity higher than that of program in C#. Thus it is not possible to say which program is more complex because different software metric gives different result. The reason that not all metrics are giving the same results is that each method covers a part and considers some parameters while leaving some others.

ACKNOWLEDGEMENT

The authors wish to thank the authority of Osun State University Osogbo and Ladoke Akintola University of Technology Ogbomoso for enabling environment provided as well as uninterrupted internet services. The moral support of member of faculty is also appreciated.

REFERENCES

[A+16] **Ali Athar Khan, Amjad Mahmood, Sajeda M. Amralla, Tahera H. Mirza** - *Comparison of Software Complexity Metrics*. International Journal of Computing and Network Technology. Vol. 4, No 1, pp 19 – 27, 2016.

[Hal76] **Halstead M. H.** - *Elements of Software Science*, New York: Elsevier North, 1976.

[Hal97] **Halstead M.** - *Elements of software science*, Elsevier North-Holland, 1997.

[I+16a] **Isola E. O., Olabiyisi S. O., Omidiora E. O., Ganiyu R. A., Ogunbiyi D. T., Adebayo O. Y.** - *Development of an Improved Cognitive Complexity Metrics for Object-Oriented Codes*. British Journal of Mathematics & Computer Science. 18(2): pp: 1-11, 2016.

[I+16b] **Isola E. O., Olabiyisi S. O., Omidiora E. O., Ganiyu R. A.** - *An Exploratory Study of Effect of Implementation Languages On Quick Sort Algorithm Using Software Complexity Metrics*. Science Focus Vol. 21 No 2, pp. 1- 6, 2016.

[Jon06] **Jones C.** - *Strength and weaknesses of software metrics*, Version 5, 2006.

[JK14] **Jayanthi B., Krishna Kumari K.** - *Brief study on Software quality metrics and software complexity metrics in Web application*, International Journal of Engineering Sciences & Research Technology, vol. 3, no. 12, pp. 441,444, 2014.

[J+09] **Jay G., Hale J. E., Smith R. K., Hale D., Kraf N. A., Ward C.** - *Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship*, J. Software Engineering & Applications, p. 7, 2009.

[KM06] **Kushwaha D. S., Misra A. K.** - *A modified cognitive information complexity measure of software*, ACM SIGSOFT Software Engineering Notes, vol 31, no. 5, pp. 1-4, 2006.

[Meu07] **van der Meulen J. P.** - *Correlations between internal software metrics and software dependability in a large population of small C/C++ programs*, 18th IEEE International Symposium on Software Reliability Engineering, pp. 203-206, 2007.

[SK10] **Sharma A., Kushwaha D. S.** - *A Complexity measure based on requirement engineering document*, Journal of Computer Science and Engineering, vol. 1, no. 1, pp. 112-117, 2010.