International Journal of Engineering Applied Sciences and Technology, 2021
Vol. 6, Issue 5, ISSN No. 2455-2143, Pages 24-29
Published Online September 2021 in IJEAST (http://www.ijeast.com)

COMPARISON OF SOFTWARE COMPLEXITY OF SEARCH ALGORITHM USING CODE BASED COMPLEXITY METRICS

Bello Muriana
Information technology and resource center,
Kogi State University, Anyigba, Nigeria

Ogba Paul Onuh
Information technology and resource center,
Kogi State University, Anyigba, Nigeria

Abstract: Measures of software complexity are essential part of software engineering. Complexity metrics can be used to forecast key information regarding the testability, reliability, and manageability of software systems from study of the source code. This paper presents the results of three distinct software complexity metrics that were applied to two searching algorithms (Linear and Binary search algorithm). The goal is to compare the complexity of linear and binary search algorithms implemented in (Python, Java, and C++ languages) and measure the sample algorithms using line of code, McCabe and Halstead metrics. The findings indicate that the program difficulty of Halstead metrics has minimal value for both linear and binary search when implemented in python. Analysis of Variance (ANOVA) was adopted to determine whether there is any statistically significant differences between the search algorithms when implemented in the three programming languages and it was revealed that the three (3) programming languages do not vary considerably for both linear and binary search techniques which implies that any of the (3) programming languages is suitable for coding linear and binary search algorithms.

Keywords: Complexity metrics, Searching algorithm, linear search, binary search, ANOVA,

Halstead's complexity metric and McCabe cylomatic complexity metrics.

I. INTRODUCTION:

Software complexity has ushered in a new age in recent years. In the field of computer science there is no universally accepted definition of software complexity most of the definitions are based on Zeus' perspective of software complexity. He opined that "Software complexity" is the level of difficulty in analyzing, maintaining, testing, designing, and modifying software" (Zuse, 1993). The term "software complexity" can be characterized as the primary determinant of software cost, dependability, and performance of software. Software complexity can also be defined as the extent wherein the design or implementation of a system or component is hard to understand and validate. Basili (1980) defines complexity as a measure of the resources consumed by a system while interacting with a software program to complete a task, if the interacting system is a computer, the complexity is determined by the computation's execution time and storage requirements while if the interacting system is a programmer, the difficulty of executing tasks such as writing, debugging, testing, or updating the software is defined as complexity. Software complexity is a vast issue in Software Engineering that has drawn attention of large number of researchers since 1976, and a number of metrics to quantify software complexity have been suggested. This metric is extremely important in software management and plays a significant influence in project success. During the development phases of software, the amount of effort required evaluating requirements, design, code, test, and debug of the system is heavily influenced by complexity. Complexity shows the difficulty in error repair and the effort required to

alter a specific software module during the maintenance phase.

The growing importance of software measurement and metrics has driven the development of new software complexity measurement and software engineering metrics, which are critical for project planning and measurement estimations. Higher quality software has resulted from greater demand for software quality, and quality is now the primary distinction between software solutions. As a result, software designers and developers must take significant steps from the start to review, enhance, and accept software products. Software measurement has been an important aspect in determining the complexity and quality of software in recent years. The paper is organized as follows Section I contains the introduction to Comparison Of Software Complexity of Search Algorithm Using Code Based Complexity Metrics; Section 2 the software complexity metrics review; Section 3 contain finding complexity of software; Section 4, the statistical analysis used to measure the equivalence of comparison groups; Section 5 describes results and discussion; Section 6 the conclusion.

II. SOFTWARE COMPLEXITY METRICS REVIEW

Several methods for determining software complexity metrics have been presented. The line of code (LOC), McCabe's cyclomatic complexity, Halstead's software metric, and the Cognitive weights model are among the most commonly referenced measurements. We will briefly talk about these measurements in the sections that follow.

A. Line of Code (LOC) Complexity

Counting the lines of executable code is the simplest technique to determine a program's complexity. There is a significant link and relationship between code complexity and code size, which affects reliability and increases implementation and running time. (Nystedt S., and Sandros C. 1999). It takes longer to design a program with a higher LOC value. Logical lines of code (LLOC) are generally more valuable than physical lines of code. LOC provides a good measure of a program's complexity because it is simple to construct, and does not necessitate sophisticated procedures and calculations (Jones C., 2006). Furthermore, counting lines of code can be converted from a manual to an automated process. It is, however, programmer and language dependent, and it does not take code functionality into account. (Yu S. and Zhou S., 2010).

B. McCabe's Cyclomatic Complexity Complexity

The cyclomatic number was defined by McCabe as the number of linearly independent pathways that a program is counted (van der Meulen M. J. P., 2007), and it is calculated by generating program's flow graph. The cyclomatic number is calculated using the formula (Sharma A., Kushwaha D.S., 2010).

$$M(C) = V(G) = e - n + 2p \quad (1)$$

Where:
V(G) is the cyclomatic complexity
e represent the number of edges of the graph
n represent the number of nodes of the graph
p represent unconnected parts in the graph. A value of M larger than 10 is not recommended for any single module.

During all phases of the development lifecycle, the cyclomatic number can be simply computed. Cyclomatic metric enhances the testing process, identifies the most important regions for testing, and provides the amount of software tests that should be performed. However the Cyclomatic number, only gives a partial picture of complexity.

C. Halstead Complexity Metric (HCM)

Halstead popularized the term "software science," which refers to the application of scientific methods to investigate the features and structure of software. The Halstead complexity metric was created as a result of this idea. The HCM is determined by the number of operators and operands (Halstead, 1977). The operators are symbols that are used in expressions to define how the alteration will be done. The operands are the basic logic units that must be used to operate the system. Based on these assertions, some variables can be calculated thus;

The length N of P: $N = N1 + N2$ (2)
The vocabulary n of P; $n = n1 + n2$ (3)
The volume V of P: $V = N \times log_2 n$ (4)
The level of L of P: $L = (2 \times n2) / (n1 \times N2)$ (5)
Program difficulty: D of P: $D = (n1/2) \times (N2 / n2)$ (6)
The effort E to generate P is calculated as; $E = D \times V$ (7)

Where n1 is the number of unique operators, n2 is the number of unique operands, N1 is the total occurrences of operators, N2 is the total occurrences of operands and P is the overall program's task. The Halstead technique is simple to implement, compute, and utilize in any programming language. It also reduces the rate of errors and maintenance labor.

D. Weighted Class Complexity

Two metrics were proposed by Mistra and Akman (2008) for inheritance and class features of the object oriented code. Both metrics are based on cognitive weights. For including the inheritance property of the object oriented code, the authors first suggested calculating the weight of individual method in a class by associating a number (weight) with each member function (method), and then add all the weights of all the methods, this is the weight (complexity) of a specific class object. Depending on the architecture, there are two cases for determining the total complexity of the complete system (if the system is composed of more than one class or object):

i. if the classes' objects are in the same level then the weights of the classes' objects are added
ii. If they are subclasses of their parent, their respective weights are multiplied.

If the object-oriented code has m levels of depth and level j includes n classes, the cognitive code complexity (CCC) of the system is: $CCC_{j=1} \prod_{k=1}^{m} [\Sigma^{n} CC_{jk}]$ 8

The second metric introduced by Misra and Akman (2008) is based on the idea that the complexity of a single method depends on both attributes and complexity of the method. Weighted Class Complexity (WCC) was proposed by the authors and is given as:

$$WCC = N_a + \sum_{p=1}^{s} MC_p \qquad 9$$

Where Na is the total number of attributes and MCp is the complexity of pth method of the class. If an object-oriented code has y classes, the total complexity of the code is equal to the sum of individual class weights. Total Weighted Class Complexity is

$$= \sum_{x=1}^{y} WWC_x \qquad 10$$

Both metrics are used in a modified and enhanced form in the proposal, Mistra and Akman (2008).

2. Complexity Of Programs

In this paper, the complexity of various languages implementation was determined using the following steps:

i. Line of Code (LOC) counts line of codes that do not contain comments

ii. McCabe method (MC): using cyclomatic complexity method MC = V(G) = e – n + 2p

iii. Program difficulty (D): using Halstead method D of P is D = (μ1 ÷2) * (N2 ÷ μ2)

Three complexity metrics were applied to linear search and binary search algorithm that are implemented in three object oriented languages: C++, Python and Java. For each program we used line of code, cyclomatic and halstead to find the complexity, hence the three metrics were compared. The Python, Java and C++ code for linear search algorithm are given in figure 1, figure 2 and figure 3 respectively while the flow graph for the languages were also prepared but are not included here because of lack of space. Software complexity metrics were calculated and the results presented in tables 1 and 2. LOC has the highest values for both linear and binary search algorithms when implemented in C++, Python has the lowest value of complexity for all the variations of different measures except with McCabe Cyclometric metric.

Complexity Values			
	Line of code metrics	McCabe Cyclomatic number	Halstead Metrics (Program Difficulty)
Python	15	7	19.3
Java	27	7	39.9
C++	29	6	28.3

Table 1: Comparison of the metrics for linear search algorithm

Complexity Values			
	Line of code metrics	McCabe Cyclomatic number	Halstead Metrics (Program Difficulty)
Python	23	5	31.3
Java	37	4	52.2
C++	40	4	42.3

Table 2: Comparison of the metrics for Binary search algorithm

```
lst = []
items = int(input("Enter the number of items: "))
for n in range(items):
   numbers = int(input("Enter the %d number: " %n))
   lst.append(numbers)
keyValue = int(input("Enter number to search for: "))
found = False
for i in range(len(lst)):
   if lst[i] == keyValue:
      found = True
      print("%d found at location %d" % (keyValue, i))
      break
if not found:
   print("%d is not in list" % keyValue)
input()
```

Fig1: Python code for linear search algorithm.

```
import java.util.Scanner;
public class linearsearch{
  public static void main(String args[])
  {
    int i, len,  keyValue,  items[];
    Scanner input = new Scanner(System.in);
    System.out.println("Enter number of items  ");
    len = input.nextInt();
    items = new int[len];
    System.out.println("Enter " + len + " items   ");
    for (i = 0; i < len; i++)
    {
      items[i] = input.nextInt();
    }
    System.out.println("Enter the search value  ");
    keyValue = input.nextInt();
    for (i = 0; i < len; i++)
    {
      if (items[i]== keyValue)
      {
        System.out.println(keyValue +" is present at
location "+(i));
         break;
      }
    }
    if (i == len)
      System.out.println(keyValue + " doesn't exist");
  }
}
```

Fig 2: Java code for linear search algorithm.

```
#include<iostream>
using namespace std;
int main() {
cout<<"Enter The Size Of items:   ";
int items;
cin>>items;
int array[items], keyValue,i, n;
// Taking Input In Array
 for(n=0; n<items; n++){
 cout<<"Enter "<<n<<" Element: ";
 cin>>array[n];
 }
 cout<<"Enter KeyValue to Search: ";
 cin>>keyValue;
   for(i=0;i<items;i++){
    if(keyValue==array[i]){
  cout<<"Key Found At Index Number :  "<<i<<endl;
  break;
    }
 }
if(i != items){
cout<<"KEY FOUND at index :  "<<i;
}
else{
cout<<"KEY NOT FOUND in Array  ";
}
   return 0;
}
```

Fig 2: Java code for linear search algorithm.

3. Statistical Analysis

Analysis of Variance (ANOVA) was used to generate inferential judgments in experimental design studies to assure the equivalence of comparison groups even when the number per group vary across the group. Therefore statistical analysis carried out used ANOVA at 0.05 significant levels for values obtained.

Tables 3 and 4 show the ANOVA table for the search algorithms and it was discovered that $f_{0.05}$ 2, 6 = 5.14 > 0.56 for linear search and $f_{0.05}$ 2, 6 = 5.14 > 0.30 for binary search, since the Ftable exceeds the Fcalculated for both linear and binary search we accept the null hypothesis Ho, therefore there is significant relationship between the metrics and the programming languages for linear and binary search techniques.

Source of variation	Sum of Squares (SS)	Degree of freedom (DF)	Mean squares (MS)	F
Between groups	182.2	2	91.1	0.56
Error (Residual)	969	6	161.6	
Total	1151.2	8		

Table 3: ANOVA table for linear search

Table 4: ANOVA table for binary search

Source of variation	Sum of Squares (SS)	Degree of freedom (DF)	Mean squares (MS)	F
Between groups	272.2	2	136.1	0.30
Error (Residual)	2754.8	6	459.1	
Total	3027	8		

III. RESULTS AND DISCUSSION

A. Complexity of Various Implementation of Linear Search

There are significant disparities in the implementation complexity of the various languages, as seen in table 1 shows a comparison of the object-oriented languages of Python, Java and C++ using linear search as a case study for comparison. The figure shows that the length (in lines) of a program written in Python is smaller than that of a program written in Java and C++. This means that Python is less difficult than Java and C++ if measure with LOC. The McCabe method of C++ language is less than that of Python and Java because the implementations are based on the same number of steps and decision points and thus have the same value for cyclomatic complexity, the program difficulty using Halstead method for Python language is less than that of Java and C++.

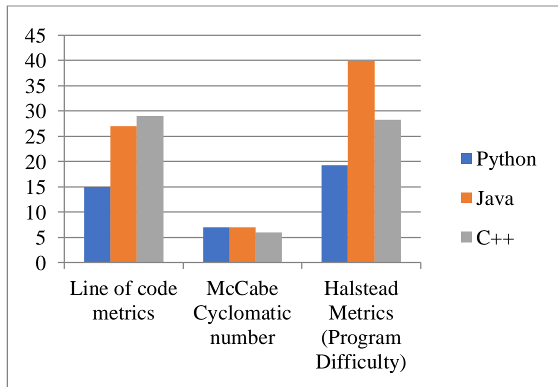

Fig 4: A comparison of the complexity of the Object Oriented Languages Python, Java and C++ for linear search algorithm

B. Complexity of Various Implementation of Binary Search

There are also significant disparities in the implementation complexity of the various languages, as illustrated in table 2. The figure depicts a comparison of the object-oriented languages Python, Java and C++ by employing binary search. The table shows that the LOC of Python is less than that of Java and C++ programs which implies that Python has less complexity than Java and C++. The McCabe method of Python is greater than that of Java and C++ while Java and C++ have the same value for McCabe method. The program difficulty of Python is less than that of Java and C++, in this case Python has less complexity than Java and C++ using binary search.

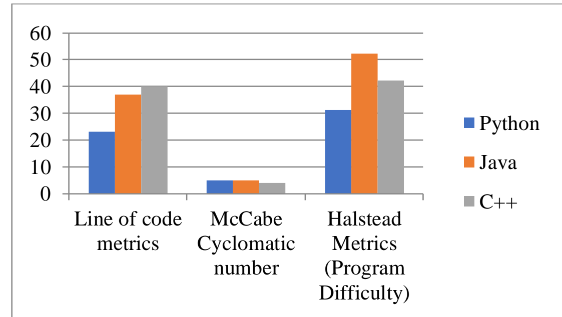

Fig 5: A comparison of the complexity of the Object Oriented Languages Python, Java and C++ binary search algorithm

IV. CONCLUSION

It was discovered that the McCabe method has relatively insignificant complexity values for linear search in Python, Java and C++, with the value of C++ language being six (6) while Python and Java being seven (7). In binary search, the calculated complexity with the McCabe approach is higher for Python while Java and C++ have the same values. Further statistical research of Analysis of Variance (ANOVA) revealed that the three (3) languages do not differ substantially for both linear and binary search methods. As a result, it can be stated that any of the three (3) programming languages is good to code linear search and binary search algorithms.

V. REFERENCES

[1] Banker R.D., Srikant M.D., Kemerer C.F., and Zweig D. (1993) "Software complexity and maintenance cost", Communications of the ACM, Vol. 36, No. 11, (pp. 81–94).

[2] Gill G.K., and Kemerer C.F. (1991) "Cyclomatic complexity density and software maintenance productivity", IEEE Transactions on Software Engineering, Vol. 17, No.12, (pp. 1284–1288,1991).

[3] Halstead M. H. (1977) "Elements of Software Science, Operating and Programming Systems Series", Elservier Computer Science Library North Holland N. Y. Elsevier North-Holland, Inc. ISBN 0-444-00205-7.

[4] Jones C. (1998), “Strength and weaknesses of software metrics”, University of Magdeburg, 3(1998)1, (pp. 35-44)

[5] Kaur H., and Verma N. (2016) “Software Complexity Measurement: A Critical Review”, International Journal of Engineering and Applied Computer Science (IJEACS) Volume: 01, Issue: 01 ISBN: 978-0-9957075-0-4

[6] McCabe, T.H. (1976) “A Complexity Measure‖, IEEE Transaction on Software Engineering”, SE – 2, 6, pp. (308 – 320).

[7] Milutin A. (2009): "Software code metrics", (Online: accessed on 2010-06-21 from Introduction to Algorithms).

[8] Munson J. C., and Khoshgoftaar T. M. (1992) “The detection offault-prone programs”, IEEE Transactions on Software Engineering, Vol. 18, No. 5, (pp. 423–433)

[9] Olabiyisi S., Omidiora E and Sotonwa K (2013) “Comparative Analysis of Software Complexity of Searching Algorithms Using Code Based Metrics”, International Journal of Scientific & Engineering Research, Volume 4, Issue 6, ISSN 2229-5518.

[10] Sharma A., and Kushwaha D.S. (2010) “A Complexity measure based on requirement engineering document”, Journal of Computer Science and Engineering, vol 1, No.1, (pp. 112-117).

[11] Van der Meulen M. J. P. (2007) “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).

[12] Yu S., and Zhou S. (2010) “A survey on metric of software complexity”, IEEE International Conference on Information Management and Engineering (ICIME), (pp. 352-356)