Halstead Complexity Analysis of Bubble and Insertion Sorting Algorithms.

T.R. Awode[1]; D.D. Olatinwo[2]; O. Shoewu[3]; S.O. Olatinwo[1]: O.O. Omitola[1]; and Mary Adedoyin[3]

[1]Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, Ogbomoso, Nigeria..
[2]Department of Mathematical & Computer Science, Ekiti State University, Nigeria.
[3]Department of Electronic and Computer Engineering, Lagos State University, Epe, Nigeria.

E-mail: tolulopedelight@yahoo.com

ABSTRACT

In this paper, the implementation and analysis of two sorting algorithms, namely, bubble sort and insertion sort, based on Halstead complexity metrics have been discussed. The Halstead complexity approach considers the mathematical relationship among the variables. The two selected sorting algorithms have been implemented in MATLAB and compared. The efficiency of each of the algorithms using Halstead parameters in handling various sorting tasks has also been discussed.

(Keywords: Halstead complexity, bubble sort, insertion sort)

INTRODUCTION

Computer algorithms are often analyzed for a number of reasons that include estimation of the run time or the storage requirements which an algorithm needed to process a particular input. Computer memory and time are key resources which users often sought for simultaneously. A good computer analysis finds the bottlenecks in a program, by determining the time spent by each module in a program; that is, sections of a program where most of the time is spent. For instance, computational complexity theory is used to investigate the problems associated with the amount of resources, such as time, needed to execute an algorithm the inherent difficulty in providing efficient algorithms for specific computational problems.

A typical question of the theory is, ‘‘as the size of the input to an algorithm increases, how do the running time and memory requirements of the algorithm change and what are the implications and ramifications of that change.’’ In other words, the theory investigates the scalability of computational problems and algorithms. In particular, the theory places practical limits on what computers can accomplish.

The time complexity of a problem is the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm. For example, a problem that is n-bits long can be solved in n2 steps. In this case, the problem has a time complexity of order n2.

The Big O notation is generally used in measuring the complexity of algorithms. If a problem has a time complexity O(n2) on one typical computer, then it will also have complexity O(n2) on most other computers, therefore, this notation allows us to generalize away from the details of a particular computer [1].

The space complexity of a problem is a related concept that measures the amount of space or memory required by the algorithm. Space complexity is also measured with Big O notation. The existing software complexity measures such as McCabe, Cyclomatic complexity, line of code complexity, and Halstead metric are supposed to cover the correctness, effectiveness, and clarity of software and to provide good estimates of the parameters out of the proposed measure.

Selecting a particular complexity measure is a problem as every measure has its own advantages and disadvantages. There are number of ways to quantify complexity in a program as the best metric, which provide such features as [16] and Cyclomatic number [11]. This therefore necessitates the need to develop a method to measure the software complexity which combines both the quantitative and analytical approaches in general together.

LITERATURE REVIEW

Complexity Theory

In information systems, complexity is used to describe the efficiency of a system in terms of time and computational capabilities in solving a computer a problem such as sorting. McCall *et al*. (1977), describes complexity as the relationship between set of data, data structures, data flow and the algorithm being implemented". It measures the degree of decision making logic within the system. Beizer states that "using only our intuitive notion of software complexity, we expect that more complex software will cost more to build and test and will have more bugs" [4]. Tourlakis (1984) study distinguished between two classes of complexity measure that is dynamic and static. Dynamic complexity measures the amount of 'resources' consumed during a computation. Static complexity measures on the other hand may be size (e.g., program length) or the structural complexity (e.g., level of nesting of do-loops) of an algorithm's description.

Ramamoorthy (1985) states that software complexity is the degree of difficulty in analysis, testing, design, and implementation of software. We will not attempt to attach a single number to software complexity. Instead, we discuss the complexity of individual characteristic of software'. Jones (1986) in his discussion on measuring programming complexity identities 'two logically distinct tasks: (i) measuring complexity of the problem that is the functions and data to be programmed; and (ii) measuring the complexity of the solution of the problem, that is the software itself. Shepperd writes in 1988 that complexity is a metaphysical property and thus not directly measurable. What is required is a means to link the behavior of the product characteristics that are measurable.

Banker *et.al.* (1989) states that software complexity refers to the extent to which a system is difficult to comprehend, modify and test, not complexity of the task which the system is meant to perform; two systems equivalent in functionality can differ greatly in their software complexity. They notice that, most complexity metrics proposed confound complexity of a program with its length. They also propose to measure length-independent complexity metrics by measuring 'density of decision making' and 'density of branching' within a program. Gill and Kemerer (1991) state that "the high correlation of cyclomatic complexity with lines of codes is given as reason for proposing a transformed metric 'complexity density' defined as the ratio of cyclomatic complexity to thousand lines of code".

Fenton (1992) states that complexity is commonly used as a term to capture the totality of all internal attributes. When people talk of the need to control complexity what they really mean is the need to measure and control a number of internal (structural) product attributes. He also states that 'there appears to be three such distinct (orthogonal and fundamental) attributes of the software" length, functionality and complexity of the underlying problem which the software is solving.

Software Quality and Complexity Metric

Software quality is closely related with testing and measurement. Fenton et al. (1997*)* defines measurement as follows; "Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined unambiguous rules." Testing techniques tend to find defects, bottlenecks and weaknesses of a software system. Measurement aims to find the complexity in order to understand the effectiveness of the software's code.

Requirement to improve the software quality is the prime objective, which promotes research projects on software metrics technology. It is always hard to control the quality if the code is complex they are hard to review, test, maintain and manage [3]. As a consequence, those handicaps increase the maintenance cost and the cost of the product. Due to these reasons, it is strongly recommended that the complexity of the code should be controlled from the beginning of the software development process.

Since this research is focused on cognitive complexity, it is worthy of mentioning that complexity decreases the comprehensibility and the complexity of software. Some of the factors that affect the procedural complexity are variables and structure for example. Some of the factors that affect the object oriented (OO) complexity are attributes, structures, and classes. Thus, in order to conceive the complexity of multi-paradigm code, the complexity factors of both of the paradigms should be considered, since multi-paradigm includes the features of both procedural and OO paradigms.

Quality of Software

There are several quality attributes of software such as security, performance, reusability, availability, testability, correctness, maintainability, reliability, integrity, and many others [2]. To achieve some of those quality attributes, complexity should be reduced. For example, to be able to test software easily it is necessary that the software is not complex. Otherwise, the testing process will be harder and thus the cost will be higher. What makes software quality assurance unique is product complexity, visibility, and development process. Actually, complexity of software products has been observed for decades. Complexity of software product is much higher than that of other industrial products. Visibility is another difficulty of software quality assurance, since other industrial products are visible but software products are not visible until the end is reached. Software development process differs with its development methodologies and difficulties in finding and removing defects [9]. Similarly, Hughes and Cotterell (2006) state that intangibility increases criticality of software, and accumulating defects during development process make the software quality unique. Furthermore, software needs to be measured in order to understand its quality. Otherwise, it may not be possible to make an effective project management. One of the most important and effective tools in assessing software quality are to use complexity metrics explained in the following section.

According to Galin (2004), software quality could be broken down into six major characteristics, which are functionality, reliability, usability, efficiency, portability, and maintainability and these quality characteristics are split into a number of quality sub-characteristics. The above mentioned product quality characteristics can be divided into two different sets: external and internal.

Customers care about external quality characteristics such as: functionality, reliability, usability, efficiency, flexibility, friendliness, simplicity, etc simply because these are the characteristics which can easily be seen by the use of the product. On the other hand, developers care about internal quality characteristics such as: maintainability, portability, reusability, testability, etc., because these characteristics relate to their development efforts.

McCall et al. (1977) started with a volume of 55 quality characteristics which have an important influence on quality, and called them "factors". For reasons of simplicity, McCall then reduced the number of characteristics to efficiency, accuracy, interface facility, re-usability, maintainability, testability, reliability, usability, flexibility, integrity, and transferability.

Related Works

During the 60th and 70th classic and important metrics, Line of Codes (LOC), Halstead Complexity Metric (HCM) and Cyclomatic Complexity Metric (CCM) were invented and considered to measure the software in three different aspects, correspondingly, the length, the volume, and the structure. With the development of the software design method, some metrics, which aim at special programming method, such as the Object-Origin programming and Aspect-programming have been introduced.

Chidamber and Kemerer Metrics

Chidamber and Kemerer (1994) proposed a metric suite that offers informative insight into whether developers are following object oriented principals in their design. They claim that using several of their metrics collectively helps managers and designers to make better design decision.

Chidamber and Kemerer (1994) metrics, have generated a significant amount of interest and are currently the most well-known suite of measurements for object oriented software. Chidamber and Kemerer (1994) proposed six metrics viz;

i. **Weighted Methods per Class (WMC):** This metric measures the complexity of a class. Its value is computed as the sum of the complexity of each individual method of the class. For the sake of simplicity, we consider all methods of a class to be equally complex. Thus the value of WMC represents the number of methods of the class.

ii. **Depth of Inheritances tree (DIT):** In their study, this metric measure the number of the ancestors of a class.

iii. **Number of children (NOC):** This metric measures the number of direct descendants of a class.

iv. **Response for a Class (RFC):** This metric measures the number of methods that can be potentially executed in response to a message received by an object of a class.

v. **Lack of cohesion in Methods (LCOM):** This metric measures the number or pairs of methods of a class that do not share any instance variables, minus the pairs of methods that do. The value is zero when the subtraction yields a negative result.

vi. **Coupling between Object Classes (CBOC):** This metric measure the number of other classes used by a class. A class uses another class if one of its members uses a member of the other class.

Weighted Class Complexity

Mistra and Akman (2008) proposed two metrics 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), then we simply add all the weights of all the methods. This gives the complexity (weight) of a single class/object. There are two cases for calculating the whole complexity of the entire system.

If the system consists of more than one class or object depending on the architecture:

i. If the classes' objects are in the same level then their weights are added.

ii. If they are subclasses or children of their parent then their weights are multiplied

If there are m levels of depth in the object oriented code and level j has n classes then the cognitive code complexity (CCC) of the system is given by:

$$CCC_{j=1} = \prod_{k=1}^{m}\left[\sum^{n} CC_{jk}\right] \quad \textbf{(1)}$$

The second metric proposed by (Mistra *et al)* is based on the theme that complexity of a single class depends on attributes and as well as on the complexity of the methods. Accordingly, the authors suggested Weighted Class Complexity (WCC) as:

$$WCC = N_{a+}\sum_{p=1}^{s} MC_P \quad \textbf{(2)}$$

Where: Na is the total number of attributes and MC_p is the complexity of p^{th} method of the class.

If there are *y* classes in an object oriented code, then the total complexity of the code is given by the sum of weights of individual classes. That is Total Weighted Class Complexity:

$$TWCC = \sum_{x=1}^{y} WCC_X \quad \textbf{(3)}$$

Lines of Code (LOC)

The line of codes (LOC) is generally the count of the lines in the source code of the software. Usually, LOC only considers the executable sentence. LOC is independent of what program language is used. The LOC evaluates the complexity of the software via the physical length. LOC is based upon rules; the relationship between the count of code lines and the bug density, the independence between the bug density and the program language. Also sometimes, the LOC is estimated by other factor.

The original purpose the development of LOC was to estimate man-hours for a project [18]. Some types of LOC include the following:

i. **Lines of Code (LOC):** It is obvious from its name that it counts the number of lines which are commented in source code. Some developers write code statement and comment on a same physical line. In such cases this metric can be further defined easily.

ii. **Kilo Lines of Code (KLOC):** It is LOC divided by 1000

iii. **Effective Lines of Code (ELOC):** It only counts the lines that are not commented, black, standalone braces or parenthesis. In a way this metric presents the actual work performed.

iv. **Logical Lines of Code (LLOC):** This metric shows the count of logical statements in a program, it only counts the statements which end at semi-colon. This definition of metric is only applicable for language like C or Java, but for languages like Haskell this metric won't work.

v. **Multiple Line of codes (MLOC):** It contains several separate instructions, multiple line or code like million lines of code.

vi. **Comment to Code Ratio:** is a derived metric from ELOC and Line of Comment metric. This simple and easy to compute metric can provide hint for "understandability of the code". It can be obtained as follows: There are more possible variants of LOC, for example counting blank lines or white spaces, etc.

Halstead Complexity Metric

In 1977, Maurice Howard Halstead introduced the concept of software science. He began to use scientific methods to analyze the characteristics and structure of the software. The idea resulted in the introduction of the Halstead Complexity Metric (HCM). The HCM is calculated on the count of the operators and operands [11]. The operators are symbols used in expressions to specify the manipulation to be performed. The operands are the basic logic unit to be operated. The HCM measures the logic volume of the software. Firstly, the HCM use the following parameters:

$\mu 1$ = the number of unique operators
$\mu 2$= the number of unique operands
N1= the total occurrences of operators
N2 = the total occurrences of operands

From these statements, some indicators can be calculated:

The length N of P:N = N1 + N2 **(4)**

The vocabulary μ of P; $\mu = \mu 1 + \mu 2$ **(5)**

The volume V of P: V = N* log 2 (μ) **(6)**

The level L of P:L = (2 ÷ $\mu 1$) * log2 (μ) **(7)**

The program difficulty: D of P:D = ($\mu 2$÷N2) * (N2 ÷ $\mu 2$) **(8)**

The effort E to generate P is calculated as: E= D*V **(9)**

Error Estimate: B = V/X* **(10)**

Programming Time: T = E/18 **(11)**

Number of Delivered Bugs: B = $E^{(2/3)}$/3000 **(12)**

The V* is the software's ideal volume:

V* = ($\mu 1$ N2 ÷ 2 $\mu 2$) (N1 + N2) log 2 ($\mu 1 + \mu 2$) **(13)**

Equation 13 is commonly used to estimate the V*. The X* means the programmer's ability. Halstead sets X* for a fixed value of 3000.

McCabe Cyclomatic Complexity Metric

Based upon the topological structure of the software, Thomas J. McCabe introduced a software complexity metric name McCabe Cyclomatic Complexity Metric. As described by McCabe, the primary purpose of the measure is to identify software modules that will be difficult to test or maintain [16]. The nodes on the graph correspond to the code lines of the software, and a directed edge connects two nodes if the second node might be executed immediately after the first one. If the conditional evaluation expression is composite, the expression should be broken down. For example, the expression "if (cl &c2) {}" should be treated as "if (cl) {if(c2) { } }". The control flow graph of a module has one and only one entry node and exit node. If one control flow graph has edges and n nodes.

MC = V (G) = e-n+2p

where:

V (G) is the cyclomatic complexity

e is the number of edges of the graph

n is the number of nodes of the graph and

p is the number of connected components.

Sorting Algorithms

In this study, the common types of sorting algorithm, namely the bubble sort and the insertion sort algorithms were studied, compared and analyzed.

Bubble Sort

The goal of this type of sorting algorithm is to sort an array of elements using the bubble sort algorithm. The elements must have a total order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers. The bubble sort is generally considered to be the simplest sorting algorithm. Because of its simplicity and ease of visualization, it is often taught in introductory computer science courses. Because of its abysmal O(n^2) performance, it is not used often for large (or even medium-sized) datasets.

The bubble sort works by passing sequentially over a list, comparing each value to the one immediately after it. If the first value is greater than the second, their positions are switched. Over a number of passes, at most equal to the number of elements in the list, all of the values drift into their correct positions (large values "bubble" rapidly toward the end, pushing others down around them). Because each pass finds the maximum item and puts it at the end, the portion of the list to be sorted can be reduced at each pass. A Boolean variable is used to track whether any changes have been made in the current pass; when a pass completes without changing anything, the algorithm exits [17].

Insertion Sort

If the first few objects are already sorted, an unsorted object can be inserted in the sorted set in proper place, this is called insertion sort. An algorithm consider the elements one at a time, inserting each in its suitable place among those already considered (keeping them sorted) as described in Figure 1. Insertion sort is an example of an incremental algorithm; it builds the sorted sequence one number at a time. This is perhaps the simplest example of the incremental insertion technique, where we build up a complicated structure on n items by first building it on n − 1 items and then making the necessary changes to fix things in adding the last item. The given sequences are typically stored in arrays. Also, the numbers are referred to as keys**.** Along with each key may be additional information, known as satellite data**.**

Algorithm: Insertion Sort

It works the way you might sort a hand of playing cards:

i. We start with an empty left hand [sorted array] and the cards face down on the table [unsorted array].

ii. Then remove one card [key] at a time from the table [unsorted array], and insert it into the correct position in the left hand [sorted array].

iii. To find the correct position for the card, we compare it with each of the cards already in the hand, from right to left.

Note that at all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

Complexity Analysis of Insertion Sort

Insertion sort's overall complexity is O(n^2) on average, regardless of the method of insertion. On the almost sorted arrays insertion sort shows better performance, up to O(n) in case of applying insertion sort to a sorted array. Number of writes is O(n^2) on average, but number of comparisons may vary depending on the insertion algorithm. It is O(n^2) when shifting or swapping methods are used and O(n log n) for binary insertion sort.

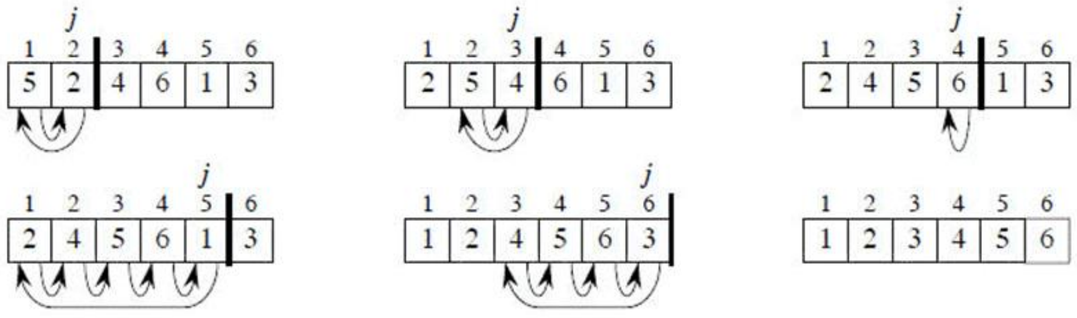

Figure 1: Insertion Sort Analysis Diagram [17].

From the point of view of practical application, an average complexity of the insertion sort is not so important. As it was mentioned above, insertion sort is applied to quite small data sets (from 8 to 12 elements). Therefore, first of all, a "practical performance" should be considered. In practice insertion sort outperforms most of the quadratic sorting algorithms, like selection sort or bubble sort [17].

RESEARCH METHODOLOGY

Approach

The considered sorting algorithms were implemented in MATLAB. For the purpose of scheduling, analysis, and reporting, Halstead metrics were used. The Halstead metrics are simply used to measure and interpret tokens. Tokens can be described as the smallest units of a text which is recognized by a compiler.

Keywords

The following are regarded as Halstead keyword:

break

case

continue

default

do

else

for

goto

if

return

sizeof

switch

while

etc.

Table 1: Halstead Operators.

Halstead Operators
(
[
.
-
>
++
--
Sizeof
&&
%
!=
=
||
==
<
+=
*=

In this study, the following tokens are considered Halstead operands:

i. identifiers,
ii. typedef name types,
iii. numerical constants, and
iv. strings.

A label and its terminating colon do not count, as they are comments according to Halstead. In addition, function headings, including the initializations included in them, do not count.

Halstead Parameters

The basic Halstead parameters used in this study are:

i. Unique operators (c1): the number of unique occurrences of Halstead operators in the program,

ii. Unique operands (c2): the number of unique occurrences of Halstead operands in the program,

iii. Total operators (C1): the total number of Halstead operators,

iv. Total operands (C2): the total number of Halstead operands.

v. Halstead program length: the total number of operator occurrences and the total number of operand occurrences.

$C = C1 + C2$ **(1)**

vi. Halstead vocabulary: the total number of unique operator and unique operand occurrences.

$c = c1 + c2$ **(2)**

vii. Program volume: proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The parameters V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.

$Pvol = C * \log2(c)$ **(3)**

viii. Programming time: shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language.

$T = E / (f * S)$ **(4)**

The concept of the processing rate of the human brain, developed by the psychologist John Stroud, is also used. Stoud defined a moment as the time required by the human brain requires to carry out the most elementary decision. The Stoud number S is therefore Stoud's moments per second with: $5 <= S <= 20$. Halstead uses 18.

Stroud number S = 18 moments / second

seconds-to-minutes factor f = 60

RESULT AND DISCUSSION

In this study, two sorting algorithms were considered and implemented in MATLAB. Tables 2 and 3 describe the parameters used for the simulation. The bubble sort parameters in Table 2 were implemented in MATLAB and the analysis results are presented in Figure 2. Also, the insertion sort parameters in Table 3 were implemented in MATLAB and the analysis results are presented in Figure 3. Figure 4 presents the overall results for both the bubble sort and the insertion sort algorithms for comparison purposes.

Table 2: Bubble Sort Implementation Parameter.
s

Metrics	Value
c1	13
c2	7
C1	21
C2	30
LOC	27
Pvoc	17
P	43
Cd	21.70
PVol	41.23

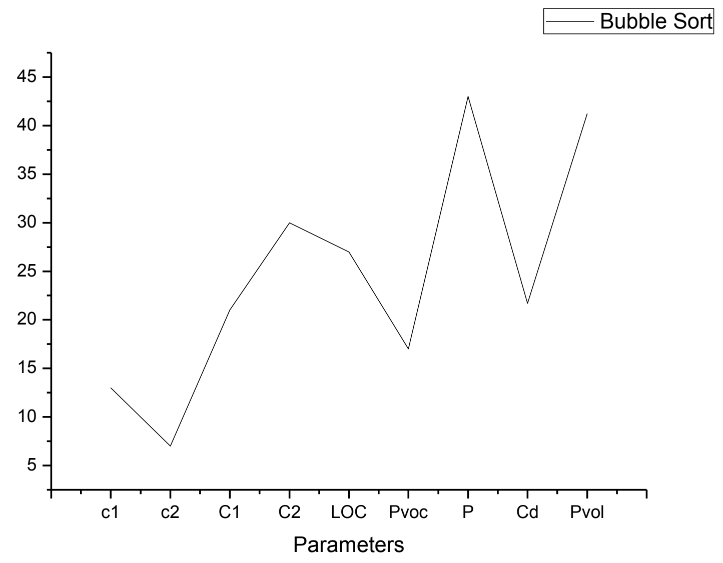

Figure 2: Bubble Sort Algorithm Analysis.

Table 3: Selection Sort Implementation Parameters

Metrics	Value
c1	14
c2	12
C1	18
C2	32
LOC	24
Pvoc	18
P	46
Cd	44
PVol	44.06

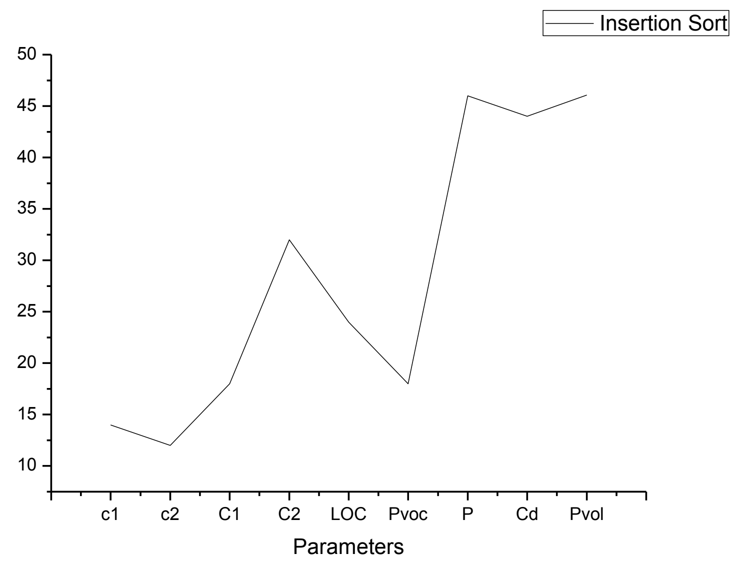

Figure 3: Insertion Sort Algorithm Analysis.

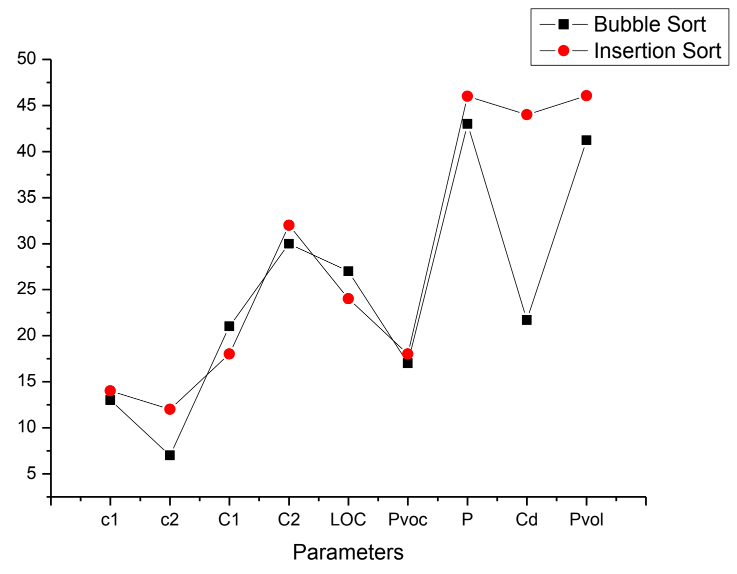

Figure 3: Bubble Sort and Insertion Sort Algorithms Analysis.

In this study, the selected Halstead based algorithms were implemented and analyzed using c1, c2, C1, C2, LOC, Pvoc, P, Cd, and Pvol.

Figure 3 shows that the bubble sort algorithm is 19.8% efficient compared to the insertion sort algorithm which is 16% efficient using LOC, the bubble sort algorithm is 5% efficient while the insertion sort algorithms are 7% efficient using c1, the bubble sort and insertion sort algorithms are both efficient using c2, the bubble sort has an efficiency of 13% compared to the insertion algorithm with 10% efficiency using C1, the insertion sort algorithm is 24% efficient compared to the bubble sort algorithm which is 22% efficient using C2, the insertion sort algorithm is 38% efficient compared to the bubble sort algorithm which is 35% efficient based on Pvol, the bubble sort algorithm is 8% efficient compared to the insertion sort algorithm which is 10% efficient based on Pvoc, the insertion sort algorithm showed a better indication which is 36% efficient compared to the bubble sort algorithm which is 12.5% efficient based on Cd.

CONCLUSION

In this study, two important sorting algorithms have been implemented in MATLAB and compared. The efficiency of each of the algorithms using Halstead parameters in handling various sorting tasks have also been discussed.

REFERENCES

1. Ammar, H.H., T. Nikzadeh, and J. Dugan. 1997. "A Methodology for Risk Assessment of Functional Specification of Software Systems Using Colored Petri Nets". *Proc. Of the Fourth International Software Metrics Symposium, Metrics'97,* Albuquerque, NM. Nov 5-7, 1997.108-117.

2. Basci, D. and S. Misra. 2009. "Measuring and Evaluating a Design Complexity Metric for XML Schema Documents' Code". *Journal of Information Science and Engineering*.1415-1425.

3. Banker, S., M. Datar and D. Zweig. 1989. "Software Complexity and Maintainability". In: *Proceedings of the Tenth International Conference on Information Systems*. Dec. 4-6, Boston, MA. 247-255.

4. Beizer B. 1984. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold: New York, NY.

5. Cafer, F. 2010. "Estimating Complexity of a Software Code".

6. Chidamber, S.R. and C.F. Kemerer. 1994. "A Metric Suite for Object Oriented Design". *IEEE Transactions Software Engineering*, SE-6:476-493.

7. Fenton, N.E. 1992. *Software Metrics – A Rigorous Approach*. Chapman & Hall: London, UK.

8. Fenton, N.E. and S.L. Pfleeger. 1997. *Software Metrics: A Rigorous and Practical Approach, 2nd Edition Revised.* PWS Publishing: Boston, MA.

9. Galin, D. 2004. *Software Quality Assurance*. Pearson Addison Wesley: London, UK.

10. Gill, G.K. and C.F. Kemerer. 1991. "Cyclomatic Complexity Density and Software Maintenance". *IEEE Trans. Software Engineering.* 17:1284-1288.

11. Halstead, M.H. 1997. *Elements of Software Science*. Elsevier North-Holland: New York, NY.

12. Hughes, B. 2006. *Software Project Management, 4th Edition*. McGraw-Hill: New York, NY.

13. Jones, C. 1986. *Programming Productivity*. McGraw Hill: New York, NY.

14. McCabe, T.J. and A.H. Watson. 1994. *Software Complexity*. McCabe and Associates, Inc. (last accessed 17.03.2010). Available at: http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp

15. McCall, J.A., P.K. Richards, and G.F. Walters. 1977. *Factory in Software Quality, Volume I-III*, US Rome Air Development Center Reports NTIS AD/A-049 014. 015, 055. National Technical Information Service, US Department of Commerce: Washington, D.C..

16. McCabe. T. 1976. "A Complexity Measure". *IEEE Transactions of Software Engineering*. SE-1:312-327.

17. Misra, S. and I. Akman. 2008. "A Complexity Metric Based on Cognitive Informatics". *Lecture Notes in Computer Science*. 5009:620-627.

18. Miller, D.R. and O.C. James. 2001. "Multi-Paradigm Design for C++' Book review and Commentary". (last accessed 22.03.2010) Available at: http://www.inkdrop.net/docs/multiParadigm.pd

19. Ramamoorthy, C.V., W.T. Ramamoorthy, T. Tsai, T. Yamura, and A. Bhide. 1985. "Metrics Guided Methodology". *COMPSAC 85*. 111-120.

20. Tourlakis G.J. 1984. *Computability*. Reston, VA.

SUGGESTED CITATION

Awode, T.R., D.D. Olatinwo, O. Shoewu, S.O. Olatinwo, O.O. Omitola, and M. Adedoyin. 2017. "Halstead Complexity Analysis of Bubble and Insertion Sorting Algorithms". *Pacific Journal of Science and Technology.* 18(1):125-135.

Pacific Journal of Science and Technology