Journal of Engineering and Applied Sciences 3 (4): 368-372, 2008
ISSN: 1816-949X

An Exploratory Study of Software Complexity Measures of Merge Sort Algorithm

[1]S.O. Olabiyisi and [2]O.A. Bello
[1]Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria
[2]Department of Physical Science, Faculty of Natural Science,
Ajayi Crowther University, Oyo, Nigeria

Abstract: Programmers find it difficult to gauge the code complexity of an application, which makes the concept difficult to understand. The McCabe metric and Halstead's software science are two common code complexity measures. The McCabe metric determines code complexity based on the number of control paths created by the code. While this information supplies only a portion of the complex picture, it provides an easy-to-compute, high-level measure of a program's complexity. The McCabe metric is often used for testing. Halstead bases his approach on the mathematical relationships among the number of variables, the complexity of the code and the type of programming language statements. In this study, the 2 software complexity measures are applied to Merge sort algorithm. The intention is to study what kind of new information about the algorithm the complexity measures are able to give and to study which software complexity measures are the most useful ones in algorithm comparison. The results explicitly show that Merge sort has the least Halstead's Volume, Program Difficulty and Program Effort when programmed in Assembly language and has the least cyclomatic number when programmed in Visual BASIC.

Key words: Merge sort algorithm, software complexity, McCabe metric, halstead's software science

INTRODUCTION

Software quality is the degree to which software possesses a desired combination of attributes such as maintainability, testability, reusability, complexity, reliability, interoperability, etc. In other words, quality of software products can be seen as an indirect measure and is a weighted combination of different software attributes, which can be directly measured. Moreover, many practitioners believe that there is a direct relationship between internal and external software product attributes. For example, a lower software complexity (seen here as a structural complexity) could lead to a greater software reliability.

Complexity is a measure of the resources that must be expended in developing, implementing and maintaining an algorithm. Productivity is chiefly a management concern while reliability is a quality factor directly visible to users of software systems.

These externally visible attributes of software processes and products are strongly influenced by engineering attributes of software such as complexity. Well-designed software exhibits a minimum of unnecessary complexity, unmanaged complexity leads to software difficult to use, maintain and modify. It causes increased development costs and overrun schedules.

Algorithms are frequently assessed by the execution time and by the accuracy or optimality of the results. For practical use, an important aspect is the implementation complexity. An algorithm, which is complex to implement, requires skilled developers, longer implementation time and has a higher risk of implementation errors. Moreover, complicated algorithms tend to be highly specialized and they do not necessarily work well when the problem changes (Akkanen and Nurminen, 2000).

Algorithms can be studied theoretically or empirically. Theoretical analysis allows mathematical proofs of the execution times of algorithms but can typically be used for worst-case analysis only. Empirical analysis is often necessary to study how an algorithm behaves with typical input see (Sedgewick, 1995).

Ball and Magazine (1981) listed criteria for the comparison of heuristic algorithm that in addition to execution time include ease implementation, flexibility and simplicity. Controlling and measuring complexity is a challenging engineering, management and research

Corresponding Author: S.O. Olabiyisi, Department of Computer Science and Engineering, Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria

problem. Metrics have been created for measuring various aspects of complexity such as sheer size, control flow, data structures and intermodule structure. Complexity measures can be used to predict critical information about reliability and maintainability of software system from automatic analysis of source code.

Complexity measures also provide continuous feedback during software project to help control the development process. During testing and maintenance they provide detailed information about software modules to help pinpoint areas of potential instability.

SOFTWARE COMPLEXITY MEASURES

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures. Current military metrics programs emphasize non-complexity metrics that track project management information about schedules, costs and defects. While such project tracking measures are necessary to any substantial software engineering effort, they lack predictive power and are thus inadequate for risk management. Complexity measures can be used to predict critical information about reliability and maintainability of software systems from automatic analysis of the source code. Complexity measures also provide continuous feedback during a software project to help control the development process. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability.

Many of the factors affecting software quality that have been identified by researchers can be seen in part as functions of the complexity and size of the program and the capabilities of the programmers and managers. This will include, but is not limited to, testability, efficiency, legibility and structuredness.

There are a number of ways to quantify complexity in a program. The best-known metrics, which provide such feature, are McCabe's (1976) cyclomatic number and Halstead and Maurice (1977) volume. These metrics have been extensively validated and compared (Aggarwal *et al*., 2002; Ramil and Lehman, 2000; Bezier, 1984; Curtis, 1981; Schneidewind and Hoffman, 1979).

HALSTEAD'S COMPLEXITY MEASURES

Halstead argued that algorithms have measurable characteristics analogous to physical laws. His model is based on four different parameters: the number of distinct operators (instruction types, keywords, etc.) in a program, called n1; the number of distinct operands (variables and constants), n2; the total number of occurrences of the operators, N1 and the total number of occurrences of the operands, N2. The sum of n1 and n2 is denoted as n while the sum of N1 and N2 is called N. From those four counts, a number of useful measures can be obtained. The number of bits required to specify the program is called the volume V of the program and is obtained through the equation.

$$V = N \log_2 n$$

The program level, which is the difficulty of understanding a program, is calculated by:

$$L = (2n2)/(n1N2)$$

and the intelligence content of a program is given by:

$$I = L \times V$$

In an attempt to include the psychological aspects of complexity in the measures, Halstead studied the cognitive processes related to the perception and retention of simple stimuli. As reported in (Olabiyisi, 2006) and (Olabiyisi *et al*., 2007), the mean number of mental discriminations per second in an average human being, also called the Stroud number, is between 5 and 20.

Halstead and Maurice (1977) uses 18 as a reference point for his studies. In his model, the number of discriminations made in the preparation of a program, called effort, is given by:

$$E = V/L$$

All of these measures are valid under the assumption that the program is "pure," i.e., free of so-called "poor programming practices." Halstead defines six classes of impurities, among them, synonymous operands, unfactored expressions and common sub expressions. The complete description of these and other impurities is beyond the scope of this study. However, for the programs used for this study, all recognizable impurities were eliminated prior to obtaining the corresponding Halstead measures.

CYCLOMATIC COMPLEXITY MEASURES

Cyclomatic complexity is the most widely used member of a class of static software metrics. Cyclomatic complexity may be considered a broad measure of soundness and confidence for a program. Introduced by Thomas McCabe (1976), it measures the number of

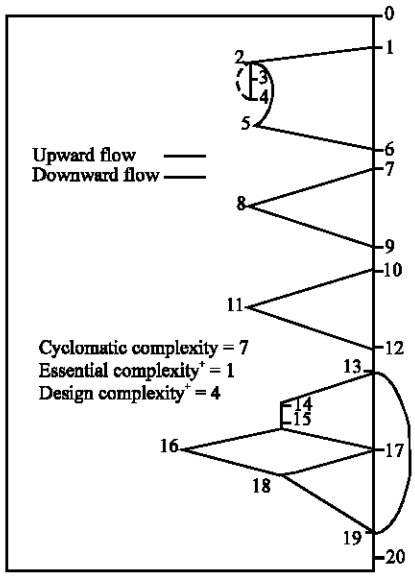

Fig. 1: Connected graph of a simple program

linearly independent paths through a program module. This measure provides a single ordinal number that can be compared to the complexity of other programs. Cyclomatic complexity is often referred to simply as program complexity, or as McCabe's complexity. It is often used in concert with other software metrics. As one of the more widely-accepted software metrics, it is intended to be independent of language and language format (McCabe, 1977). Cyclomatic complexity has also been extended to encompass the design and structural complexity of a system (McCabe *et al*., 1989; Olabiyisi, 2006; Olabiyisi *et al*., 2007).

The cyclomatic complexity of a software module is calculated from a connected graph of the module (that shows the topology of control flow within the program):

$$\text{Cyclomatic complexity (CC)} = E - N + p$$

where,

E = The number of edges of the graph.
N = The number of nodes of the graph.
p = The number of connected components.

To actually count these elements requires establishing a counting convention (tools to count cyclomatic complexity contain these conventions). The complexity number is generally considered to provide a stronger measure of a program's structural complexity than is provided by counting lines of code. Figure 1 is a connected graph of a simple program with a cyclomatic complexity of seven. Nodes are the numbered locations, which correspond to logic branch points; edges are the lines between the nodes.

EXPERIMENT WITH MERGE SORT ALGORITHM

The merge sort splits the list to be sorted into two equal halves and places them in separate arrays. Each array is recursively sorted and then merged back together to form the final sorted list. Like most recursive sorts, the merge sort has an algorithmic complexity of O(n log n). Elementary implementations of the merge sort make use of three arrays-one for each half of the data set and one to store the sorted list in. There are non-recursive versions of the merge sort, but they don't yield any significant performance enhancement over the recursive algorithm on most machines.

For the experiment, we used the complexity finder machine designed in Olabiyisi (2006) to calculate the complexity measures. To do so, the following actions were taken:

- The studied algorithm was coded using Assembly Language, C, Java, Pascal, Visual BASIC resulting in five programs. for each algorithm.
- The same programming style (modular programming) was employed in the coding.
- All the programs were run on the same computer.
- Operands, operator, keywords and identifiers were similarly defined for all the programs.

RESULTS AND DISCUSSION

Table 1 presents complexity measures of different implementation languages for Merge sort algorithm.

Figure 2 plots the graph of Halstead's volume for different implementation languages for Merge sort algorithm.

Figure 3 gives the graph of program difficulty for different implementation language of the algorithm. While Figure 4 presents the graph of Program Effort for different implementation languages for the studied algorithm.

There are interesting points to observe about these graphs. Figure 2 shows that Merge sort has the lowest and highest Halstead's Volume when coded in Assembly language and Java, respectively. By implication, the graph shows that Merge sort is best implemented in Assembly language followed by Visual Basic, Pascal, C and Java in that order.

Figure 3 indicates that if Program Difficulty is to be considered, Merge sort algorithm implemented in Assembly Language is the best while Merge sort implemented in Java is the worst.

Table 1: Merge sort complexity measures by different implementation languages

S. no.	Algorithm name	Language	Program vol. (v)	Program diffculty (D)	Program effort (E)	Cyclomatic no. V(G)
1	Merg sort	Assembly language	717	211	151287	13
2	Merg sort	C	1029	1095	1126755	7
3	Merg sort	Java	1046	1100	1150600	7
4	Merg sort	Pascal	807	588	474516	7
5	Merg sort	Visual basic	714	574	425334	4

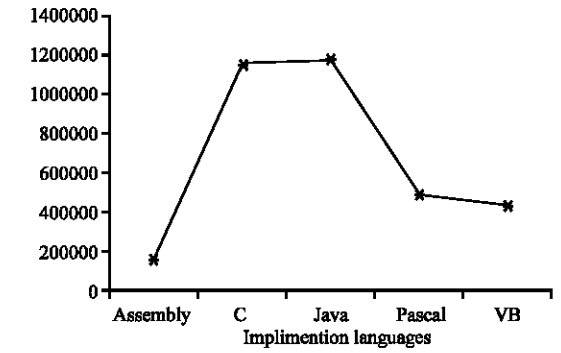

Fig. 2: Graph of different implementation of the merge sort algorithm

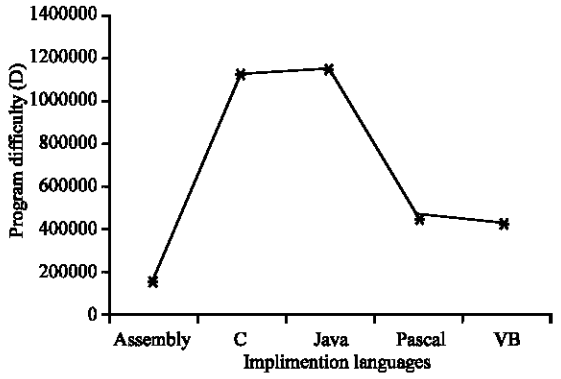

Fig. 3: Graph of program difficulty for different implemetation of the merge sort algorithm

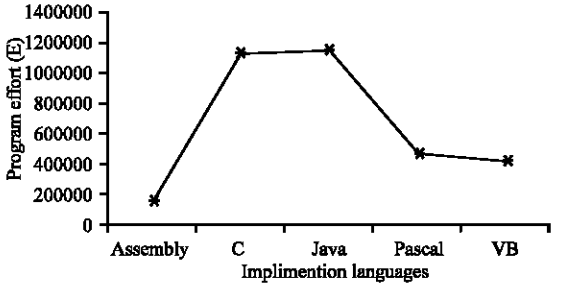

Fig. 4: Graph of program effort for different implementations of the merge sort algorithm

In Fig. 4, we discover that considering the program effort, Merge sort algorithm is best implemented in Assembly language followed by Visual Basic, Pascal and worst implemented in Java.

For all the implementation languages, the cyclomatic number is the same (i.e., 7) except for Assembly language and Visual Basic, which has a cyclomatic number of 13 and 4, respectively.

CONCLUSION

This research has considered software complexity measure experiment with Merge sort algorithm. We study the Merge sort algorithm by computing the Halstead's volume (V), the program effort (E), the program difficulty (D) and the cyclomatic number V(G) using different implementation languages.

Software complexity measures might help practitioners to choose, out of a large number of alternatives, the algorithms that best match their needs. Understanding the trade-off between implementation and performance would give a firmer basis to decision-making.

REFERENCES

Akkanen, J. and J.K. Nurminen, 2000. Case-study of the evolution of routing algorithms in a network planning tool. J. Syst. Software, 58: 181-198.

Aggarwal, K.K., Y. Singh and J.K. Chhabra, 2002. An Integrated Measure of Software Maintainability. In Proceeding Annual Reliability and Maintainability Symposium, IEEE.

Ball, M. and M. Magazine, 1981. The design and analysis of heuristics. Networks, 11: 215-219.

Bezier, B., 1984. Software System Testing and Quality Assurance, Van Nostrand Reinhold, New York.

Curtis, B., 1981. The Measurement of Software Quality and Complexity, Software Metrics. Perlis A. *et al.*, (Eds.). MIT Press, Cambridge.

Halstead and H. Maurice, 1977. Elements of Software Science, Elsevier North-Holland, New York.

McCabe, T.J., 1976. A Complexity Measure. IEEE. Trans. Software Eng., 2 (4): 308-320.

McCabe, J. Thomas and Charles Butler, 1989. Design Complexity Measurement and Testing. Commun. ACM., 32: 1415-1425.

Olabiyisi, S.O., 2006. Universal Machine for Complexity Measurement of Computer Programs. Ph.D Thesis Ladoke Akintola University of Technology Ogbomoso.

Olabiyisi, S.O., R.A. Ganiyu, M.O. Ekundayo, O.O. Okediran and O.O. Oderinde, 2007. Using Software Complexity Measures to Analyze Algorithms. An Experiment with Selection Sort Algorithm: Ghana J. Sci. C.S.I.R.-INSTI.

Ramil, J.F. and M.M. Lehman, 2000. Metrics of Software Evolution as Effort Predictors: A Case Study. In: Proceeding International Conference Software Maintenance, IEEE.

Sedgewick, R., 1995. Algorithms in C++. Reading, MA: Addison-Wesley.

Schneidewind, N.F. and H.M. Hoffman. 1979. An Experiment in Software Error Data Collection and Analysis. IEEE. Trans. Software Eng., 5 (3): 276-286.