Journal of Engineering and Applied Sciences 3 (1): 118-122, 2008
ISSN: 1816-949X

Comparative Study of Implementation Languages on Software Complexity Measures of Quick Sort Algorithm

S.O. Olabiyisi, R.O. Ayeni and E.O. Omidiora
Department of Computer Science and Engineering,
Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria

Abstract: In this study, we apply different software complexity measures to Quick sort algorithm. Our intention is to study what kind of new information about the algorithm the complexity measures (Halstead's volume and Cyclomatic number) are able to give and to study which software complexity measures are the most useful ones in algorithm comparison. The results explicitly show that Quick sort has the least Halstead's Volume, Program Difficulty and Program Effort when programmed in Assembly language.

Key words: Software complexity, quick sort, halsted complexity measure, cyclomatic complexity measure

INTRODUCTION

A programmer usually has a choice of data structures and algorithms to use. Choosing the best one for a particular job involves, among other factors, two important measures:

Time complexity: How much time will the program take?
Space complexity: How much storage will the program need?

A programmer will sometimes seek a tradeoff between space and time complexity. For example, a programmer might choose a data structure that requires a lot of storage in order to reduce the computation time. There is an element of art in making such tradeoff, but the programmer must make the choice from an informed point of view. The programmer must have some verifiable basis on which the selection of a data structure or algorithm. Complexity analysis provides such a basis.

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 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.

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

MATERIALS AND METHODS

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's (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 by 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 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 in 1976, it measures the number of 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. Cyclomatic complexity has also been extended to encompass the design and structural complexity of a system (Mc Cabe and Charles, 1989; Olabiyisi, 2006; Olabiyisi *et al*., 2006).

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):

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

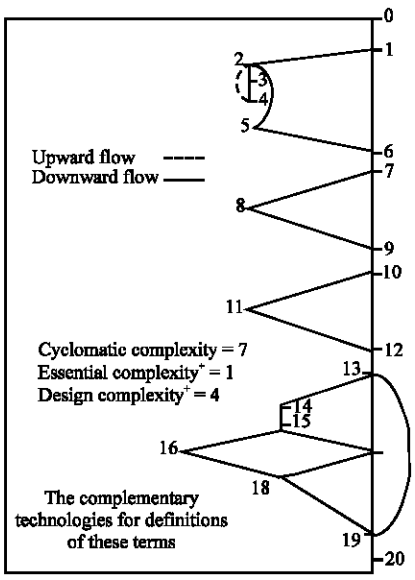


Fig. 1: Connected graph of a simple program

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 QUICK SORT ALGORITHM

The quick sort is an in-place, divide-and-conquer, massively recursive sort. As a normal person would say, it's essentially a faster in-place version of the merge sort. The quick sort algorithm is simple in theory, but very difficult to put into code (computer scientists tied themselves into knots for years trying to write a practical implementation of the algorithm and it still has that effect on university students).

The recursive algorithm consists of four steps (which closely resemble the merge sort):

- If there are one or less elements in the array to be sorted, return immediately.
- Pick an element in the array to serve as a "pivot" point. (Usually the left-most element in the array is used.)
- Split the array into two parts-one with elements larger than the pivot and the other with elements smaller than the pivot.
- Recursively repeat the algorithm for both halves of the original array.

The efficiency of the algorithm is majorly impacted by which element is chosen as the pivot point. The worst-case efficiency of the quick sort, $O(n^2)$, occurs when the list is sorted and the left-most element is chosen. Randomly choosing a pivot point rather than using the left-most element is recommended if the data to be sorted isn't random. As long as the pivot point is chosen randomly, the quick sort has an algorithmic complexity of $O(n \log n)$.

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

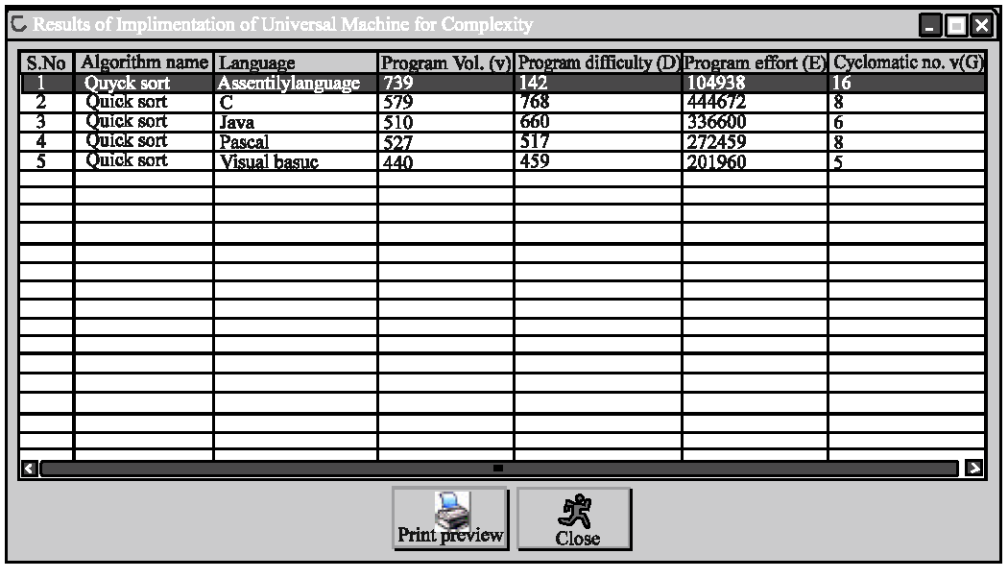
Table 1 presents complexity measures of different implementation languages for Quick sort algorithm.

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

Figure 3 gives the graph of program difficulty for different implementation language of the algorithm. While Fig. 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 Quick sort has the highest Halstead's Volume when code in C. By implication, the graph shows that Quick sort is best implemented in Assembly language followed by Visual Basic, Pascal, Java and C in that order.

Figure 3 indicates that if Program Difficulty is to be considered, Quick sort algorithm implemented in Assembly language is the best while Quick sort implemented in C is the worst.



S.No	Algorithm name	Language	Program Vol. (v)	Program difficulty (D)	Program effort (E)	Cyclomatic no. v(G)
1	Quyck sort	Assemtilylanguage	739	142	104938	16
2	Quick sort	C	579	768	444672	8
3	Quick sort	Java	510	660	336600	6
4	Quick sort	Pascal	527	517	272459	8
5	Quick sort	Visual basuc	440	459	201960	5

Table 1: Quick sort complexity measures by different implementation languages

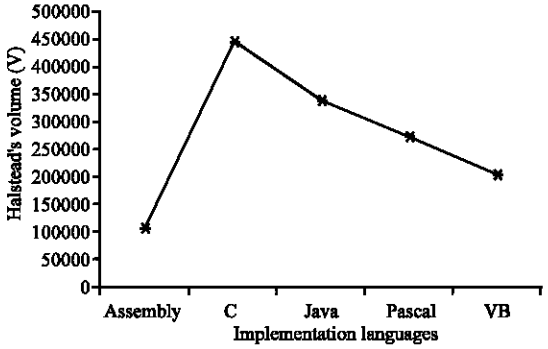


Fig. 2: Graph of different implementation of the quick sort algorithm

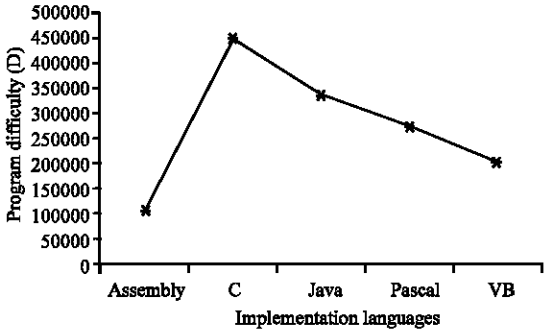


Fig. 3: Graph of program difficulty for different implementation of the quick sort algorithm

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

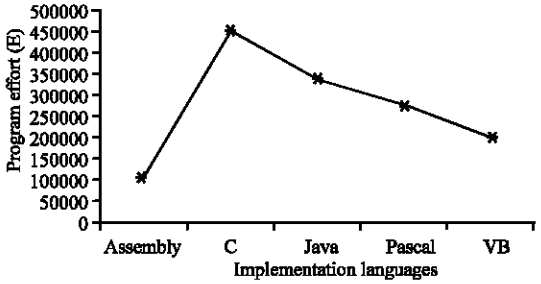


Fig. 4: Graph of program effort for dieerent implementation of the quick sort algorithm

CONCLUSION

This research has considered software complexity measure experiment with Quick sort algorithm. We study the Quick 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. Sys. Software, 58: 181-98.

Aggarwal, K.K., Y. Singh and J.K. Chhabra, 2002. An Integrated Measure of Software Maintainability. In: Proceedings of 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 (Eds.). A. Perlis *et al*., 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: 308-320.

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

Olabiyisi, S.O., 2007. Universal Machine for Complexity Measurement of Computer Programs. Ph.D Thesis Ladoke Akintola University of Technology Ogbomoso, 2006.

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 Journal of Science C.S.I.R.-INSTI.

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

Schneidewind, N.F. and H.M. Hoffman, 1979. IEEE. Trans. Software Eng., 5: 276-286.

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