

USING SOFTWARE COMPLEXITY MEASURES TO ANALYSE ALGORITHMS – AN EXPERIMENT WITH HEAP SORT ALGORITHM

Olabiyisi S. O¹., Omidiora E.O¹. & ²Omidiora M.A².

¹Department of Computer Science and Engineering

Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomosho, Nigeria.

²Electrical, Electronic & Computer Engineering Department,
Nnamdi Azikiwe University, Awka, Nigeria

Abstract

In this paper, we apply different software complexity measures to Heap 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 Heap sort has the least Halstead's Volume, Program Difficulty and Program Effort when programmed in Pascal language and has the least cyclomatic number when programmed in Visual BASIC.

Keywords: Heap sort algorithm, Cyclomatic complexity, Halstead's Complexity Measures

1. 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 [1].

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 [2].

Ball and Magazine [3] 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.

2. 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 cyclomatic number [4] and Halstead's volume [5]. These metrics have been extensively validated and compared [6], [7], [8], [9], [10].

3. 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 n_1 ; the number of distinct operands (variables and constants), n_2 ; the total number of occurrences of the operators, N_1 ; and, the total number of occurrences of the operands, N_2 . The sum of n_1 and n_2 is denoted as n while the sum of N_1 and N_2 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

$$L = (2n_2)/(n_1N_2)$$

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. Research by Stroud (1966) had shown that 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 paper. However, for the programs used for this paper, all recognizable impurities were eliminated prior to obtaining the corresponding Halstead measures.

4. 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 (McCabe, 1994). Cyclomatic complexity has also been extended to encompass the design and structural complexity of a system [11].

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.

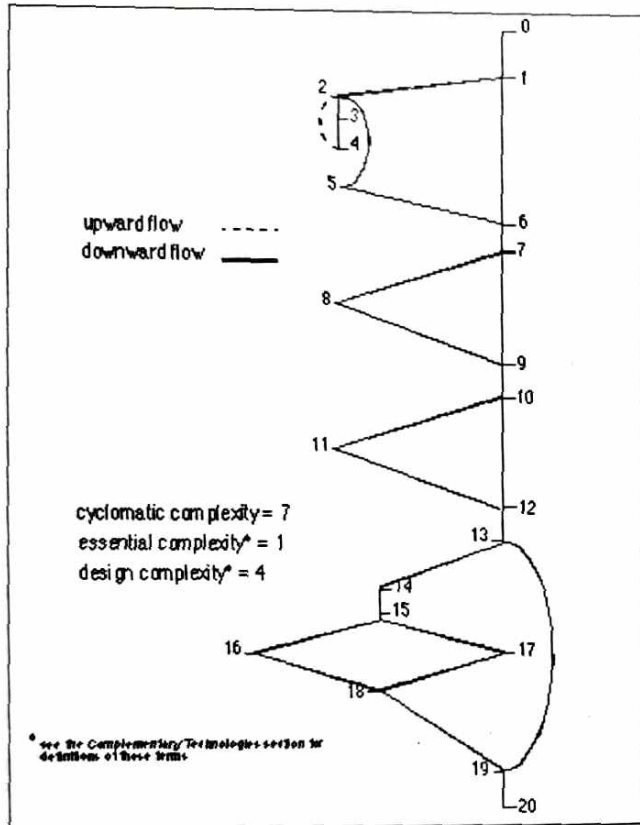


Figure 1: Connected Graph of a Simple Program

5. Experiment With Heap Sort Algorithm

The heap sort is the slowest of the $O(n \log n)$ sorting algorithms, but unlike the merge and quick sorts it doesn't require massive recursion or multiple arrays to work. This makes it the most attractive option for *very* large data sets of millions of items. The heap sort works as its name suggests - it begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap and removes the largest remaining item and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements. To do an in-place sort and save the space the second array would require, the algorithm below "cheats" by using the same array to store both the

heap and the sorted array. Whenever an item is removed from the heap, it frees up a space at the end of the array that the removed item can be placed in.

For the experiment, we used the complexity finder machine designed in [12] 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.

6. Result And Discussions

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

Table 1: Heap sort Complexity Measures by different Implementation Languages.

S. No.	Algorithm Name	Language	Program Vol. (V)	Program Difficulty (D)	Program Effort (E)	Cyclomatic No. v(D)
1	Heap sort	Assembly Language	972	265	257388	17
2	Heap sort	C	723	1001	723723	7
3	Heap sort	Java	741	1024	751374	7
4	Heap sort	Pascal	573	610	349530	7
5	Heap sort	Visual Basic	585	630	368550	6

Figure 2 plots the graph of Halstead's volume for different implementation languages for Heap 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.

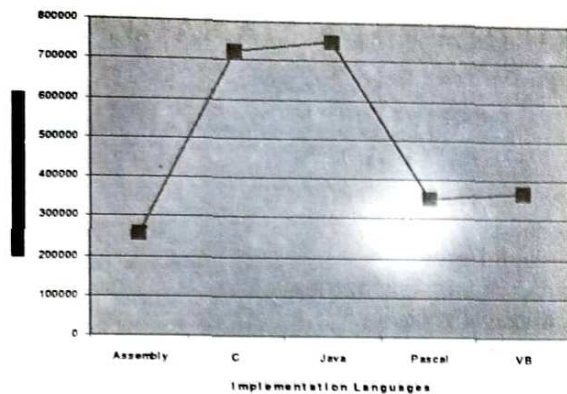


Figure 2: Graph of different implementation of the Heap sort Algorithm

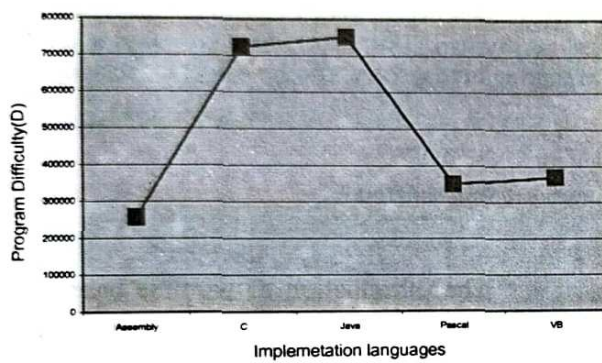


Figure 3: Graph of Program Difficulty for Different Implementation of the Heap sort Algorithm

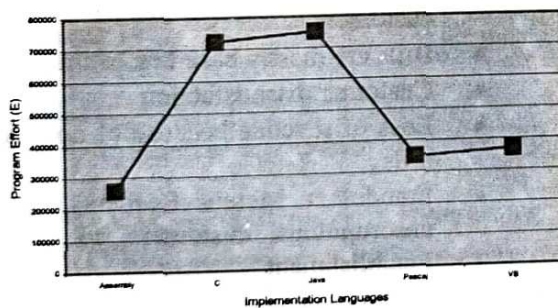


Figure 4: Graph of Program Effort for Different Implementations of the Heap sort Algorithm

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

Figure 2 indicates that if Program Difficulty is to be considered, Heap sort algorithm implemented in Pascal is the best while Heap sort implemented in Java is the worst.

In figure 3, we discover that considering the program effort, Heap 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 Visual BASIC, which has a cyclomatic number of 6.

7. Conclusions

This work has considered software complexity measure experiment with Heap sort algorithm. We study the Heap 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

- [1] Akkanen J, Nurminen JK. *Case-study of the evolution of routing algorithms in a network planning tool*, Journal of Systems and Software:58, 2000, 181-98.
- [2] Sedgewick R. *Algorithms in C++*. (Reading, MA: Addison-Wesley, 1995).
- [3] Ball M, Magazine M. *The design and analysis of heuristics. Networks*, 1981
- [4] McCabe T. J. *A Complexity Measure*, IEEE Transactions on Software Engineering, SE-2, Number 4, December 1976, 308-320.
- [5] Halstead, Maurice H., *Elements of Software Science*, (Elsevier North-Holland, New York, 1977).
- [6] Aggarwal K. K., Singh Y., and Chhabra J. K., *An Integrated Measure of Software Maintainability*. In Proceedings of Annual Reliability and Maintainability Symposium, IEEE, 2002.
- [7] Ramil J. F. and Lehman M. M., *Metrics of Software Evolution as Effort Predictors - A Case Study*, In Proceedings of International Conference on Software Maintenance, IEEE, 2000.
- [8] Bezier B. *Software System Testing and Quality Assurance*, (Van Nostrand Reinhold, New York, 1984).
- [9] Curtis B., *The Measurement of Software Quality and Complexity*, *Software Metrics*, ed. By A. Perlis, et al., (MIT Press, Cambridge, 1981).
- [10] Schneidewind N. F. and Hoffman H. M., *An Experiment in Software Error Data Collection and Analysis*, IEEE Transactions on Software Engineering, SE-5, Number 3, May 1979, 276-286.
- [11] McCabe, Thomas J., and Charles Butler, *Design Complexity Measurement and Testing*, Communications of the ACM, 32, December 1989, 1415-1425.
- [12] Olabiyisi S. O., *Universal Machine for Complexity Measurement of Computer Programs*, Ph.D Thesis Ladoke Akintola University of Technology Ogbomoso, 2005.