Comparative Study of Complexities of Breadth-First Search and Depth-First Search Algorithms using Software Complexity Measures

Akanmu T. A., Olabiyisi S. O., Omidiora E. O. ,Oyeleye C. A., Mabayoje M.A. and Babatunde A. O.

Abstract: In this study, two different software complexity measures were applied to breadth-first search and depth-first search algorithms. The intention is to study what kind of new information about the algorithm the complexity measures (Halstead's volume and Cylomatic number) are able to give and to study which software complexity measure is the most useful one in algorithm comparison. The results clearly show that with respect to Program Volume, breadth-first search algorithm is best implemented in Pascal language while depth-first search is best implemented in C language. The values of Program Difficulty and Program Effort indicate that both the algorithms are best implemented in Pascal language. Cyclomatic number is the same for both algorithms when programmed in Visual BASIC (i.e. 6).

Keywords: Search algorithms, software complexity, breadth-first search, depth-first search.

Akanmu T. A. is with the Department of Computer Science, Osun State Polytechnic,Iree, P.M.B. 301, Iree, Nigeria. **(e-mail:** funmi_kunle@yahoo.com).
Olabiyisi S. O. is with the Department of Computer Science and Engineering, Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria (corresponding author, phone no: +2348036669863; e-mail: tundeolabiyisi@hotmail.com).
Omidiora E. O. is with the Department of Computer Science and Engineering, Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria. (e-mail: omidiorasayo@yahoo.co.uk).
Oyeleye C. A. is with the Department of Computer Science and Engineering, Ladoke Akintola University of Technology, P.M.B. 4000, Ogbomoso, Nigeria (e-mail: **letuskii@yahoo.com).**
Mabayoje M. A. and Babatunde A. O. are both with the Department of Computer Science, University of Ilorin, Ilorin Nigeria.

I. 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, 2 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 attribute 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 caused increased development costs and overrun schedules.

Algorithms are frequently assessed by the execution time and by the accuracy or optimality of the result. For practical use an important aspect is the implementation complexity. An algorithm, which is complex to implement, require skilled developers, longer implementation complexity 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 allow mathematical proofs of the execution time of algorithm but can typically by used for worst-case analysis only. Empirical analysis is often necessary to study how an algorithm behave with typical input [2].

[3] listed criteria for the comparison of heuristic algorithm that in addition to execution time include ease of implementation, flexibility and simplicity. Controlling and measuring complexity is a challenging engineering, management and research problem. Metric have been created for measuring various aspect of complexity such as sheer size, control flow, data structure and intermodule structure. Complexity measure 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 developmental process. During testing and maintenance they provide detailed information about software modules to help pinpoint areas of potential instability.

II. SOFTWARE COMPLEXITY MEASURES

Software complexity is one branch of software metric that is focused on direct measurement of software attributes, as opposed to indirect software measure such as project milestone status and reported system failures. Military program emphasize non- complexity metric that track project management information about schedules, cost 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 system 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 researcher can be seen in part as function 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 number of ways to quantify complexity in a program. The best-known metrics, which provide such features, are [4] Cyclomatic number and [5] Halstead's volume. These metrics have been extensively validated and compared [6]-[10].

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, and keywords) in a program, called nl; 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.

$$V = N \log2 n$$

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

$$L = (2n2)/(n1\ N2)$$

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 [11], [12] and [14], the mean number of mental discriminations per second in an average human being, also called the Stroud number, is between 5 and 20.

[5] 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 [6], 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 McCabes' 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 [6]. Cyclomatic complexity has also been extended to encompass the design and structural complexity of a system ([13], [11], and [12].

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):

Cyclomatic complexity (CC) = E – N + p

where,

E	=	The number of edge 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 below 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.

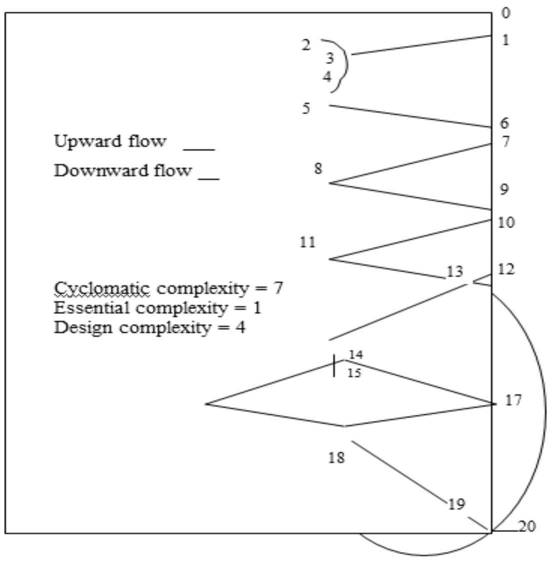

Fig 1: connected graph of a simple program

III. EXPERIMENT WITH BREADTH-FIRST SEARCH ALGORITHM

Breadth-first search is an algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbour nodes, and so on, until it finds the goal.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a First in First out (FIFO) queue. In typical implementations, nodes that have not yet been examined for their neighbours are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

Experiment With Depth-First Search Algorithm

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it has not finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a Last in-first out (LIFO) stack for expansion.

For the experiment, we used the complexity finder machine designed in [11] to calculate the complexity measures. To do so, the following actions were taken:

a. The studied algorithm was coded using C, C++, Pascal and Visual BASIC resulting in three programs for each algorithm
b. The same programming style (modular programming) was employed in the coding.
c. All the programs were run on the same computer.
d. Operands, operators, keywords and identifiers were similarly defined for all the programs.

IV. RESULTS AND DISCUSSION

Table 1 presents complexity measure of different implementation languages for breadth-first search algorithm.

Table 2 presents complexity measure of different implementation languages for depth-first algorithm.

Figure 2 presents the graph of Program volume for different implementation languages for breadth-first search algorithm.

Figure 3 presents the graph of Program Difficulty for different implementation languages of the breadth-first search algorithm.

Figure 4 presents the graph of Program Effort for different implementation language for the breadth-first search algorithm. While figure 5 presents the graph of Cyclomatic Number for different implementation language for the breadth-first search algorithm.

Figure 6 presents the graph of Program Volume for different implementation languages for depth-first algorithm.

Figure 7 presents the graph of Program Difficulty for different implementation languages of the depth-first algorithm.

Figure 8 presents the graph of Program Effort for different implementation language for the depth-first algorithm. While figure 9 presents the graph of Cyclomatic Number for different implementation language for the depth-first search algorithm.

There are interesting points to observe about these graphs. Figure 2 shows that breadth-first search has the lowest and highest Program volume when coded in Pascal language and Visual BASIC language respectively. By implication, the graph shows that breadth-first search algorithm is best implemented in Pascal language followed by C language, C++ language and Visual BASIC language in that order.

Figure 3 indicates that if Program Difficulty is considered, breadth-first search algorithm implemented in Pascal language is the best while breadth-first search algorithm implemented in C++ language is the worst.

In figure 4, It was discovered that considering the Program Effort, breadth-first search algorithm is best implemented in Pascal language and worst implemented in C++ language.

Cyclomatic number has the least value in Pascal and the highest value in Visual BASIC language. Its value in C and C++ languages are the same (i.e., 6) as it can be seen in figure 5.

Figure 6 shows that if Program Volume is considered, depth-first search algorithm is best implemented in C and worst implemented in Visual BASIC language.

It is discovered in figure 7 that considering the Program Difficulty, depth-first search algorithm implemented in Pascal language is the best while depth-first search algorithm implemented in C language is the worst.

Depth-first search algorithm is also best implemented in Pascal language and worst implemented in Visual BASIC language as can be shown in Program Effort graph in figure 8.

The values of the Cyclomatic number are the same (i.e. 5) for C, C++ and Pascal languages, but it is 6 for Visual BASIC language as can be seen in figure 9.

Table 1: Breadth-First Search Algorithm Complexity Measures By Different Implementation Languages
Results of Implementation Universal Machine for Complexity

Algorithm Name	Language	Program Vol(V)	Program Difficulty(D)	Program Effort(E)	Cyclomatic No V(G)
Breadth-First Search	C	733	20	14660	5
Breadth-First Search	C++	998	17	16966	5
Breadth-First Search	Pascal	640	9	5760	3
Breadth-First Search	Visual BASIC	1197	10	11970	6

Table 2: Depth-First Search Algorithm Complexity Measures By Different Implementation Languages
Results of Implementation Universal Machine for Complexity

Algorithm Name	Language	Program Vol(V)	Program Difficulty(D)	Program Effort(E)	Cyclomatic No V(G)
Depth-First Search	C	491	16	7851	5
Depth-First Search	C++	515	18	9270	5
Depth-First Search	Pascal	539	8	4312	5
Depth-First Search	Visual BASIC	1069	9	9627	6

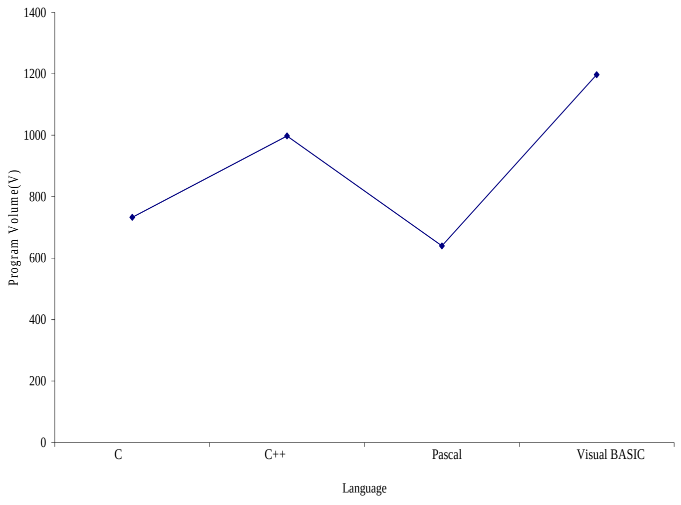

Fig.2: Graph of Program Volume(V) for different

Implementation of Breadth-First Search Algorithm.

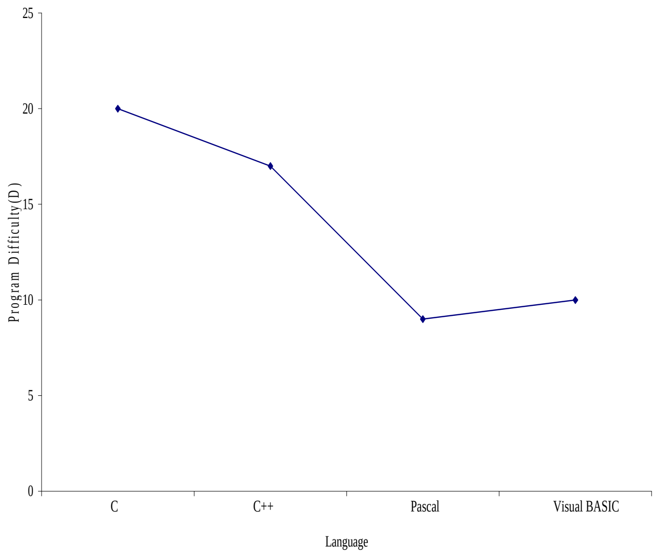

Fig.3: Graph of Program Difficulty(D) for different Implementation of Breadth-First Search Algorithm.

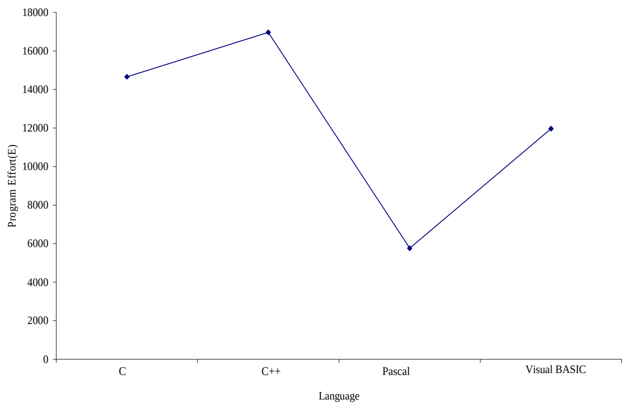

Fig.4: Graph of Program Effort(E) for different Implementation of Breadth-First Search Algorithm.

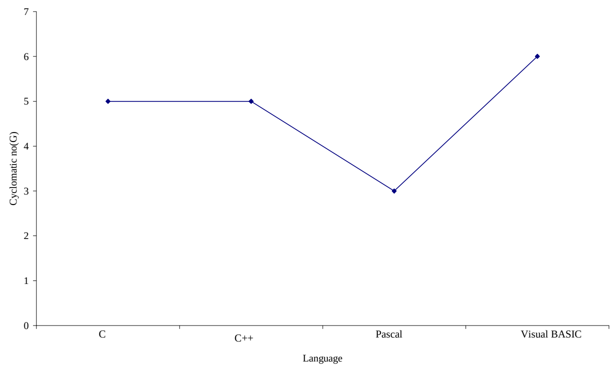

Fig.5: Graph of CyclomaticV(G) Number for different Implementation of Breadth-First Search Algorithm.

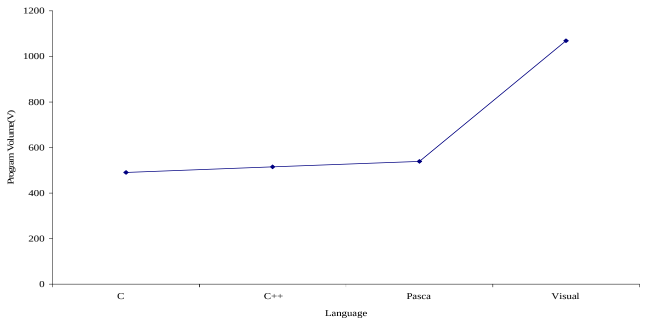

Fig.6: Graph of Program Volume(V) for different implementation of Depth-First search algorithm

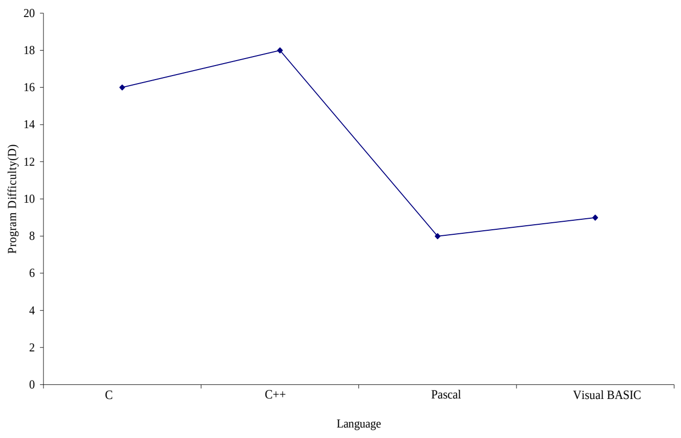

Fig.7: Graph of Program Difficulty(D) for different implementation of Depth-First search algorithm

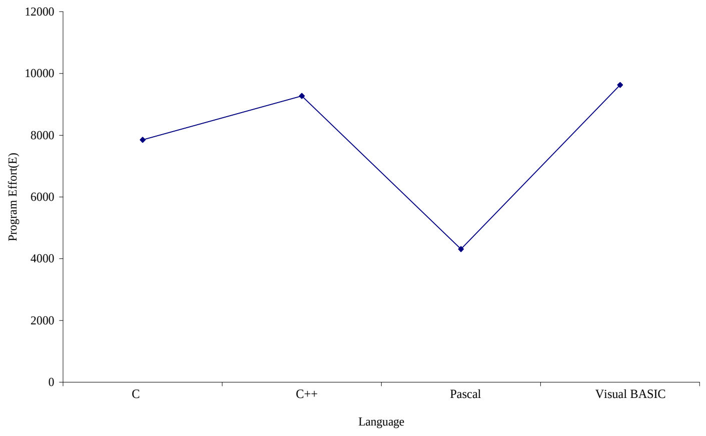

Fig.8: Graph of Program Effort(E) for different implementation of Depth-First search algorithm

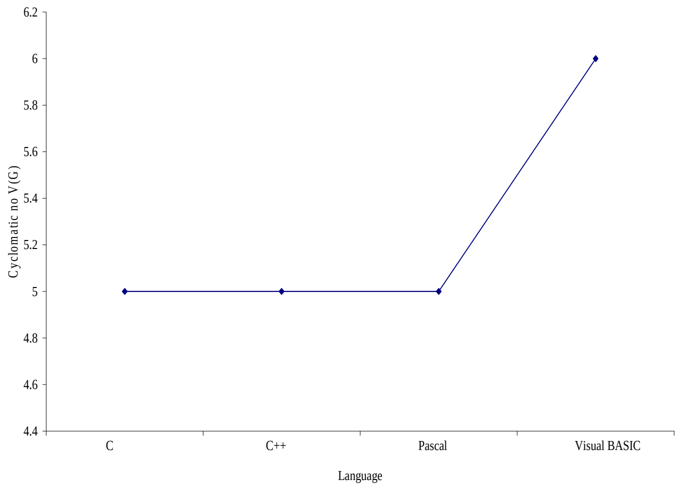

Fig.5: Graph of Cyclomatic Number V(G) for different implementation of Depth-First search algorithm

V. CONCLUSION

This research has considered software complexity measure experiment with breadth-first search and depth-first search algorithms. Both the breadth-first search and depth-first search algorithms were studied by computing the Program 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 and Nurminen, J.K.(2000). Case- Study of the Evolution of Routing Algorithms in a Network Planning tool. J. Syst. Software, 58: 181-198

[2] Sedgewick, R.,(1995). Algorithm in C++. Reading

[3] Ball, M. and Magazine,M. (1981). The Design and Analysis of Heuristics Network, 11: 215-219.

[4] McCabe, T.J, (1976). A Complexity Measure, IEEE Trans Software ENG., 2(4) 308-320.

[5] Halstead, and Maurice,H. (1977). Element of Software Science, Elsevier North- Hollland , New York.

[6] Aggarwal, K., Singh, K. Y. and Chhabra J.K. (2002). An Integrated Measure of Software maintainability. In proceeding Annual Reliability and Maintainability Symposium, IEEE.

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

[8] Bezier, B. (1984). Software System Testing and Quality Assurance, Van Nostrand Reinhold, New York.

[9] Curtis, B. (1981). The Measurement of Software Quality and Complexity, Software Metrics, Perils A. et al., (Eds.) MIT Press, Cambridge.

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

[11] Olabiyisi, S.O.(2006). Universal Machine for Complexity Measurement of Computer Programs. Ph.D Thesis Ladoke Akintola Unversity of Technology Ogbomoso.

[12] Olabiyisi, S.O., Ganiyu, R.A., Ekundayo, M.O., Okediran, O.O., and Oderinde, O.O (2007). Using Software Complexity Measure to Analyze Algorithm- An Experiment with Selection Sort Algorithm, Ghana J sci C.S.I.R.-INSTI.

[13] McCabe, J., Thomas and Charles B., (1989). Design Complexity Measurement and Testing. Common ACM, 32:1415-1425.

[14] Akanmu, T.A.(2009). An Exploratory Study of Software Complexity Measures of Breadth-First Search Algorithm. Journal of Pure and Applied Sciences (JOPAS) Volume 7 (1&2)