Numerical Study of Depth of Recursion in Complexity Measurement Using Halstead Measure

A.E. Okeyinka
Department of Mathematical Sciences
Ondo State University of Science and Technology
Okitipupa, Nigeria

O.M. Bamigbola
Department of Mathematics
University of Ilorin
Ilorin, Kwara state, Nigeria

Abstract

Complexity of algorithms has been studied analytically using the concept of Big O notation. One of the flaws of the study is that the complexities obtained for algorithms are in most cases the same; whereas in reality such algorithms might vary in terms of efficiency. The reason for the disparity is, of course, due to the definition of the Big O itself which mathematically is sound anyway. However, for pragmatic purposes, there is need for estimating actual complexities of each algorithm to be sure of which one is the best given more than one algorithms solving the same problem. In this study, recursion is considered and the complexities of recursive algorithms are estimated numerically using Halstead measure. Our findings show that recursive algorithms are more complex and hence less efficient than non-recursive algorithms.

Keywords: Numerical, Complexity, Depth, Algorithms, Big O

1. Introduction

There are a number of important practical and theoretical reasons for analyzing algorithms. The principal reason is that we need to obtain estimates or bounds on the storage or run time which our algorithm will need to successfully process a particular input. Computer time and memory are relatively scarce resources which are often simultaneously sought by many users. It is advantageous to avoid runs that are aborted because of insufficient time. One would like to predict such things with pencil and paper in order to avoid disastrous runs. A good analysis is also capable of finding bottlenecks in our programs, that is, sections of a program where most of the time is spent.

Computational complexity theory investigates the problems related to the amount of resources required for the execution of algorithms (e.g. execution time), and 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, among other things, 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. Intuitively, we consider the example of an instance that is n bits long, that can be solved in n^2 steps. In this case we say the problem has a time complexity of order n^2. The Big O notation is generally used in measuring the complexity of algorithms. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2)$ on most other computers, so this notation allows us to generalize away from the details of a particular computer [49]. 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.

2. Research Motivation and Methodology

The need for choosing a better algorithm out of two or many that claim to solve the same problem can not be over-emphasised. The following are factors influencing program or algorithm efficiency:

- Problem being solved
- Programming language being used
- Computer compiler being used
- Computer hardware
- Programmer's ability (effectiveness)

Using these factors would only give subjective results, hence the rationale for making recursion to the use of Big O concept. However, the Big O itself, being an analytic solution often gives same results for algorithms that are apparently not the same considering efficiency. For instance it gives O(n) efficiency to both bubble sort and successive minima sort. It gives the same result also (i.e. O(nlogn) to heap sort, quick sort and merge sort. Hence, as expected though, it gives general results rather than specific results. But in this study, we employ numerical approach; we also take into consideration recursive and non-recursive functions. To achieve these, we design a tool for implementing Halstead complexity measure.The tool is used to measure computational complexity of algorithms using the traditional Halstead concept and a modified concept that overcomes the inherent limitations in the traditional Halstead measure.

These limitations which our tool has overcome are as follows:

- traditional Halstead regards multiplication and division as having the same complexity as addition and subtraction respectively
- traditional Halstead does not take into consideration the depth of recursion.

In general Halstead measure estimates complexity metrics directly from the operators and operands in the source code.The measureable and countable properties are:

- n1 = number of unique or distinct operators appearing in the code
- n2 = number of unique or distinct operands appearing in the code
- N1 = total number of all of the operators appearing in the code
- N2 = total number of all of the operands appearing in the code

Given these metrics the following measures (among others) can be computed: program volume, difficulty level, effort, time to implement, program level, program lenght and program vocabulary.

3. The conceptual framework of the complexity evaluator

The conceptual model of the tool used (the complexity evaluator) in this study is presented below; also its major modules and their functions are stated. The model is made up of the Parser, code generator, and code compiler. Other key players are: program loader, stream reader, file locators, string manipulator, parse tree manipulator, parse tree reader and stream writer

3.1 The Complexity Evaluator

The complexity evaluator developed is capable of scanning given programs and compiling operators and operands in the programs. It can also do multiplication, division and recursion taking cognisance of their depths.

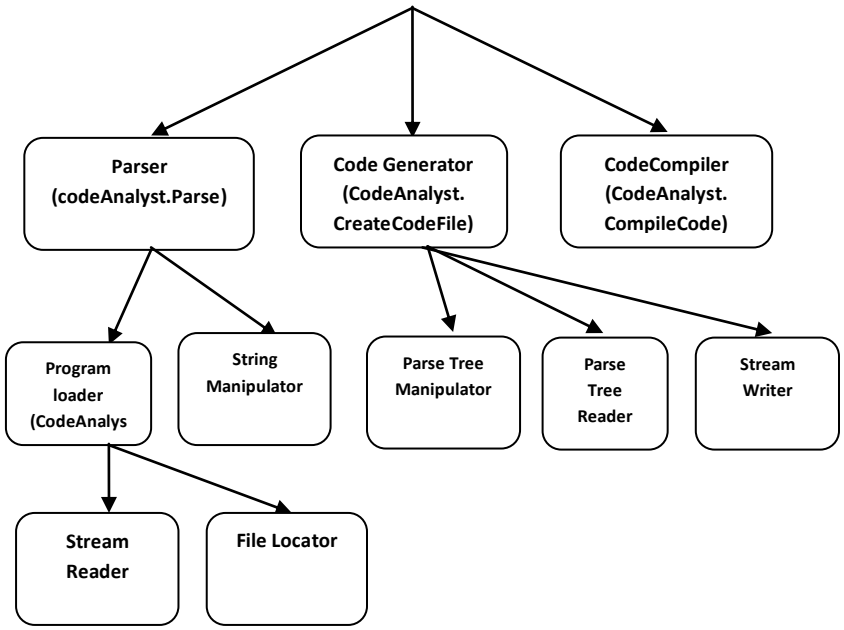

Figure: Hierarchy diagram of complexity evaluator

3.2 Major Modules and Functions

The complexity evaluator is a structured tool with the following major modules

(i) The CodeAnalyst Class

- involves the CodeCompiler to compile the C# code
- involves the CodeGenerator to generate the C# file from the parse tree generated by the CodeParser
- executes the executable file. The program starts by running the input program and extracting the required values after which the obtained values are displayed.
- Invokes the program loader to load the input program to be analysed.

(ii) The ProgramLoader is used by CodeAnalyst to obtain the loaded program

(iii) CompileCode compiles the generated C# code file to produce an executable file

(iv) CodeCompiler is used by the CodeAnalyst to compile the generated C# file

(v) Function: This class inherits from the Abstract Class ‘‘CodeBlock’’, defines other specific functions for operating on functions extracted from the input program

(vi) WholeUserCode: This class forms the top of the Parse tree for the input program. It keeps various lists such as:

- list of operands in the input
- list of operators in the input
- list of classes in the input
- total number of operands using traditional and modified Halstead
- total number of operators using traditional and modified Halstead

(vii) (CodeBlock: This class forms the base class for class ‘‘Function’’ and class ‘‘Property’’

4. Implementation

The following four cases are considered in a test code and the numerical results obtained are displayed below as Table I to

(i) Expanded recursive functions without considering the main function

(ii) Expanded recursive functions and translation of every occurence of multiplication and division to successive addition and substraction respectively without considering the main function

(iii) Expanded recursive function considering the main function

(iv) Expanded recursive function and translation of every occurence of multiplication and division to successive addition and subtraction respectively considering the main function

Often times, the main function of a program contributes little or nothing to the complexity of an algorithm; because mostly, main functions are concerned with input, output, and house-keeping functions. Hence, if the complexity of the main function is negligible then cases (i) and (ii) above are justified, otherwise cases (iii) and

(iv) are necessitated. The program which is analysed using our tool is given as Appendix

Table 4.1: Expanded Recursive Function Without Considering the Main Function.

	Program Volume V=(Nlogn) Log to Base 2	**Difficulty Level D=(n1/2)(N2/n2)**	**Effort E = (VD)**	**Time to Implement T = E/18**	**Program Level L = 1/D**	**Program Lenght N=N1 + N2**	**Program Vocabulary n = n1 + n2**
Traditional Halstead	69.8	6.0	418.6	23.3	0.2	21	10
Modified Halstead	229.2	20.0	4584.3	254.7	0.1	69	10
Modified Data	n1 (UOperators): 5 n2 (UOperands): 5 n(UOperators + UOperands): 10			N1 (UOperators): 29 N2 (UOperands): 40 N(UOperators + UOperands): 69			
Traditional Data	n1 (UOperators): 5 n2 (UOperands): 5 n(UOperators + UOperands): 10			N1 (UOperators): 9 N2 (UOperands): 12 N(UOperators + UOperands): 21			

Table 4.2: Expanded Recursive Function and Translation of Every Occurence of Multiplication and Division to Successive Addition or Subtraction Respectively Without Considering the Main Function.

	Program Volume V=(Nlogn) Log to Base 2	**Difficulty Level D=(n1/2)(N2/n2)**	**Effort E = (VD)**	**Time to Implement T = E/18**	**Program Level L = 1/D**	**Program Lenght N=N1 + N2**	**Program Vocabulary n = n1 + n2**
Traditional Halstead	69.76	6.00	418.56	23.25	0.17	21	10
Modified Halstead	458.12	7.63	3493.20	194.07	0.13	106	20
Modified Data	n1 (UOperators): 4 n2 (UOperands): 16 n(UOperators + UOperands): 20			N1 (UOperators): 45 N2 (UOperands): 61 N(UOperators + UOperands): 106			
Traditional Data	n1 (UOperators): 5 n2 (UOperands): 5 n(UOperators + UOperands): 10			N1 (UOperators): 9 N2 (UOperands): 12 N(UOperators + UOperands): 21			

Table 4.3: Expanded Recursive Function Considering the Main Function.

	Program Volume V=(Nlogn) Log to Base 2	**Difficulty Level D=(n1/2)(N2/n2)**	**Effort E = (VD)**	**Time to Implement T = E/18**	**Program Level L = 1/D**	**Program Lenght N=N1 + N2**	**Program Vocabulary n = n1 + n2**
Traditional Halstead	246	5	1292	72	0	59	18
Modified Halstead	446	12	5466	304	0	107	18
Modified Data	n1 (UOperators): 6 n2 (UOperands): 12 n(UOperators + UOperands): 18			N1 (UOperators): 58 N2 (UOperands): 49 N(UOperators + UOperands): 107			
Traditional Data	n1 (UOperators): 6 n2 (UOperands): 12 n(UOperators + UOperands): 18			N1 (UOperators): 38 N2 (UOperands): 21 N(UOperators + UOperands): 59			

Table 4.4: Expanded Recursive Function and Translation of Every Occurence of Multiplication and Division to Successive Addition or Subtraction Respectively Considering the Main Function.

	Program Volume V=(Nlogn) Log to Base 2	**Difficulty Level D=(n1/2)(N2/n2)**	**Effort E = (VD)**	**Time to Implement T = E/18**	**Program Level L = 1/D**	**Program Lenght N=N1 + N2**	**Program Vocabulary n = n1 + n2**
Traditional Halstead	246.0	5.3	1291.6	71.8	0.2	59	18
Modified Halstead	561.9	10.5	5899.4	327.7	0.1	130	20
Modified Data	n1 (UOperators): 5 n2 (UOperands): 15 n(UOperators + UOperands): 20				N1 (UOperators): 67 N2 (UOperands): 63 N(UOperators + UOperands): 130		
Traditional Data	n1 (UOperators): 6 n2 (UOperands): 12 n(UOperators + UOperands): 18				N1 (UOperators): 38 N2 (UOperands): 21 N(UOperators + UOperands): 59		

5. Analysis of Results

- In Table 4.1, Modified Halstead metrics are greater than the traditional Halstead metrics except in the case of program vocabulary where both are equal.
- In Table 4.2, Modified Halstead metrics are greater than the traditional Halstead metrics in all cases.
- In Table 4.3, Modified Halstead metrics are greater than the traditional Halstead metrics in five of the seven cases. In the remaining two cases the metrics are the same.
- In Table 4.4, Modified Halstead metrics are greater than the traditional Halstead metrics in all cases.

Implication of these results is that Modified Halstead measure yields accurate results. It shows that recursion is a more tasking activity than sequential processing. It further shows that multiplication and division are more involving and therefore more complex than addition and subtraction. Hence, this study has revealed the limitations of the traditional Halstead measure and has proposed an algorithm to overcome them.

6. Conclusion

From this study, it has been established that numerical solution of complexity of algorithm gives more detailed, more specific and hence more pragmatic results than those obtained by use of Big O notation. Furthermore, the numerical study reveals that the depth of recursion has an unnegligible effect when evaluating complexity of algorithm

Appendix 1: Test Code

```
using system;
class illustrate
{
        public int Power(int a, int Raisedto)
        {
                if (Raisedto < 1) return 1;
                else
                        return a * Power(a, Raisedto – 1);
        }
        public double Div(int a, int b)
        {
                if (b < 1) return 0;
                return (double)a/b;
        }
        public static void Main()
        {
                System console.WriteLine(‘‘Starting Test...’’);
                System console.WriteLine(‘‘.....’’);
                System console.WriteLine();
                System console.WriteLine(‘‘Please Input a value to find its Power:’’);
                System console.WriteLine(‘‘Power:                                {}’’;
                Power(int.Parse(System.Console.Readline()),
                int.Parse(System.Console.ReadLine())));
                System.Console.WriteLine();
                System.Console.Write(‘‘Please Input a value to find its division:’’);
                System console.WriteLine(‘‘Division:                             {}’’;
                Div(int.Parse(System.Console.ReadLine())));
                int.Parse(System.Console.ReadLine())));
                System.Console.WriteLine();
                System.Console.WriteLine(‘‘Press any Key to Exit...’’);
                System.Console.ReadKey(true);
        }
  }
```

References

Abu T.M., Abran A. and Ormandjieva O. (2005): COSMIC-FFP and Functional Complexity Measures: A study og their scales, units and scale types. Proceedings of the 15th International Workshop on Software Measurement, Montreal, Canada, pp. 209-225.

Anany L. (2003): Introduction to the Design and Analysis of Algorithms, pp 41-50, Addison-Wesley, Reading, Massachusetts.

Behrouz A. (2008): Cryptography and Network Security, McGraw-Hill International Edition, London.

Christos H. (1994): Computational Complexity, Addison-Wesley, Reading, Massachusetts.

info@verifysoft.net(2005): Halstead Metrics

Jukka K.N. (2003): Using Software Complexity Measures to Analyse Algorithms – An Experiment with the Shortest-Paths Algorithms; Computers and Operations Research 30, pp. 1121 – 1134, New York, U.S.A.

Glenn J. (2007): Computer Science; an Overview, Tenth Edition, Pearson Education, New Jersey, U.S.A.