Ilorin Journal of Science
Volume 1, Number 1, 2014, pp. 18 - 27 (Printed in Nigeria)
ISSN: 2408 – 4840 © 2014 Faculty of Physical Sciences, University of Ilorin
https://doi.org/10.54908/iljs.2014.01.01.002

ILORIN
JOURNAL OF SCEINCE

ILJS-14-042

Efficiency Comparison of Some Selected Programming Languages

Aremu, D. R. and Salako, R. J.

Department of Computer Science, University of Ilorin, Ilorin, Nigeria.

Abstract

The choice of an algorithm comes down to a question of efficiency, which is a measure of complexity of an algorithm. This paper addresses the questions of how to measure and compare the complexities of different algorithms. The aim of the study was to analyze and to comparing the complexities of tree search algorithms using software metrics approach to discover the most efficient programming language for implementing the tree search algorithms. Implementations of the algorithms were carried out using C, C++, Pascal, Visual BASIC, and Java programming languages. Complexity appraisals of the algorithms were carried out after the analyses and it was discovered that the choice of programming language affects the complexity of the tree search algorithms. The results showed that Pascal is the most efficient language for implementing Breadth-first algorithm, Java is the best for Depth-first, Java or Pascal is the best for Depth-limited while C is the best programming language for implementing A-star tree search algorithm. Visual BASIC is the worst language for implementing the entire search algorithm. It was further revealed that the entire codes are structurally and logically simple.

Keywords: Complexity, Tree search, Halstead volume, Cyclomatic number, and Software,

1. Introduction

Recently, research has shown that different algorithms exist for solving a particular problem. It is however difficult to readily determine which algorithm is better than the other. Given a problem therefore, how can we find an efficient algorithm for its solution? How can we compare this algorithm with other algorithms? Questions of these types are of interest to programmers and to theoretically oriented computer scientists. The aim of this paper is to find out the most efficient language for implementing the tree search algorithms. The methodology used to achieve this aim involves: (i) implementing the tree search algorithms using C, C++, Pascal, Visual BASIC, and Java programming languages; and (ii) analyzing and comparing the complexities of the tree search algorithms by adopting the Halstead's volume and Cyclomatic number analysis measures. The result of complexity appraisals of the tree search algorithms showed that the choice of

Corresponding Author: Aremu, D.R.
Email: draremu2006@gmail.com

programming language affects the complexity of the tree search algorithm. Pascal programming language was found to be the most efficient language for implementing Breadth-first algorithm, while Java is the best for Depth-first algorithm, Java or Pascal is the best for Depth-limited while C is the best programming language for implementing A-star tree search algorithm. Visual BASIC is the worst language for implementing the entire search algorithm. It was further revealed that the entire codes are structurally and logically simple. The rest part of the paper is organized as follows: Section 2 presented the related work, while section 3 discussed the tee search algorithms models; in section 4, we discussed the analysis of the complexities of the tree search algorithms; while section 5 presented the results of the analysis, and section 6 concluded the paper.

2. Related Work

To compare the efficiency of algorithms, a measure of the degree of the difficulty of an algorithm called computational complexity was developed by Juris Hartmains and Richard Steams (Alfred et al., 1974).

We can express an approximation of a function using a mathematical notation called order of magnitude of a function, or Big-O notation. Theoretical analysis concentrates on a proportionality approach, expressing the complexity in terms of its relationships to some known functions such as: N, N^2, N^3, 2^N e.t.c. which are respectively linear, quadratic or double nested loop, triple-nested loop, and exponential running times. This type of analysis is known as asymptotic analysis (Kleinberg, 2005).

Empirical analysis focuses on the implementation complexity by using software complexity measures available. Complexities of tree search algorithms have been mostly evaluated either mathematically or by computing the computer execution time. Neither of the two approaches is good enough for practical and realistic purpose especially in the situation where more than one algorithm exists for solving a given problem or class of problems. There is a need therefore to seek for pragmatic means of computing complexity of algorithms. Empirical analysis focuses on the implementation complexity by using software complexity measures available. Implementation complexities which involves software metrics is a pragmatic field that arises out of attempts to estimate the amount of time it will take to code and maintain software.

Complexity of an algorithm is the determination of the amount of resources such as time and storage necessary to develop, maintain, and execute the algorithm. Other items to be considered under resources are: (a) Man-hours needed to supervise, comprehend code, test, maintain, and

change software, (b) Travel expenses, (c) the amount of re-used code modules, (d) Secretarial and technical support, etc. Prominent among the measured resources are; time and space complexities. Time complexity measures how much time the program take, while Space complexity measures how much storage the program need to develop, maintain, and execute it. 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. The choice between algorithms comes down to a question of efficiency. Which one takes the least amount of computing time?. Or which one was the jobs with the least amount of work are paramount questions asked by programmers (Nell, 2003). Given two or more software that solve a particular problem, a programmer is faced with the problem of the choice of the most efficient one in terms of quantitative measure of quality, understanding, difficulty of testing and maintenance, as well as the measure of ease of using the software.

Algorithms are frequently assessed by the execution time, memory demand, and by the accuracy or optimality of the results. For practical use, another important aspect is the implementation complex. An algorithm which is complex to implement required 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 et al., 2000).

Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimate for the resources needed by any algorithm which solve a given computational problem. These estimates provide an insight into reasonable direction of search of efficient algorithms (Jimmy, 2000).

Complexities of tree search algorithms have been mostly evaluated either mathematically or by computing the computer execution time. Neither of the two approaches is good enough for practical and realistic purpose especially in the situation where more than one algorithm exists for solving a given problem or class of problems. There is a need therefore to seek for pragmatic means of computing complexity of algorithms. Empirical analysis focuses on the implementation complexity by using software complexity measures available. Implementation complexities which involves software metrics is a pragmatic field that arises out of attempts to estimate the amount of time it will take to code and maintain software.

In the realm of software metrics, code is looked at as output of labour. The complexity of a piece of software is thought of in the same way as the complexity of an automobile; the number of parts and the nature of the assembly may affect the amount of labour and time needed to create the end product.

Parse and Oman (1995) applied a maintenance metrics index to measure the maintainability of C source code before and after maintenance activities. This technique allows the project engineers to track health of the code as it was being maintained. Maintainability is accessed but not in term of risk assessment.

Stark (1996) collected and analyzed metrics in the categories of customer satisfaction, cost, and schedule with the objective of focusing management's attention on improvement areas and tracking improvements over time. This approach aided management in deciding whether to include changes in the current release, with possible schedule slippage, or include the changes in the next release. However, the author did not relate these metrics to risk assessment.

Olabiyisi (2005) applied different software complexity measure such as Halstead metrics, and Cyclomatic number to a set of sorting algorithms. For the calculation of the complexity measurement, he developed a machine which is capable of finding the various implementation complexity values of algorithms written in different programming languages.

(Norman, 2001) shifted the emphasis from design and code metrics to metric that characterize the risk of making requirement changes. Although his software attributes can be difficult to deal with due to fuzzy requirement from which they are derived, the advantage of having early indicators of future soft are problems outweighs this inconvenience. He developed an approach for identifying the requirements change risk factors as predictors of reliability and maintainability problems. His case examples consist of twenty-four Space Shuttle change requests, nineteen risk factors, and the associated failure and software metrics.

3. Tree Search Algorithms Models

A tree is the collection of objects usually referred to as nodes with hierarchical relations defined on them. By manipulating the data structure, the tree is explored in different orders, for instance level by level (Breadth-first search) or reaching a leaf node first and backtracking (Depth-first search) e.t.c. (Thomas, 2000). In this project, the tree search algorithms are discussed as follows:

3.1 Breadth-First Search (BFS)

Breadth-first search (BFS) 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 neighbor – nodes and so on, until it finds the goal.

Algorithm of Breadth-First Search

```
procedure bfs (v)
        q: = make_queue( )
        enqueue (q, v)
```

```
mark v as visited
while q is not empty
v = dequeue (q)
process v
for all univisited vertices v' adjacent to v
        mark v' as visited
        enqueue (q, v')
```

3.2 **Depth-First Search** (DFS)

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 had not finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a last- in-first- out (LIFO) stack for expansion (Thomas, 2000). Time complexity of both algorithms are proportional to the number of vertices plus the number of edges in the graphs they traverse.

Algorithm of Depth-First Search

```
dfs (graph G)
{
        list L = empty
        tree T = empty
        choose a starting vertex x
        search (x)
        while (L is not empty)
                remove edge (v, w)from end of L
                if w not yet visited
                {
                        add (v, w) to T
                        search (w)
                }
}
search (vertex)
{
        visit v
        for each edge (v, w0
                add edge (v, w) to end of L
}
```

3.3 Depth-Limited Search

Like the normal depth-first search, depth-limited search is an uninformed search. It works exactly like depth-first search, but avoids its drawbacks regarding completeness by imposing a maximum limit on the depth of the search (Thomas, 2000).

Algorithm of Depth-Limited search

```
DLS (node, goal, depth)
{
        if (node = = goal)
                return node;
        else
        {
                stack ;= expand (node)
                while (stack is not empty)
                {
                        node' := pop (stack);
                        if (node' . depth () < depth);
                                DLS(node', goal, depth);
                        Else
;                               // no operation
                }
        }
}.
```

3.4 A* Search

A* (Pronounced 'A star') is a tree search algorithm that finds a path from a given initial node to a given goal node. It employs a heuristic estimate that ranks each node by an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate. The A* algorithm is therefore an example of a best-first search (Hart et al., 1968).

Algorithm of A* Search

```
function A* (start, goal)
    var closed := the empty set
    var q := make_queue 9path (star))
    while q is not empty
        var p:= remove_ first (q)
    var x:= the last node of p
    if x in closed
        continue
    if x= goal
        return p
    add x to closed
    foreach y in successors (p)
    if the last node of y not in closed
        enqueue (q,y)
```

4. Analysis of the Complexities of the tree search algorithms

.

This section presents the analysis of the complexities of the tree search algorithms by adopting the Halstead's volume and the Cyclomatic number analysis measures.

4.1 Halstead's Volume

The Halstead measures are based on four scalar numbers derived directly from a program's source code i.e. N_1, N_2, n_1, and n_2 which are respectively Total number of operators, Total number of operands, number distinct of operators and number of distinct operands. The Halstead measures are described as shown in table 1 below.

Table 1: The Halstead Measures.

Measure	Symbol	Formula
Program Length	N	$N = N_1 + N_2$
Program Vocabulary	N	$n = n_1 + n_2$
Program Volume	V	$V = N*(LOG_2n)$
Program Difficulty	D	$D = (n_1/_2)*(N_2/n_2)$
Program Effort	E	$E = D*V$

4.2. Cyclomatic Number

Using graph theory, the cyclomatic number is mathematically computed using the formula: v(G) = number of closed loops + 1.

4.3. Analysis of Codes Generated for the tree Search Algorithms

We used Halstead's volume and Cyclomatic number software complexity measures to evaluate the complexity of each of the tree search algorithms, and implemented them using C, C++, Pascal, Visual BASIC and Java programming languages. The results of the implementation are presented in tables 2-5 below, for comparison purposes.

5. Results of the Analysis

Table 2: Breadth-First Search Algorithm Results.

LANGUAGES	PROGRAM VOL (V)	PROGRAM DIFFIC (D)	PROGRAM EFFORT (E)	CYCLOMATIC NUMBER
C	733	20	14660	5
C++	723	18	13014	5
PASCAL	558	17	9486	3
Visual BASIC	1045	22	22990	6
JAVA	1059	28	29707	4

Table 3: Depth-First Search Algorithm Results.

LANGUAGES	PROGRAM VOL (V)	PROGRAM DIFFIC (D)	PROGRAM EFFORT (E)	CYCLOMATIC NUMBER
C	459	20	9180	5
C++	481	21	10101	5
PASCAL	454	11	4994	5
Visual BASIC	883	15	13245	6
JAVA	348	9	3235	5

Table 4: Depth-Limited Search Algorithm Results.

LANGUAGES	PROGRAM VOL (V)	PROGRAM DIFFIC (D)	PROGRAM EFFORT (E)	CYCLOMATIC NUMBER
C	595	21	12495	5
C++	544	19	10569	5
PASCAL	626	14	8764	5
Visual BASIC	1297	22	28534	7
JAVA	543	18	9589	5

Table 5: A-Star Search Algorithm Results

LANGUAGES	PROGRAM VOL (V)	PROGRAM DIFFIC (D)	PROGRAM EFFORT (E)	CYCLOMATIC NUMBER
C	936	19	17784	8
C++	1201	40	48040	8
PASCAL	1171	24	28104	6
Visual BASIC	1873	44	82412	9
JAVA	1416	14	19954	5

6. Conclusion

In this work, software complexity measures have been used to evaluate and compare the complexities of a set of tree search algorithms implemented in different languages in order to determine the best programming language for implementing each of the tree search algorithms. It was discovered from the results the analysis that the choice of programming language affects the complexity of tree search algorithms. It was discovered that Pascal programming language is the best language for implementing breadth-first search, Java language was the best for implementing depth-first search, while depth-limited search algorithm is best implemented in Java and Pascal programming languages. C and Java languages are two candidate languages contending for A-star tree search algorithm.

It was very apparent that the performance characteristics of Pascal and Java stand out very clear. The biggest myth about Pascal is that it is a language without power but it is very convincing that Pascal language, though an old language not commonly used in large companies is highly efficient for implementing tree search algorithm. Visual BASIC; a worst programming language for implementing the entire tree search algorithm was built for only windows (i.e not cross-platform capable)

We concluded that the chosen search algorithms coded in five languages are logically and structurally simple. This conclusion is reached because none of the cyclomatic complexity risk evaluation numbers measured up to 10 which is the bench -mark for risk evaluation. This implies that decision statements such as IF/THEN, IF/ELSE e.t.c. and control statements such as DO/WHILE, DO/UNTIL e.t.c. in the source codes are not too many or very manageable in number. It can therefore be deduced that testing and maintenance could be done with ease in the

codes generated using the entire programming languages. The cyclomatic complexity risk evaluation result also means that modules in the source codes are highly cohesive. Cohesiveness or binding refers to the relationships among pieces of modules. High cohesion is characterized by a module that performs one distinct procedural task. Effort in future research will be geared towards more complicated algorithms.

Acknowledgement

The authors: Aremu D. R. & Salako R.J. acknowledge the efforts of the reviewers in improving the quality of the manuscript. Also we are grateful to the Faculty of Physical Sciences, University of Ilorin, Ilorin, Nigeria for providing us the opportunity to publish this article in the maiden edition which was sponsored by the faculty.

References

Alfred, V.A., John, E. & Jeffrey, P.U. (1974). *The Design and Analysis of Computer Algorithms.* New York. Addison- Wesley Publishing Company, 78-98.

Hart, P.E. Nilsson, N.J. & Raphael, B. (1968). Correction to: A Formal for the Heuristic Determination of Minimum Cost Paths, SIGART Newsletter,**37**, 28-29.

Jimmy, W. (2000). Software size Measurement: A framework for counting source code maintenance Activities, International Conference on software Maintenance Activities, Opio (Nice), France, **12**, 295-303.

Kleinberg, (2005). *Algorithm Design.* Pearson Addison-Wesley. Hongkong, 29-35.

McCabe, T.J. (1994): Software complexity. *Crosstalk*, 7, 12.

Nell, D. (2003). *C++ Plus Data Structures*. Jones and Batlett Publishers, Canada.

Norman, F.S. (2001). Investigation of the RISK TO software Reliability and Maintenability of Requirements Changes. Proceedings of the international Conference on Software Maintenance. Florence, Italy, 127-136.

Olabiyisi, S.O. (2005). *Universal Machine for Complexity Measurement of Computer Programs,* Ph.D. Thesis,Department of Pure And Applied Mathematics, LAUTECH, Ogbomoso.

Oman, P & Hagemeister, J. (*1994*): Construction and Testing of Polynomials Predicting Software maintainability. Jour. of System and Software, 251-266.

Stark, P.B. (1996). A few considerations for ascribing statistical significance to earthquake predictions, Geophysical Research Letters, 23, 1399–1402.

Thomas, H.C. (2000). *Introduction to Algorithms*. McGraw-Hill companies, New York, 67-72.