COMPARATIVE STUDY OF SOFTWARE COMPLEXITIES OF TREE SEARCH ALGORITHMS

Salako R.J. and Aremu D.R.

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

Abstract - *This research paper studies the complexities of four tree search algorithms in order to determine the most efficient programming language for implementing each of the algorithms. Each the tree search algorithm was implemented in C, C++, Pascal, and Visual BASIC programming languages. The codes were empirically analysed using Halstead Volume and Cyclomatic number. The result of the analysis revealed that Pascal programming language is the best language for implementing breadth-first, depth-first, and depth-limited search algorithms while C language was isolated as the best for implementing A-Star search algorithm.*

Keywords: Tree search, implementation, complexity metrics, software complexity

1. Introduction

Given two or more software that solve a particular problem, a programmer is faced with the problem of 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. The analysis of algorithm is a major task in computing. A computer scientist, most especially a programmer, who is faced with the problem of choosing an appropriate algorithm to solve his problem from myriad of available ones, may have his problem solved by analyzing the complexity of each of these algorithms in order to know the most efficient one. This is a nontrivial issue that leads to the analysis of algorithms and the means by which they can be compared. The aim of this paper is to study the complexities of four tree search algorithms in order to determine the most efficient programming language for implementing each of the algorithms. The objective to achieve this aim was to study tree search algorithms such as breadth-first, depth-first, and depth-limited, and to implement each search algorithm in C C++, Pascal, and Visual Basic programming languages. We analysed the codes of each of the algorithms empirically using Halstead Volume and Cyclomatic number. The result of the analysis showed that Pascal programming language is the best language for implementing breadth-first, depth-first, and depth-limited search algorithms, while C language was best for implementing A-Star search algorithm.

The rest of the paper is organized as follows: Section 2 presented the related work, while section 3 discussed complexity measurement. In section 4, we discussed the tree search algorithms. Section 5 presented the results of the complexity measurements, while section 6 concluded the paper.

2. Related Work

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 Waks, (2000).

Algorithm can be studied theoretically or empirically. Theoretical analysis allows mathematical proofs of the execution time of algorithms which studies how an algorithm behaves with typical inputs. It is therefore tend to focus on the execution time and optimality of the result Sedgewick, (1995). 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. 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.

Okeyinka, (2003) designed a scan machine (software) that could evaluate the complexity of computer programs written in pascal language. The tool designed can be used to identify the most efficient algorithm from among myriad of algorithms solving the same problem.

3. Complexity Measurement

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. In this section, we presented a brief review of algorithm complexities measurement.

3.1 HALSTEAD Complexity Measure

Halstead complexity measure was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module (Halstead, 1977). The Halstead measures are based on four scalar numbers derived directly from a program's source code. n_1 = the number of distinct operators, n_2 = the number of distinct operands

N_1 = the total number of operators, N_2 = total number of operands

Table 1 : Complexity Measurement

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

3.2 Cyclomatic Complexity Measure

Cyclomatic complexity directly measures the number of linearly independent paths through a program's source code. Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to the program. A directed edge connects two nodes if the second command might be executed immediately after the first command. Cyclomatic complexity, v(G), is derived from a flow graph and is mathematically computed using graph theory. More simply stated, it is found by determining the number of decision statements in a program: v (G) = e – n + p

v(G) is a cyclomatic complexity.

e is the number of edges in the flow graph

n is the number of nodes in the flow graph, and p is the connected components.

4. Tree Search Algorithms

In this section, we presented tree search algorithms for consideration of algorithm complexity measurement.

4.1 Breadth-first Search

A tree-search in which the adjacency lists of the vertices of *T* are considered on a first-come first-served basis, that is, in increasing order of their time of incorporation into *T*, is known as *breadth-first search*. In order to implement this algorithm efficiently, vertices in the tree are kept in a *queue*; this is just a list *Q* which is updated either by adding a new element to one end (the *tail* of *Q*) or removing an element from the other end (the *head* of *Q*). At any moment, the queue *Q* comprises all vertices from which the current tree could potentially be grown. Initially, at time $t = 0$, the queue *Q* is empty. Whenever a new vertex is added to the tree, it joins *Q*. At each stage, the adjacency list of the vertex at the head of *Q* is scanned for a neighbour to add to the tree. If every neighbour is already in the tree, this vertex is removed from *Q*. The algorithm terminates when *Q* is once more empty.

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 unvisited vertices v' adjacent to v
                mark v' as visited
                enqueue (q, v')
```

(Thomas H Cormen et al, 2001).

4.2 Depth-first Search

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.

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
}   (Thomas H Cormen el al, 2001)
```

4.3 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. Even if the search cold still expand a vertex beyond that depth, it will not do so and thereby it will not follow infinitely deep paths or get stuck in cycles.

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
                }
        }
```

4.4 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 P.E. et, 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
        f 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)
```

5 Results

The results of complexities measurement for each of the above tree search algorithms implemented using C, C++, Pascal, and Visual Basic programming languages are presented as shown in Tables 2 – 5 bellow.

Table 2: Results For Breadth First Tree Search Algorithm

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

Table 3: Results For Depth-First Tree Search Algorithm

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

Table 4: Results For Depth-Limited Search ALGORITHM

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

Table 5: A-Star Search Algorithm Complexity Measures

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

6 Conclusion

It was observed from the results of implementation (Tables 2 – 5) that Pascal programming language perform best for implementing breadth-first search, depth-first search, and depth-limited search algorithms, while C programming language is the best implementation language for A* search algorithm. We therefore conclude from these results that Pascal programming language is the best language for implementing breadth-first search, depth-first search, and depth-limited search algorithms but C language is the best implementation language for A* search algorithm.

Furthermore, it is apparently concluded that the choice of programming languages affects the complexities of programs of tree search algorithms.

References

Akanmu, T.A.(2009): An explanatory study of software complexity measure of Breadth-first search Algorithm. Journal of Science& Technology vol 1

Alfred, V.A., John, E, and Jeffrey, P.U(1974): The Design And Analysis Of Computer Algorithms. New York. Addison-Wesley Publishing Company.

Floyd, R.W. (1962): "Algorithm 97: Shortest Path" Communication of the ACM5 (6) 345. DOI: 10. 1145/367766.368168.
Geer, Daniel et al (2003): Cyber security: The cost of monopoly(PDF)2004

Halstead, M (1977): "Elements of software science, operating, and Programming systems" series volume 7. New York, NY: Elsevier, 1977.

Hart, P.E. Nilsson, N.J, and Raphael, B.(1968): Correction to: "A Formal For the Heuristics Determination Of Minimum Cost Paths", SIGART Newsletter 37: pp. 28-29

McCabe, T.J.(1994): Software complexity. Crosstalk, vol 7, no 12.
Okeyinka, E.A. (2000): Design of a scan Machine for complexity Measure of computer program. Journal of Science & Technology vol 1 No 1 pp70-77

Olabiisi, S.O.(2005): ''Universal Machine For Complexity Measurement Of Computer Programs'' PhD Thesis, Department Of Pure And Applied Mathematics, LAUTECH, Ogbomoso.

Oman, P and Hagemeister, J.: Construction And Testing of Polynomials Predicting Software maintainability Journal Of System And Software 24, March 1994: 251-266

Robert E. Park(2006): Software Size Measurement: A framework For Counting Source Statements. Technical Report CMU/SEI-92-TR-20

Russell, S.J., Norvig P (2003): Artificial Intelligence: A modern approach, pp. 97-104

Shola, P.B.(2008): Data structure And Algorithm Using C++. Reflect Publishing house, Ibadan.

Thomas, H.C. (2000): Introduction to algorithms. McGraw-Hill companies, New York.