RESEARCH ON THE COMPLEXITY MEASUREMENT TECHNOLOGY OF SOFTWARE STRUCTURE BASED ON AST

Qiao LIPING[1], Li JING[2], Song YAQING[3]

With the increasing scale and complexity of software, people are demanding more and more analysis and measurement of software complexity, on this basis, this paper proposes an approach to measure software structure complexity based on abstract syntax tree (AST). The method firstly removes the redundant code information by preprocessing and carries out lexical semantic analysis to generate the corresponding AST; On the AST traversal basis, secondly records the frequency of the program key, and gets the sequence of program attributes and method calls by the information marking algorithm, finally, obtains the measurement results by the methods of the Line Count metric calculation, McCabe metric calculation and Halsted metric calculation. The experimental results show that this method can effectively measure the complexity of software structure.

Keywords: Abstract Syntax Tree, Software Complexity, Complexity Measure, Program Structure

1. Introduction

Now society has entered the era of big data information, Computer software has changed enormously in terms of size and complexity. According to authoritative statistics, software complexity is the main cause of software errors. When software complexity exceeds a certain limit, errors and failures in the software will rise rapidly and even cause failure in software development. At present, the research on software complexity analysis technology is not yet mature interiorly, and it is even on the traditional method of traversing pile. In this context, this paper puts forward a research on the complexity measurement technology of program structure based on AST, which is the most stable analysis result of the program codes, this method can be more effective and more accurate to analyze the structure complexity of the target program, especially for national defense, military industry, and aerospace software which require high accuracy and stability of the software [1-2].

[1] Department of Information Engineering, Xingtai Polytechnic College, Xingtai, Hebei, China, e-mail: cheast@163.com
[2] Department of Information Engineering, Xingtai Polytechnic College, Xingtai, Hebei, China
[3] Department of Information Engineering, Xingtai Polytechnic College, Xingtai, Hebei, China

It is of great significance to analyze and research the Software complexity analysis technology which is a key point and a difficult point, during the software process. Today, the domain, scale, and cost of computer software have increased dramatically, but the stability and reliability of software have not improved [3]. In order to change this situation, people begin to study and analyze the software complexity which is a measure of resource depletion in software development, software maintenance, and software usage process [4].

At present, the research on the complexity analysis of program structure is not mature. Some technologies are also in the second stage of source code analysis, which is a centralized analysis phase; they mainly use code piling and path traversal to get the analysis data. This method can reflect the complexity of the program structure to a certain extent, but it does not get the most direct information from the code structure, so it is not timely to deal with errors introduced by developers in the program. However, the technology based on the AST proposed by this paper, directly established the static analysis basis of code, and got the code structure information in time, which can comprehensively and accurately analyze the complexity of program structure.

This paper firstly introduces the parameters of the software structure complexity measure, and gives the simulation function of the complexity gradient, then obtains the program control flow by analyzing the structure of the AST, and designs a program analysis algorithm to record all the analysis data. On the basis of the algorithm, this paper uses McCabe metric, Halstead metric and code line metric calculation methods to measure the complexity of program code, finally, compares the similarity between the real results of the code complexity and the measurement results used by complexity measurement technology based on AST

2. Software Structure Complexity

Software structure complexity which is the inherent attribute of software mainly refers to the complexity of program code. Software product is an invisible logic product, and its development process is the complex thinking process of human brain. During the software development process, the complexity changes gradually, and the mutation occurs after reaching a certain value gradually. In this process, there are stable and unstable states. The complexity of software, from user's demand to the final product, is actually the process of function transformation. Set this function as:

$$y = f(x_{1,} x_{2,} \ldots x_n) \tag{1}$$

Among the formula (1), y is a function of time t; in this transformation process, it is affected by many conditions $(x_{1,} \ldots x_n)$ effect. Among the conditions, x_1 represents the size of the program, x_2 represents the difficulty of the development program, x_3 stands for program structure, x_4 stands for program intelligence, x_5 stands for the change in requirements, and x_6 stands for other factors, then (1) is equivalent to (2).

$$y = f(x_1, x_2, x_3, x_4, x_5, x_6) \tag{2}$$

Since y is a function of the time t variable, after the derivative of the time t, we get the formula (3):

$$\frac{dy}{dt} = f_t^{'}(x_{1,} x_{2,} x_{3,} x_{4,} x_{5,} x_6) \tag{3}$$

The formula (3) represents the rate at which the function $f(x_1, x_2, x_3, x_4, x_5, x_6)$ varies from time t. When the function y has a mutation, the function curve will have an inflection point. As a result, we know that software complexity varies from development time and even changes. Therefore, the measurement of software complexity is definitely not evaluated after the completion of software development, but is constantly carried out in the entire development process [5-6]

3. Program Analysis Algorithm Based on AST

The measurement of software complexity is based on the analysis of program structure. The more thorough we analyze the procedure, the more accurate the result of complexity measurement will be. In the Program parsing algorithm based on AST, we first study the formation and structure of the abstract syntax tree and the information contained in the nodes of the tree. Then, we use the Information Mark Parsing algorithm to traverse all the nodes, and form the hash table data model, the model records complex parameters of program structure, which is used as a data base for subsequent measurements [7].

3.1 The Abstract Syntax Tree

Abstract syntax tree is the product of parse tree after lexical analysis and syntax analysis of program understanding. The static analysis will translate the abstract syntax tree according to the actual needs and generate the data which the static analysis needs finally, such as, the program control flow chart, the data flow diagram, the function call diagram and so on. Thus, the abstract syntax tree can be used either as and output of intermediate results or as an input to other data forms [8-10], the abstract syntax tree is a graphical representation of a program's syntax structure, and the nodes in the syntax tree are derived directly from the rules of the grammar. Fig. 1 provides an abstract syntax tree for some test code.

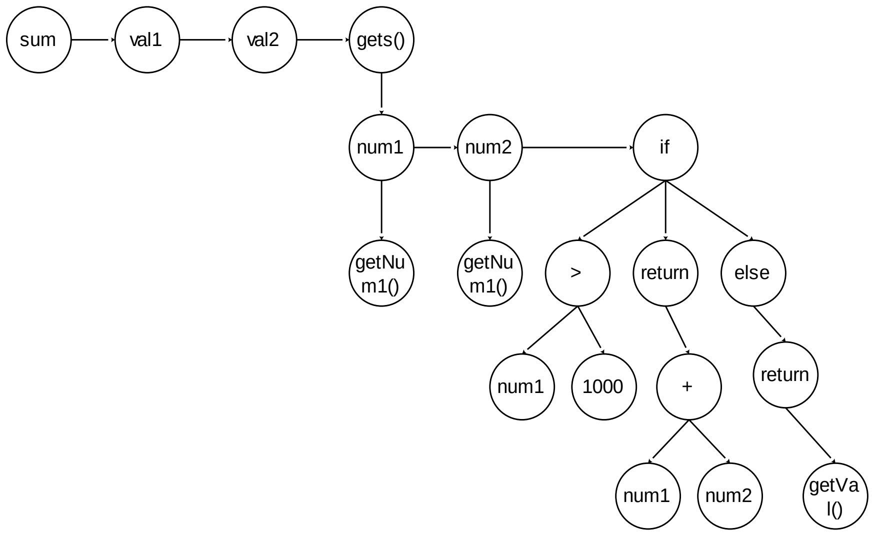

Fig.1. The Sketch map of abstract syntax tree

As seen in Fig.1, the operands in program codes usually are leaf nodes, and their operators are used as parent nodes. The information about the nodes of an abstract syntax tree can be roughly divided into three categories: 1. the name of a word code represented by a node, also known as the label of a node; 2. attribute of label which can be stored in the nodes or stored in the symbol table; 3. several pointers to their child nodes. The abstract syntax tree consists of leaf nodes that represent a non - reserved word terminator and intermediate nodes that represent a syntax structure.

3.2. Information Mark Parsing Algorithm

The abstract syntax tree resolves the key information in the program into a tree structure, and clearly describes how many branches and how many loops are in the program control structure. According to the above analysis, it is easy to obtain the measurable data basis of the program structure complexity, by using the information mark parsing algorithm.

Procedural language, like human language, is the most important way to obtain the key information in a language if you want to know the complexity of the language. For example, a book written in human language is compared to software developed in Procedural language, the language structure and the key information between the two are almost similar, details are shown in Table 1:

Table 1

Comparison of human language and procedural language

Term	Human Language	procedural language
Product	book	software
Product Structure	chapter	sub-system
	paragraph	modular

	section	assembly
Product composition	Sentences, phrases	Operator, expression
	subject	object
	Predicate	Event
	noun	variable
	verb	Method
	Modifiers	attribute

We know that the information rules of human language conform to the law of Zipf. For example, there is an article containing n words. We sort and classify these words according to the frequency of their occurrence, and use the letter r to indicate the vocabulary number, we will get the formula:

$$n = \sum_{r=1}^{s} n_r \tag{4}$$

Among them, s is the total number of lexical categories, n_r is the number of category r words, if we use the f_r to indicate the frequency of the category r words, so:

$$f_r = \frac{n_r}{n} \tag{5}$$

We know that f_r and r are linear, by the law of Zipf, then:

$$f_r \,.\, r = C \tag{6}$$

According to formulas (5) and (6), we can obtain:

$$n_r = \frac{C\,.\,n}{r} \tag{7}$$

If we interpret frequency f_r as the probability of the occurrence of words, we can see from the definition of probability:

$$\sum_{r=1}^{s} f_r = \sum_{r=1}^{s} \frac{n_r}{n} = 1 \tag{8}$$

According to formulas (7), we get:

$$\sum_{r=1}^{s} n_r = C\,.\,n \sum_{r=1}^{s} \frac{1}{r} \tag{9}$$

We expand the formula (9), then:

$$\sum_{r=1}^{s} \frac{1}{r} = 0.5772 + \ln t + \frac{1}{2s} - \frac{1}{2s(s+1)} \cdots\cdots \tag{10}$$

$$C \approx \frac{1}{0.5772 + \ln s} \tag{11}$$

Now we suppose that we already knew the total number of lexical categories, and we can estimate the value C, $r_{max} = s$. In addition, we suppose $n_{r_{max}} = 1$. It means that the smallest probability word occurs only once. According to formulas (11), we get:

$$C = \frac{s}{n} \tag{12}$$

Combining formulas (11) and (12), we can obtain formula (13) for calculating the total number of words:

$$n = s(0.5772 + \ln s) \tag{13}$$

According to the above structure comparison of human language and procedural language, we know that the program language also accords with the formula (13), if a program code contains 500 types of operands and operators, then the calculation formula for the length of the program will be:

$$n = 500(0.5772 + \ln 500) = 3396$$

Human language and programming language are similar in both ways of expression and purpose of expression. Therefore, they have the same law of information about the expression process. We can guide the information entropy of the information source according to Shannon theory.

$$H(U) = E[-\log_2 p_i] = -\sum_{i=1}^{n} p_i \log_2 p_i = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} \tag{14}$$

p_i is the probability of the occurrence of information, If the program to be measured is a sequence of symbols that are randomly taken out of the alphabet consisting of η_1 operators and η_2 operands, and the probability that each symbol is taken out of the alphabet is equal, it is $\frac{1}{\eta_1 + \eta_2}$, according to formulas (13), we get:

$$H(U) = \sum_{i=1}^{N} \sum_{j=1}^{s} p_{ij} \log_2 \frac{1}{p_{ij}} = N \log_2(\eta_1 + \eta_2) \tag{15}$$

According to Zipf's law, the probability that each symbol is taken out of the alphabet of operands and operators is not equal. According to the formula (12), the formula (15) can be converted to

$$H(U) = \frac{N}{\ln 2}\left[\frac{(\ln s)^2}{2(\ln s + 7/12)} + \ln(\ln s + 7/12)\right] \tag{16}$$

If s>100, we can omit the 7/12, and $\ln s = \ln 2 . \log_2 s$, thus the formula (16 can be reduced to:

$$H = H(U) \approx N\log_2(\sqrt{s} + \ln s) \tag{17}$$

If the program code is long and the S is large, then $\ln s = \ln 2 . \log_2 s$, we can obtain the formula:

$$H = \frac{N}{2}\log_2 s = \frac{N}{2}\log_2(\eta_1 + \eta_2) \tag{18}$$

According to the information theory, when the probability of each message in the information source appears, H is the largest. From the formula (18), we know that the complexity of the program code is directly related to the times of operands and the operators appear in the code. As a result, the program markup parsing algorithm traverses the abstract syntax tree in a quantized way, during the traversal [11-13]; we use the key value pairs at hash table to record the frequency of the operands and the operators. The storage structure of AST nodes is attached to the appendix.

In the traversal of the above algorithm, we set up the variable *infoMark* to mark the program information about the software complexity statistics. The leaf nodes of an abstract syntax tree are generally operands, and branch nodes are generally operators that connect leaf nodes. Through this parsing algorithm, we obtain a hash table containing complex parameters of the program.

4. Complexity Measurement of Software Structure

The research of software complexity analysis technology should be applied to the research of one or more software complexity measurement algorithms, so the most important problem is the data measurement metrics. Based on program comprehension, these metrics are directly or indirectly from the abstract syntax tree, according to the results of the information mark parsing algorithm, we can measure the structure complexity of the program by using two kinds of methods: the McCabe structure complexity measurement method and the Halstead software science measurement method, the McCabe complexity is mainly a measure of the complexity of software program structure. It needs to analyze the program control flow chart of software, so it belongs to the indirect call from the abstract syntax tree. The Halstead complexity uses the operands and operators in the program as a statistical object. It belongs to the direct call from the abstract syntax tree. In addition, we can statistic the number of code lines, the blank lines, and the commented lines of a comprehensive analysis [14-15].

4.1. McCabe Metric Calculation Method

McCabe complexity is essentially a measure of the complexity of the program topology. It needs to analyze the software program control flow chart, so it belongs to the indirect call to the abstract syntax tree. Fig. 2 illustrates the function call relationship based on the McCabe metric of AST.

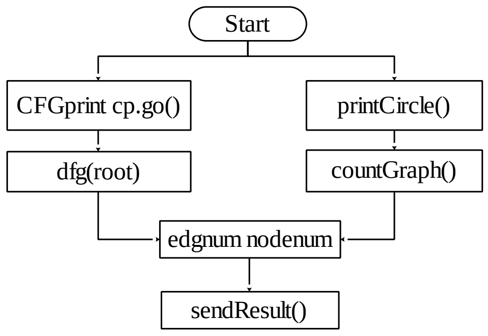

Fig. 2 Function Call Relations

When the McCabe complexity measuring, the object *cp* of the type *CFGprint* is firstly generated, we initialize some data by the function *go* of the object *cp*. Then we call the *dfg (root)* function, which implements recursive traversal of the program control flow, so that, we can analyze the number of arcs and the number of nodes in the control flow graph, we mark these two values with variables edgnum and nodenum, and then we call the function *printCircle* and the function *countGraph* to calculate the complexity by the variables *edgnum* and *nodenum*. Finally, the output is computed by using the function *sendResult*. Specific algorithm codes are attached to the appendix

4.2. Halstead Metric Calculation Method

The Halstead complexity uses the operands and operators in the program as a statistical object. The Information Mark Parsing Algorithm described earlier in this article has parsed the parameters of the operators and operands in the program [16-18]. We know that all the analysis results from the AST are stored in the hash table, on this basis, we can carry out the calculation of Halstead complexity, and its specific algorithm is attached to the appendix.

4.3 Experimental Analysis of the Complexity Measurement Technology

We see that, our measurement of the complexity of software structures is primarily the complexity of program code. According to the AST and the Information Mark Parsing Algorithm, we design the experimental test flow, as shown in Fig. 3

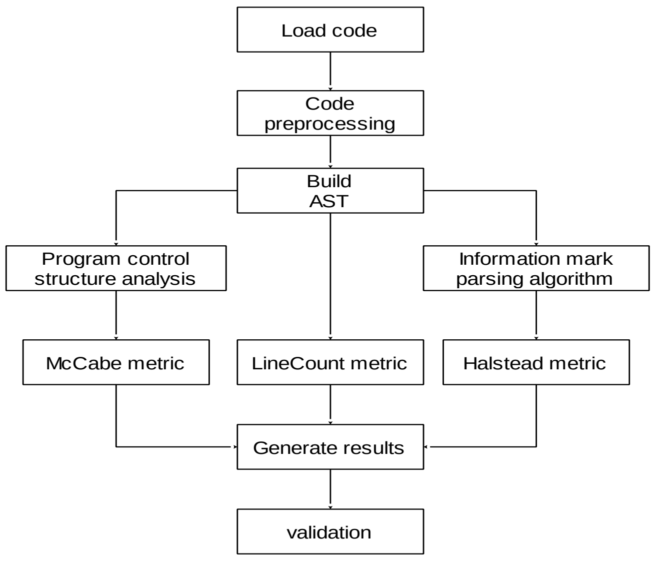

Fig. 3 Experiment flow design

During this experimental process, we first load the test code named test.c, and generate the AST; then on the basis of the AST, we obtain metadata for the program complexity analysis, by the Information Mark Parsing Algorithm and program control structure analysis. According to these metadata, we get the McCabe metric results, the Line Count metric results, and the Halstead metric results of the test code. Finally, we compare the measurement results with the actual complex metrics of the test code, the results are as shown in Table 2.

Table 2

Comparison table of experimental data with actual data

Code Lines	Experiment Data	Real Data	Accuracy Rate	Result
Total Lines	327	327	100%	Accurate
Blank Lines	13	13	100%	Accurate
Comment lines	21	21	100%	Accurate
McCabe	7	7	100%	Accurate
Halstead	97.26	97.33	99.92%	Accurate
Normalization	2.53	2.534	99.84%	Accurate

From the table, we can see that the results of the calculation using the software architecture complexity metric technology are almost identical with the actual data. Especially, in aspects of Line Count Metrics and McCabe metrics, they are exactly the same; and in aspects of Halstead metrics and Normalization metrics, the accuracy is close to 99.9%. To clarify the consistency of the two kinds of data, we draw their histogram comparison and line chart comparison, as shown in Fig. 4:

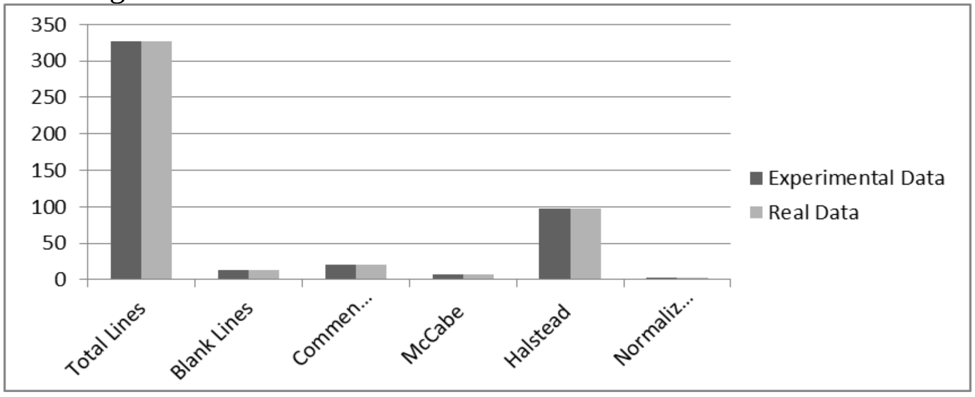

Fig.4.The histogram between experimental data and real data

In the histogram, we see that the experimental data are almost identical to the actual data onto the Y axis, which is also the numerical axis. These results furtherly prove the rationality and accuracy of the techniques under study.

5. Conclusion

The technology of Software complexity measurement technique based on AST used the abstract syntax tree which is a key factor as the main source of the program code complexity analysis. After the preprocessing of the irrelevant information about the program code, the efficiency of the abstract syntax tree analysis will be further improved. We use the information mark parsing algorithm to get the operands and operators from the program code and use the hash table structure to save the analysis result by key value pairs. This data storage method will greatly improve the speed of data reading, and directly affect the efficiency of subsequent complexity computing methods.

In this paper, we extract the required parameter factors of code complexity analysis, by abstract the syntax tree, and use the Line Count metric calculation method, the McCabe metric calculation method and the Halstead metric calculation method to analyze and calculate the program structure of software and give a comprehensive analysis report finally. In the end of this paper, we establish the environment of application tests, and design the application test flow in detail, and then test the complexity metric for the randomly selected program code. The analysis of the experimental results shows the consistency between the system test results and the program complexity analysis results; therefore, from the point of view of practical application, it proved that the technology of Software complexity measurement technique based on AST is accurate and feasible.

REFERENCES

[1]. *Nalinee Sophatsathit*, Complexity Measure of Software Composition Framework. Journal of Software Engineering and Applications, (2017), No.4, pp.324-337

[2]. *Jagvir Brar; Hans van der Meij*, Complex software training: Harnessing and optimizing video instruction. Computers in Human Behavior, (2015), **vol**.70, pp.475-485.

[3]. *Einollah Pira; Vahid Rafe; Amin Nikanjam*, Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm. Journal of Systems and Software, (2017), **vol**.131, pp.181-200.

[4]. *CherylL.Coyle; Mary Peterson*, Learnability Testing of a Complex Software Application.Design, User Experience, and Usability: Novel User Experiences, (2016), **vol**.9747, pp.560-568.

[5]. *José Roberto C. Piqueira*, Weighting order and disorder on complexity measures. Journal of Taibah University for Science, (2017), **vol**.11, No.2, pp.337-343.

[6]. *Ning Cai*.On quantitatively measuring controllability of complex networks. Physica A: Statistical Mechanics and its Applications, (2017), **vol**.474, pp.282-292.

[7]. *Il Hong Suh; Sang Hyoung Lee; Nam Jun Cho; Woo Young* Kwon.Measuring motion significance and motion complexity. Information Sciences, (2017), **vol**.388, pp.84-98.

[8]. *Kimio Kuramitsu1). Fast, Flexible,* and Declarative Construction of Abstract Syntax Trees with PEGs. Journal of Information Processing, (2016), **vol**.24, No.1, pp.123-131

[9]. *Hiroshi Kikuchi; Takaaki Goto; Mitsuo Wakatsuki; Tetsuro Nishino*, A Source Code Plagiarism Detecting Method Using Sequence Alignment with Abstract Syntax Tree Elements. International Journal of Software Innovation, (2015), **vol**.3, No.3, pp.41-56.

[10]. *Emma Söderberg; Torbjörn Ekman; Görel Hedin; Eva Magnusson*, Extensible intraprocedural flow analysis at the abstract syntax tree level. Science of Computer Programming, (2013), **vol**.78, No.10 pp.1809-1827.

[11]. *Deqiang Fu; Yanyan Xu; Haoran Yu; Boyang Yang*, WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Code Plagiarism Detection.Scientific Programming, (2017), **vol**.2017, doi:10.1155/2017/7809047.

[12]. *Wafaa S. Sayed; Hossam A. H. Fahmy,* What are the Correct Results for the Special Values of the Operands of the Power Operation?; ACM Transactions on Mathematical Software,(2016), **vol**.42,No.2,doi:10.1145/2809783.

[13]. *Yalin Chen; Jamie I. D. Campbell*, Operator and operand preview effects in simple addition and multiplication: A comparison of Canadian and Chinese adults. Journal of Cognitive Psychology, (2015), **vol**.27, No.3 pp.326-334.

[14]. *José Roberto C. Piqueira*, Weighting order and disorder on complexity measures. Journal of Taibah University for Science (2017), **vol**.11, No.2 pp.337-343

[15]. *Mortoza LP; Piqueira JR.*, Measuring complexity in Brazilian economic crises...PLoS One. (2017), **Vol**.12, No.3, pp e0173280

[16]. *NicholasV.Sarlis.* Entropy in Natural Time and the Associated Complexity Measures. Entropy. (2017), **Vol**.19, No.4, pp: 177.

[17]. *Measuring Pregnancy Intention*: The Complexity of Comparison., Perspect Sex Reprod Health (2017), **Vol**. 49, No.1, pp: 69-70.

[18]. *Nalinee Sophatsathit.*Complexity Measure of Software Composition Framework.Journal of Software Engineering and Applications, (2017), No.4, pp:324-337.

APPENDIX

Code 1. The storage structure of AST nodes

```
 Struct Type
{         BTNode* ptr;    // Pointer to node
      Enum {0, 1, 2} visitMark; // Access mode flag
   Enum {0, 1} infoMark; // Information type marker
  String key;      // the key name of a hash table
};              /* Node pointer type with mark field */
```

Code 2. Mccabe method algorithm codes

```
Int edgnum=0, nodenum=0;        // the number of arcs and the number of nodes
/*dfg () is used to statistic the number of nodes and arcs*/
Void dfg (NextPtr node)
{   Nodenum++;      //the number of nodes add 1
      (node.cont ()? seenCont: seen).add (node.stmt ());
   NextPtrList successors;
   node.stmt () ->getSuccessors (successors, node.cont ());
  If (! rootNodePrinted)
{  Edgnum++;    // the number of arcs add 1
          Nodenum++; // the number of nodes add 1
     RootNodePrinted = true;
  }
For (int i=0; i < successors.length (); i++)
{ NextPtr succ = successors[i];
    Bool haveSeenIt = (succ.cont ()? seenCont: seen).contains (succ.stmt ());
    Edgnum++;
    If (haveSeenIt)      ;
    Else
      Dfg (succ);   // visit the succeeding node of this node
```

Code 3. Healstead method algorithm codes

```
Int HalStead_Calculate (int n1, int n2)
{//n1 is the number of the operands; n2 is the number of the operators
Int n;
If (n1! =0&&n2! =0) // the value of the Halstead Complexity measure
n=n1log2n1+n2log2n2;
Return n; // return the finally result
}
```