USING A COMPETITIVE LEARNING NEURAL NETWORK TO EVALUATE SOFTWARE COMPLEXITY

John W. Sheppard
William R. Simpson

ARINC Research Corporation

ABSTRACT

With recent advances in neural networks, an increasing number of application areas are being explored for this technology. Also, as software takes a more prominent role in systems engineering, ensuring the quality of software is becoming a critical issue. This paper explores the application of one neural network paradigm—the competitive learning network—to the problem of evaluating software complexity. The network was developed by ARINC Research Corporation for its SofTest software analysis system, developed on a Sun workstation. In this paper, we discuss the network used in SofTest and the approach taken to train the network. We conclude with a discussion of the implications of the approach and areas for further research.

INTRODUCTION

> The degree to which characteristics that impede software maintenance are present is called software maintainability and is driven primarily by software complexity, the measure of how difficult the program is to comprehend and work with [1].

In 1987, ARINC Research Corporation began researching approaches for analyzing software to facilitate efficient testing and maintenance of code. Much of that work focused on the problem of modeling software in order to automatically generate software test strategies. This work included exploring the possibility of automatically building a functional model from source code. During two and a half years of research, ARINC Research developed two approaches to modeling and assessed a third approach [2-5]. With the knowledge gained, we developed a prototype tool—SofTest—to generate a functional model from FORTRAN source code [6].

In addition to the FORTRAN prototype, which exists on an HP-1000 minicomputer, we redesigned SofTest on a Sun workstation. This refined and redesigned version was intended to separate the subsystems that relied on specific programming languages from the analysis portion of the system. The result was a language-independent analysis tool with a language-dependent front-end. As an extension of our work, another language was used as the object of our analysis—the C programming language. As a result, the new version of SofTest analyzes C source code and generates several reports describing the structure and complexity of the software analyzed.

We apply SofTest when attempting to assess the overall maintainability of a software system. In performing such an analysis, SofTest looks for structural characteristics to determine the complexity of the code analyzed. As a result of the complexity analysis, several descriptive reports can be generated by SofTest, thus documenting the condition of the code.

MEASURES OF SOFTWARE COMPLEXITY

Lines of Code

The most basic measure of software complexity, lines of code (LOC), provides a measure of the size of a program. SofTest, using LOC as an indicator of complexity, determines the number of source LOC by program unit (or function). This measure includes executable, nonexecutable, and comment lines.

McCabe's Cyclomatic Number

LOC, by itself, is not a good measure of complexity because the number depends largely on the programmer's style. Some programmers freely use space and place individual commands on one or more lines. Other programmers prefer compact code and include several commands on one line. Thus, other measures are needed to assess the complexity of code.

Thomas J. McCabe [7, 8] developed an approach to examining the control structure of a function using the cyclomatic number of a graph. The cyclomatic number is based on a graph representation of the control flow of a function. The function is represented with nodes corresponding to commands in the code and edges corresponding to transfer of control. In graph theory, the cyclomatic number is defined to be equal to the "maximum number of linearly independent circuits" [7]. This assumes that the graph is strongly connected (i.e., there is a path from any node to any other node in the graph) and directed (i.e., a direction of flow is specified on the edges of the graph). The cyclomatic number may then be computed as follows:

$$V(G) = e - n + p$$

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

where e = the number of edges, n = the number of nodes, p = the number of connected components (which, for a software function, is always 1). In order to apply this expression to software, a control flow graph of the function analyzed is constructed. This graph is constructed with single point of entry and single point of exit. But the resulting graph is not strongly connected, so an edge is added from the exit to the entry. Because an edge has been added to the graph, McCabe specifies that the expression for $V(G)$ needs to be modified to the following (see reference 8 for an explanation for this modification):

$$V(G) = e - n + 2p$$

Using a simpler process, $V(G)$ can be determined without generating the control graph. The result is two versions of the cyclomatic number. $V(G)_1$ is computed by counting all of the decision points in the function (e.g., IF, FOR, WHILE, CASE) and adding 1. $V(G)_2$ is computed by counting all of the individual predicates in the function used in a decision point (e.g., EQUAL, LESS THAN) and adding 1. SofTest only computes $V(G)_1$.

The cyclomatic number is used to assist in path testing a software function. In path testing (path analysis), test data are generated that provide coverage of a software function. In other words, test data should cause every statement in the code to be executed at least once. The cyclomatic number provides a measure of the amount of testing required to "cover" a function. The measure, as our definition of cyclomatic number indicates, provides the number of independent circuits in the code. By removing the edge between the exit and entry, we see this corresponds to the number of independent paths from entry to exit in the function. SofTest imposes the standard specified by McCabe [8].

Halstead's Software Science

$V(G)$ also is not a sufficient measure, in itself, of software complexity. For example, the measure fails to consider the effects of nesting, and the CASE statements are treated with complexity equal to that of several IF statements. In 1977, Maurice Halstead discussed an approach to analyzing software using primitive characteristics in an attempt to eliminate these types of concerns [9]. The intent of Halstead's Software Science is to assess the complexity of software as independently of the programming language as possible. The SofTest program computes 12 metrics, including the following Halstead measures:

$n1$	= the number of unique operators		
$n2$	= the number of unique operands		
$N1$	= the total number of operators		
$N2$	= the total number of operands		
n	= $n1 + n2$	=	the vocabulary of the function
N	= $N1 + N2$	=	the length of the function
V	= $N(\log_2 n)$	=	the volume of the function
$\hat{L}$	= $(2/n1)(n2/N2)$	=	an estimate of the program level of the function
E	= $V/\hat{L}$	=	the effort of the function

The last three measures merit some additional comment. Volume, V, is a measure of the size of the function. Halstead based this measure on an estimate of the number of bits required to store the function in memory. The program-level estimate, $\hat{L}$, is a measure of the level of complexity of the function. Its intent is to provide a sense of how difficult it is to "understand" the source code for that function. Finally, effort, E, is based on both volume and level and is intended to provide a measure of the effort that was required to write the function.

CONTROL DENSITY

In addition to LOC, $V(G)$, and the Halstead metrics, SofTest computes a measure that combines lines of code and cyclomatic number, which we call the control density, D, of a function. The control density is computed as the ratio of cyclomatic number to lines of code. The interpretation of this measure is currently uncertain,* but the neural network (described below) found the measure to be significant in assessing software complexity.

ANALYSIS METHODS

In addition to computing the complexity metrics, several analyses are performed which more directly describe the structures of the code and recommend where one should focus efforts to improve the code. The following sections discuss the complexity analysis process of the SofTest system.

Metric Analysis

SofTest operates on a "reduced" version of the software being analyzed. This reduced version is a collection of symbol tables with each symbol in the software categorized by type. The metric analysis that SofTest performs proceeds from these symbol tables to compute the metrics discussed above. The measures are then compared with the standards of software engineering practice.

Pattern Analysis

In addition to the traditional metric analysis, SofTest incorporates a new approach using a neural network paradigm. The neural network is presented with complexity of data for each function analyzed, and the network detects and extracts patterns within the data. The network within SofTest was initially trained with data from more than 4,000 C functions from approximately 20 programs. The data consisted of normalized values of the 12 metrics: LOC, $V(G)$, $n1$, $n2$, $N1$, $N2$, n, N, V, $\hat{L}$, E, and D. Once the system was trained, we discovered that the network identified three categories of functions which, after examining the training data, we labeled as Standard, Marginal, and Nonstandard.

COMPETITIVE LEARNING

Background

No standards are currently accepted for Halstead's measures, but Halstead's measures have been accepted by industry as useful relative measures. Therefore, we sought to combine Halstead's measures with LOC and $V(G)$ and to identify

*It was expected that a high control density would indicate high complexity. According to the neural network, the opposite was true. Complex functions either had high LOC and low $V(G)$, or they had low LOC and high $V(G)$.

patterns in the data using a competitive learning neural network. The competitive learning paradigm was developed to identify features (or patterns) in a set of input data without explicitly training the network to identify these features. Thus, competitive learning is said to use a form of "unsupervised" learning (i.e., the network learns without direct supervision by an external agent) [10, 11].

The object of competitive learning is for the network to develop a set of "feature detectors." When data containing a learned feature are submitted to the network, then the activity of the network identifies which feature is present. To identify features, nodes within the network "compete" among themselves to respond to the stimulus pattern. The node that "wins" the competition has a feature associated with it. Consequently, when that node becomes active, the feature has been identified.

Two excellent applications for competitive learning networks are factor analysis and classification. In factor analysis, the network tends to respond according to factors (or common elements) within the data. An examination of the weights associated with the various nodes assists in identifying what these factors are. The classification problem, on the other hand, is more interested in the nodes that win the competition. Here the network attempts to classify the training data in such a way that, when similar or related data are presented, the network will correctly classify the data. The SofTest application uses the classification aspect of competitive learning networks.

Figure 1 shows an example of a competitive learning network. These networks usually consist of two layers of nodes with some restrictions on how the nodes communicate. These networks are referred to as "feedforward" networks, which means that data enter at one level and are transferred to the second level for processing. No data are passed back to the input layer. One layer of the network (labeled layer 0) is used strictly for input. This layer is fully connected to the second layer and simply passes data to that layer. A second layer (labeled layer 1) is used strictly for output.

When referring to the connections between the nodes of the network, note that the connections between layers are strictly excitatory, and the weights, corresponding to the connection strengths, are between 0 and 1. In addition, all weights associated with connections to a single output node must sum to 1. These weights are examined when performing a factor analysis to determine what factors exist in the data.

Although there are no connections between nodes at layer 0, intralayer connections do exist at layer 1. In fact, layer 1 nodes are completely interconnected, and these connections define the competitive aspect of the network. The difference in these connections is that they are inhibitory. The inhibitory nature of these intralayer connections is determined by a winner-take-all algorithm [11]. Further, the connections are implied in the competition algorithm, so they do not appear in a weight matrix (i.e., inhibitory weights are not learned). In the competition process, the node that wins the competition inhibits all activity on the other nodes at layer 1. (See reference 11 for a detailed discussion of competitive learning.)

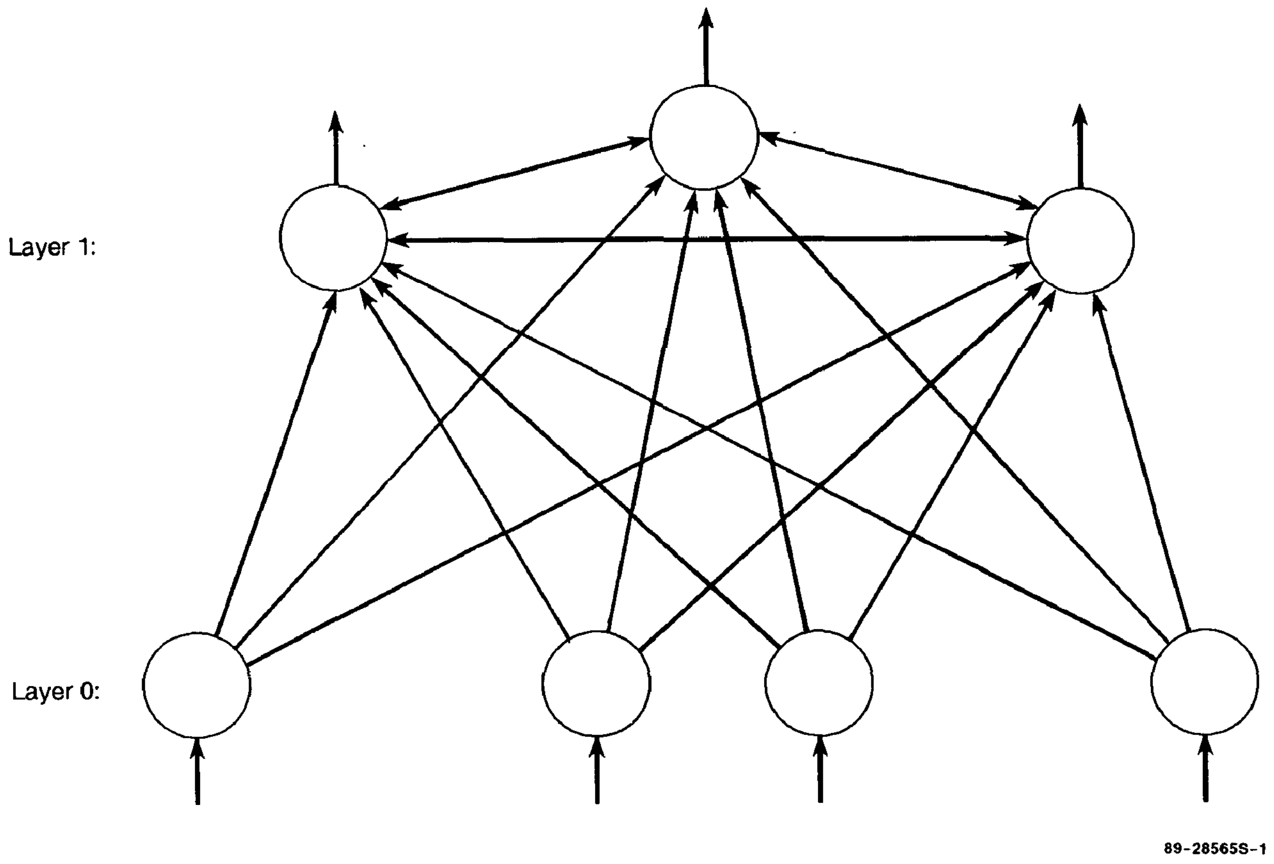

Figure 1. Competitive Learning Network

The SofTest Application

The network used in SofTest is a two-layer unsupervised competitive learning network that uses the software metrics listed earlier as the input to the network. In training the classification network for SofTest, it was hoped that certain characteristics of software would be identified to indicate code blocks that were overly complex or poorly structured. More than 4,000 functions written in C from approximately 20 programs were analyzed by SofTest to generate the 12 measures discussed above. Each program analyzed was written by varying numbers of programmers (from two to five per program).

The 12 measures were normalized with fixed maximum values by measure to the range of [0, 1]. They were then submitted to a network with 12 input nodes and 12 output nodes. Following training, three output nodes were detected having activity. A network was then constructed with 12 input nodes and 3 output nodes. The results were the same. The network generated is shown in Figure 2.

Following an examination of the source code, the three nodes at level 1 were found to indicate three levels of concern. (Note: The three levels of concern were identified after an examination of the actual source code in light of the information provided by the network.) These three levels are:

- *Standard Code*—Code that was written according to accepted software engineering practices (95.1%)
- *Marginal Code*—Code that has some characteristics contrary to accepted software engineering practices (4.75%)
- *Nonstandard Code*—Code that blatantly violates accepted software engineering practices (0.15%)

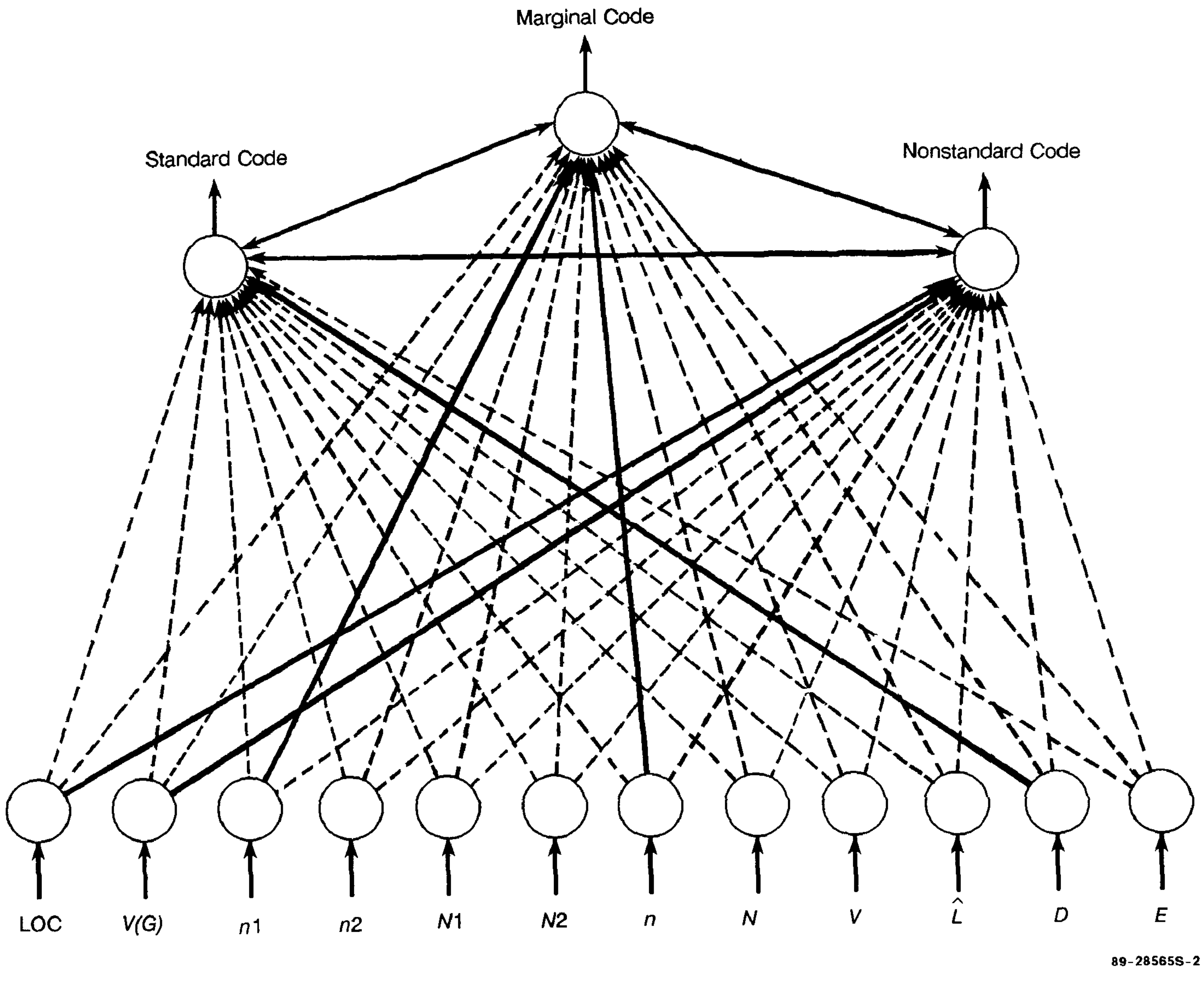

Figure 2. Neural Network for Classifying Software Complexity

Figure 2 also shows which metrics correspond to the three classifications. The connections shown by dashed lines had almost no impact on the competition. In fact, all 12 metrics contribute to each classification (an important consideration in factor analysis), but certain metrics can be identified as the major contributors to classifying the input. For our example, the metrics "owned" by the output nodes are as follows:

- ***Standard Code***—D: A high control density, when balanced by the other 11 metrics, appears to indicate well-structured code.
- ***Marginal Code***—$n1$ and n: The vocabulary is beginning to get excessive, especially in the use of language elements. This is balanced by the other 10 metrics, especially the effort, E, metric.
- ***Nonstandard Code***—LOC and $V(G)$: When the lines of code and control complexity become excessive, the code clearly has not been written within the bounds of standard software engineering practice. This class is balanced by the other 10 metrics with a particular emphasis on level, $\hat{L}$, and density, D.

Following the training of the neural network, several software systems were analyzed by the network to test the results. Part of a report generated by the network for a parser is shown in Figure 3. This report lists the complexity measures by function value and whether or not a standard has been violated or the network has identified a problem.

CONCLUSION

Neural networks are beginning to exhibit tremendous power and flexibility in interpreting large quantities of data. The competitive learning network is one example of a network paradigm that we applied to the software complexity analysis problem. The results of the research for this project show considerable promise for applying neural networks to the analysis of similar problems or expanding the capabilities of SofTest by incorporating other network paradigms.

Future work at ARINC Research in neural networks will include examining behavior patterns and symptoms of software anomalies. Using the ability of neural networks to generalize from a set a sample patterns, we hope to train the neural network to identify symptoms and to detect unusual patterns of activity on the basis of CPU utilization, memory utilization, or other performance measures.

In conclusion, the SofTest software analysis program was developed to identify portions of code that are potentially difficult to test and maintain. The neural network trained for SofTest provides an approach that extends beyond simply checking standards: it identifies functions with unusual or abnormal complexity characteristics. Such identification is expected to simplify the development and testing process for our major software efforts.

REFERENCES

1. Harrison, Warren, et al., "Applying Software Complexity Metrics to Program Maintenance," *Computer*, September 1982, 65–78.

2. Korel, Bogdan, "The Program Dependence Graph in Static Program Testing," *Information Processing Letters*, New York: Elsevier Science Publishers, January 30, 1987, 103–108.

3. Natour, I. A., "On the Control Dependence in the Program Dependence Graph," *Proceedings of CSC–88*, Atlanta, Georgia: Association for Computing Machinery, February 1988, 510–516.

4. Sheppard, John W., and William R. Simpson, "Functional Path Analysis: An Approach to Software Verification," *Proceedings of CSC–88*, Atlanta, Georgia: Association for Computing Machinery, February 1988, 266–272.

5. Bond, Rodney, and John W. Sheppard, "Structural Analysis Methods to Aid in Software Testing, Debugging, and Maintenance," Software Engineering Technical Note 2402, ARINC Research Corporation, June 1989.

6. Sheppard, J. W., and W. R. Simpson, "SofTest User's Manual HP-1000/A-900 System Version 1.0 Prototype," Software Engineering Technical Note 2210, ARINC Research Corporation, June 1989.

7. McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, no. 4, December 1976, 3–15.

8. McCabe, Thomas J., "Software Complexity Measurement," *Second Software Life Cycle Management Workshop*, Atlanta, Georgia, August 1978, 174–179.

9. Halstead, Maurice H., *Elements of Software Science*, New York: Elsevier North Holland, 1977.

10. Grossberg, Stephen, "Competitive Learning: From Interactive Activation to Adaptive Resonance," *Cognitive Science*, 11: 1987, 23–63.

11. Rumelhart, D. E., and J. L. Zipser, "Feature Discovery by Competitive Learning," *Parallel Distributed Processing*, D. E. Rumelhart and J. L. Zipser eds., Cambridge: The MIT Press, 1986.

```
There are 23 functions in this system.
                                                                                           D            E
Function              LOC    V(G)    n1     n2      N1      N2      n       N         V    L_hat  (V(G):LOC)   (Effort)
------------------------------------------------------------------------------------------------------------------------
yylex                 374    126     23    229    1368     251    252    1619  12915.22    0.079    0.3369  162865.2656
     *** Lines of code excessive!
     *** Control complexity is excessive!
yywrap                  4      1      5      1       6       1      6       7     18.09    0.400    0.2500      45.2367
comment                12      3     14      6      31      13     20      44    190.16    0.066    0.2500    2885.6570
count                  14      4     18      9      33      21     27      54    256.76    0.048    0.2857    5394.1992
check_type             17      1      6      1       7       1      7       8     22.46    0.333    0.0588      67.3831
yylook                148     32     42     52     433     248     94     681   4463.67    0.010    0.2162  446367.5000
     *** Control complexity is excessive!
yyback                 11      4     11      4      27      11     15      38    148.46    0.066    0.3636    2246.0183
yyinput                 3      1      6      0       8       0      6       8     20.68    0.048    0.3333     427.2665
yyoutput                4      1      6      1       8       3      7      11     30.88    0.111    0.2500     277.9559
yyunput                 4      1      6      1       8       3      7      11     30.88    0.111    0.2500     277.9559
main                  271     68     52    109     673     423    161    1096   8034.69    0.010    0.2509  811584.3750
     *** Lines of code excessive!
     *** Control complexity is excessive!
setup_global            8      2     11      6      18      12     17      30    122.62    0.091    0.2500    1348.9979
setup                   8      2     11      6      18      12     17      30    122.62    0.091    0.2500    1348.9979
initialize             90      1      9     87     427     255     96     682   4490.94    0.076    0.0111   59247.2852
     --- The pattern analysis has indicated that this function warrants some attention.
update_token           50     10     26     14     149     114     40     263   1399.67    0.009    0.2000  148900.7656
add_token              13      2     17     10      39      31     27      70    332.84    0.038    0.1538    8759.0029
update_ftoken          50     10     26     14     149     114     40     263   1399.67    0.009    0.2000  148900.7656
add_ftoken             13      2     17     10      39      31     27      70    332.84    0.038    0.1538    8759.0029
print_table            79     20     23     53     184     112     76     296   1849.39    0.041    0.2532   44997.2383
     *** Control complexity is excessive!
output_table           23      5     16     20      60      54     36     114    589.37    0.046    0.2174   12729.4062
store_label            12      3     16      6      34      20     22      54    240.81    0.038    0.2500    6421.5811
pass_two               93     13     31     46     234     151     77     385   2412.71    0.020    0.1398  122472.7344
     --- Control complexity is marginally high.
find_token             11      3     16      6      30      17     22      47    209.59    0.044    0.2727    4752.6826
------------------------------------------------------------------------------------------------------------------------
Minimum:                3      1      5      0       6       0      6       7     18.09    0.009    0.0111      45.2367
Maximum:              374    126     52    229    1368     423    252    1619  12915.22    0.400    0.3636  811584.3750
Mean:               57.04  13.70  17.74  30.04  173.17   82.52  47.78  255.70   1723.26    0.079    0.2260   87003.3516
Standard Deviation: 90.99  28.04  11.57  51.09  305.73  110.28  57.39  398.81   3072.59    0.094    0.0801  183639.6562
------------------------------------------------------------------------------------------------------------------------
Number of functions with marginally high LOC: 0
Number of functions with extremely high LOC: 2
Number of functions with marginally high cyclomatic number: 1
Number of functions with extremely high cyclomatic number: 4
Number of functions with marginal concern from pattern analysis: 1
Number of functions with high concern from pattern analysis: 0
Number of functions requiring attention: 6
```

Figure 3. Complexity Report Example