Finding Most Feasible Path in Weighted Control Flow Graph of a C Program for Testing Optimization

Author

[1]Vikas Agrahari, [2]Shubha Jain

[1](Research Scholar/M.Tech(CSE)/K.I.T Kanpur, Uttar Pradesh (India))
[2](Associate professor/ Dept of CSE/K.I.T Kanpur, Uttar Pradesh (India))

Abstract

An effective testing can reduce the cost and time considerably. In this study our aim is to design a software tool that will compute the most feasible path for all programs in C language. Finding of most feasible path requires a construction of weighted control flow graph, which helps in determining the frequency of all paths in control flow graph and finally most frequent path(as the weight decides the frequency) is termed as most feasible path. Concept of most feasible path makes the test data generation easy and in optimized manner and also act as an effective tool for efficient testing of entire software.

Key Words : *CFG, Cyclomatic Complexity, Edge Weight, Path Frequency*

1. Introduction

The problem is related to software testing and this paper helps in designing the test cases optimally by finding the most feasible path in weighted control flow graph of a C program.
In first module, we generate the simple control flow graph by identifying all the tokens of a program and then calculating the closed region in control flow graph which is useful for calculating cyclomatic complexity of code. Then after we find out the independent paths in our program using adjacency matrix.
In second module, we generate the weighted control flow graph by assigning the weight on to the edges of simple control flow graph by using different methods. Using this weighted control flow graph we generate the most feasible path from source node to destination node. So our project does following four tasks based upon the input program:

- Drawing of Control flow graph of the given input program.
- Determining Cyclomatic Complexity for finding independent paths of program using graph matrix of Control flow graph.
- Drawing of Weighted Control Flow graph by using three different methods.
- Determining most feasible path using Weighted Control Flow graph.

These two modules play an important role in the project. The project uses the advance features of C library for generating window and printing date of the System. While working through task that creates, develop & display this C compliant application, we have used many different library functions, graphics and console applications.

2. Specification

This software tool will take as its input, a program or code in C language, draws CFG, calculate Cyclomatic Complexity and no. of independent paths then at last find out most feasible path of

the program on the basis of frequency of nodes in the CFG. The data structure used here is a weighted control flow graph and weights are decided on the basis of three different techniques.

2.1 Design

There are three different methods for finding the frequency of all paths in Control Flow Graph and they are- number of statements, time of execution and memory used methods for determining the frequency of paths and finally for finding most feasible path (the one with highest frequency). The design of this software tool can be specified by the following steps:

- Draw the control flow graph for the given application.
- Show different paths covered in control flow graph.
- Determine the cyclomatic complexity of the software and find the number of
- Independent paths by using graph matrix.
- Determine frequencies of all paths(using any of the above three methods)
- Draw weighted control flow graph
- Find most feasible path out of them.

3. Software Testing Fundamentals

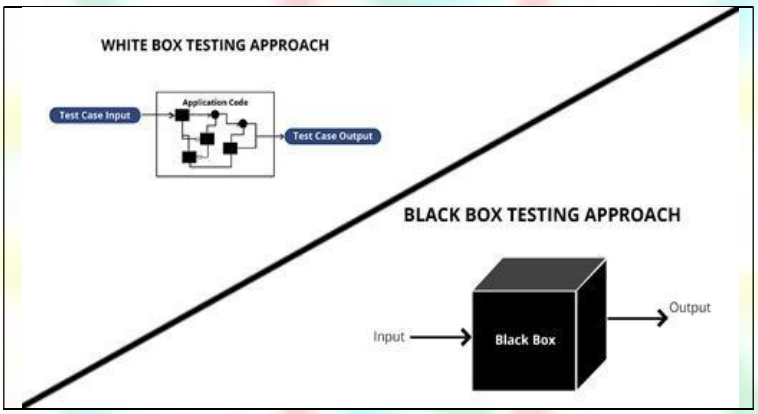

Fig 1: Software Testing Fundamental

3.1. The Nature of Software Defects

Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. General processing tends to be well understood while special case processing tends to be prone to errors. We often see that a logical path is not likely to be executed when it may be executed on a regular basis. Our unconscious assumptions about control flow and data lead to design errors that van only be detected by path testing. Typing errors cannot be predicted.

3.2. Basis Path Testing

This method enables the designer to derive a logical complexity measure of a procedural design and use it as a guide for defining a basis set of execution paths. It is a white box technique which analyzes the CFG. Test cases that exercise the basis set are guaranteed to execute every statement in the program at least once during testing.

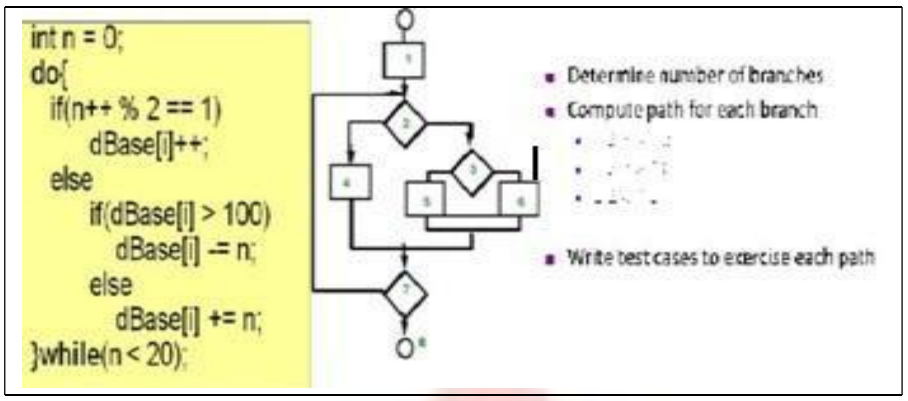

Fig 2: Basis Path Testing

3.3. Code Complexity Metrics

Most of the time, these jobs are non-trivial due to the complexity of most code. At least four items can make things complex: the obvious problem of understanding what is written, what the code is supposed to do, both at the macro and micro level the environment in which the code is to run, and the assumptions made about each of these things.

A lot of tools are available to help sort out each of these issues. Few programs out there, however, try to measure the complexity of the code[1]. We define complexity of code as the amount of effort needed to understand and modify the code correctly. As we explain in this article, computing complexity metrics often is a highly personal task. Also, few metrics have been shown to be of real value in determining the amount of effort needed to test the code.

Performance metrics measure, how well a valid program executes. Profiling tools fall into this category, and many tools are available. But for maintenance metrics, there are surprising few tools. Therefore, this column concerns creating maintenance metric tool that measures complexity. It can be used as a prototype for general tools in other languages.

3.3.1. Software Complexity Measurement

Software complexity is one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures. There are hundreds of software complexity measures [2], ranging from the simple, such as source lines of code, to the esoteric, such as the number of variable definition/usage associations.

An important criterion for metrics selection is uniformity of application, also known as “open reengineering.” The reason “open systems” are so popular for commercial software applications is that the user is guaranteed a certain level of interoperability-the applications work together in a common framework, and applications can be ported across hardware platforms with minimal impact. The open reengineering concept is similar in that the abstract models used to represent software systems should be as independent as possible of implementation characteristics such as source code formatting and programming language. The objective is to be able to set complexity standards and interpret the resultant numbers uniformly across projects and languages.

A particular complexity value should mean the same thing whether it was calculated from source code written in Ada, C, FORTRAN, or some other language. The most basic complexity measure, the number of lines of code, does not meet the open reengineering criterion, since it is extremely sensitive to programming language, coding style, and textual formatting of the source

code. The cyclomatic complexity measure, which measures the amount of decision logic in a source code function, meets the open reengineering criterion. It is completely independent of text formatting and is nearly independent of programming language since the same fundamental decision structures are available and uniformly used in all procedural programming languages.

3.3.2. Relationship Between Complexity and Testing

There is a strong connection between complexity and testing, and the structured testing methodology makes this connection explicit.

First of all, complexity is a common source of errors in software. This is true in both an abstract and a concrete sense. In the abstract sense, we can say that a highly complex software defeats the human mind's ability to perform accurate symbolic manipulations, and it results in errors. The same psychological factors that limit people's ability to do mental manipulations of more than the infamous "7 +/- 2" objects simultaneously apply to software. Structured programming techniques can push this barrier further away, but not eliminate it entirely. In the concrete sense, numerous studies and general industry experience have shown that the cyclomatic complexity measure correlates with errors in software modules. Other factors being equal, the more complex a module is, more likely to have errors. Also, beyond a certain threshold of complexity, the likelihood that a module contains errors increases sharply. Given this information, many organizations limit the cyclomatic complexity of their software modules in an attempt to increase overall reliability.

Second, complexity can be used to allocate testing effort by leveraging the connection between complexity and error to concentrate testing effort on the most error-prone software. In the structured testing methodology, this allocation is precise- i.e. the number of test paths required for each software module is exactly the cyclomatic complexity. So the complexity gives the clear understanding about the testing efforts. Other common white box testing criteria have the inherent anomaly that they can be satisfied with a small number of tests for arbitrarily complex (by any reasonable sense of "complexity") software. Complexity Reduction although the amount of decision logic in a program is to some extent determined by the intended functionality, software is often unnecessarily complex, especially at the level of individual modules. Unnecessary complexity is an impediment to effective testing for three major reasons. First, the extra logic must be tested, which requires extra tests. Second, tests that exercise the unnecessary logic may not appear distinct from other tests in terms of the software's functionality, which requires extra effort to perform each test. Finally, it may be expended to identify the dependencies and show that the criterion is satisfied to the greatest possible extent.

Unnecessary complexity also complicates maintenance, since the extra logic is misleading unless it’s unnecessary nature is clearly identified. Even worse, unnecessary complexity may indicate that the original developer did not understand the software, which is symptomatic of both maintenance difficulty and outright errors. This section quantifies unnecessary complexity, and discusses techniques for removing and testing it.

4. Weighted Control Flow Graph

There are different ways for depending upon characteristics of edges of Control Flow Graph in order to assign frequencies to nodes. Here we are discussing various methods for assigning the frequencies and out of all these methods we choose the most optimistic one to built weighted Control Flow Graph.

The methods can be defined as follows:

- No. of statements
- Execution time
- Memory used

4.1. Number of Statements

In this method frequencies are assigned to the edges of CFG on the basis of number of statements traversed from one node to another. Different control structure may contain same no. of statements or different no. of statements, within a same program if same control structure is defined more than once their frequency distribution may be same or different depending upon the no. of statements they contain in their definition. Thus within the same program edges of Control Flow Graph varies in their frequency distribution on the basis of no. of statements traversed by the pair of nodes which make the edge. In a recursive code, if similar edges are traversed more than once then its frequency is dependent only upon the no. of statements traversed through it. We recognize each statement by the symbol of semicolon in the statement since in C language each statement is terminated by semicolon.

4.1.1. Example

We are using a program in C language to show the frequency distribution.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int
x[20],y=9,i,*u;
for(i=0;i<y;i++)
{
scanf("%d",&x[i]);
printf("%dth element is:",x[i]);
*u=x[i];
u++;
}
if(x[0]!=0)
{
printf("Given array is not empty");
}
if(x[9]!=0)
{
printf("Given array is full");
}
else
{
printf("Given array is not full");
}
}
else
{
printf("Given array is empty");
}
getch();
}
```

Frequency assignment using no. of statements each edge traverse

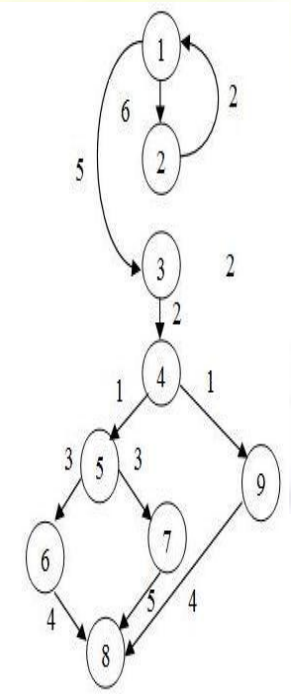

Fig 3: Weighted CFG for No. of Statements

4.2. Execution Time

In this method frequencies are assigned on the basis of time taken by the different statements present in the code i.e. in terms of graph it is the time taken by the edges to execute the statements covered by the connecting nodes. Using the concept of variation in time for executing different type of statements like simple print statement, initialization, and increment/decrement, computational, logical and other type of statements. Since the execution time is calculated in micro second so in this type of method the weight of CFG is assign in micro second. Depending upon compilation of program, some control statements take less time for execution and also preferred over others, some statements are executed recursively and some statements need to be executed before the execution of other group of statements. Therefore execution time of statements comprising the node varies dynamically and acts as a source of frequency distribution of nodes in Control Flow Graph.
Frequency assignment using execution time (in micro second) of edges is shown below -

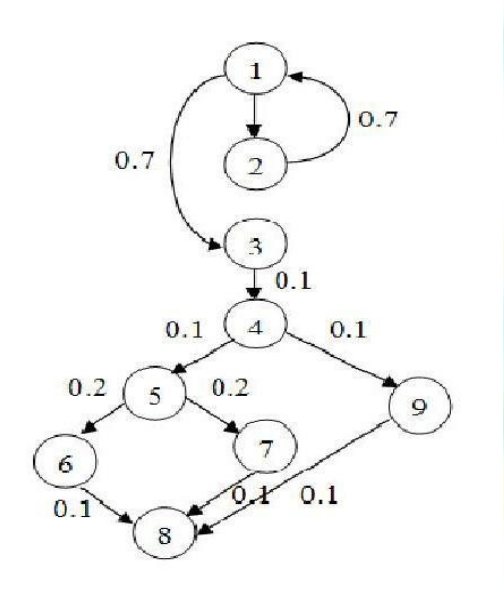

Fig 4: Weighted CFG for Execution Time

4.3. Memory Used

In this method frequencies are assigned on the basis of memory used by group of statement covered by traversal of nodes. This method always assigned the frequencies in a dynamic manner to the edges depending upon the memory used by the statements of different type of control structures like if-else, for, while and switch. Memory utilization of statements of control structures such as if else, for, while, do while is totally depends upon their definitions or we can say that the operation they are going to perform on the basis of requirement of program under execution. We assign the memory to the statements on the basis of bytes used by the statements such as integer uses 2 bytes; floating point uses 4 bytes and so on. So in this method the weights are assigned in the form of byte to the nodes. It is a complex method due to dispatching between memory and processor for calculating the memory unit.

Some edges are traversed many no. of times in a recursive manner where as other are executed only once, thus memory requirement of edges forming the Control Flow Graph varies and act as a source of frequency distribution.

Frequency assignment using memory utilization of edges is shown below –

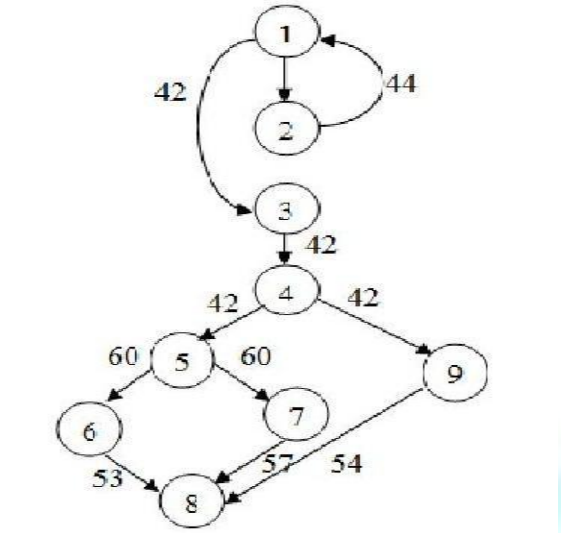

Fig 5: Weighted CFG for Execution Time

From the methods defined above for the frequency distribution, we find that the second method which is "number of statements" is easy to implement and shows more variation among the frequency of nodes in control flow graph.

5. Finding Most Feasible Path

After drawing weighted control flow graph by above mentioned methods next we have to find most feasible path in weighted control flow graph. For this purpose we have to compare the weights provided by the various alternatives (methods to calculate) to reach from one node to another. After comparing various alternatives finally we select the one path by which we have to make minimum effort. Like in no. of statement weight assigning method we select the path where we have to execute minimum statements, in execution time weight assigning method we have to select the path of minimum execution time and in memory used weight assignment method we have to select the path of minimum memory requirement.
Finding feasible path through above mentioned weight assignment method (Using Memory and Execution time method).

5.1. Example: Finding Feasible Path using Execution Time Weight Assignment Method

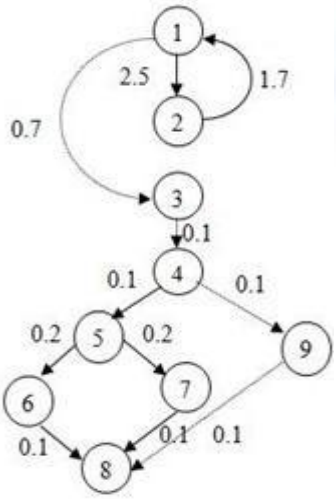

Fig 6: Feasible Path Using Execution time

5.2. Example: Finding Feasible Path using Memory Requirement Method

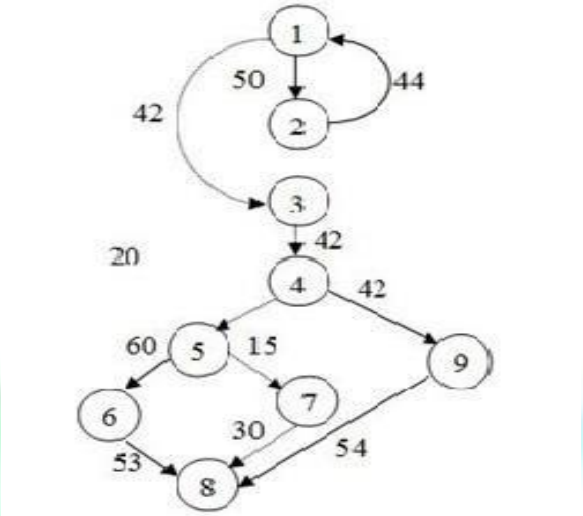

Fig 7: Feasible Path Using Memory Requirement

6. Conclusion

In this paper we are taking a C program as input and producing a weighted control flow graph for further finding the most feasible path as an output. We are producing output by considering the program's execution on the basis of time of execution and basis of memory consumption of each edge as described by weighted control flow graph.

7. Future Scope

As we are taking a C program as input and producing a weighted control flow graph as output and then finding out the most feasible path, in the same way the future researchers can create the same application for different programming languages like java, .net and c++ etc.

References

[1]. H. Zuse, "Software Complexity: Measures and Methods" De Gruyter, Berlin,(1991)
[2]. I. Chowdhury, "Using Complexity, Coupling, and Cohesion Metrics as Early Indicators of Vulnerabilities"
[3]. Ayman Madi, Oussama Kassem Zein And Seifedine Kadry: On the Improvement of Cuclomatic Complexity Mitric
[4]. IEEE transactions on software engineering, volume 19, March 1992.
[5]. Pressman r "Software engineering", Tata McGraw Hills.
[6]. Mall R, "Fundamentals of software Engineering', Prentice Hall.
[7]. Edvardsson, J., "A Survey on Automatic Test Data Generation," in Proceedings of the Second Conference on Computer Science and Engineering, Linkoping pp. 21-29, October 1999.
[8]. McMinn, P., "Search-based Software Test Data Generation: A Survey". Software Testing, Verification and Reliability, 14(2), pp. 105-156, June 2004.
[9]. McMinn, P., M. Holcombe, "Evolutionary TestingUsinganExtendedChaining Approach", Evolutionary Computation, 14(1), pp. 41-64,
[10]. Rumbaugh & Loresson W, "Object Oriented Modeling & Design", PHI 1991.