ISBN 978-981-11-3671-9
Proceedings of 2017 the 7th International Workshop on Computer Science and Engineering (WCSE 2017)
Beijing, 25-27 June, 2017, pp. 176-180
doi: 10.18178/wcse.2017.06.031

Number of Test Cases Required in Achieving Statement, Branch and Path Coverage using 'gcov': An Analysis

Ram Chandra Bhushan and Dharmendra Kumar Yadav[+]

Department of Computer Science & Engineering, Motilal Nehru National Institute of Technology Allahabad, Allahabad, U.P.-211004, India

Abstract. In software development, code coverage is a mechanism used to know the degree to which a program is executed when a particular test suite executes on it. The facts regarding code coverage as a forecaster of software quality may be contradictory and inconclusive. However, an estimation of the software testing practices practiced by the majority of the software professionals can help researchers to know how code coverage is being used. It is required to know for software development professionals that whether all the statements, branches and paths of source code are getting covered or not. Manual testing is difficult for most of the applications that is why testing process has been automated. But, in case of automated software testing, what should be the number of test cases required to achieve maximum coverage is not known. Most of the time even after executing several test cases, code coverage is not maximized. As per the earlier research, code coverage data provides us an important insight for our test cases and test suit's effectiveness and answers about parts of our source code that are executed? The primary objective of the paper is to present analytical results statistically for several codes of C and C++. Finally, it has been concluded that what should be the maximum limit for number of test cases required in achieving maximum statement, branch and path coverage.

Keywords: Code Coverage, Statement, Branch, Path, Cyclomatic Complexity.

1. Introduction

Code coverage is an evaluation technique used in software testing [2] [4]. It evaluates the degree of extent to which the source code of a program has been tested. Code coverage analysis is the process of:

- Finding portions/areas/modules of a program being not covered by a test suite or set of test cases.
- Generating required additional test cases to increase code coverage and
- Formation of a quantitative measure [1] of code coverage which is an indirect way of measuring quality.

One of the main objectives of the code coverage analysis is to identify redundant test cases that provide same results in terms of coverage or which covers the same portion of code. Code coverage is also a measure of extent of testing [4] performed on a particular test suite.

1.1. Objective and Scope

The work aims at statistical analysis performed through many 'C' and 'C++' programs. In this paper an application [5] has been shown and discussed but in background more than 100 applications has been analyzed. An alternate approach has been also taken to find out the cyclomatic complexity of all programs and then a correlation has been established in between the two approaches. The main focus of the work is to identify the minimum number of test cases required to cover the maximum statements, branches as well as paths in the code. The scope of this paper is limited to only 'C' and 'C++' codes as we have not analyzed the codes written in different languages but the same analysis can be used for other language codes as well.

[+]Corresponding author. Tel.: +91-9820839705(M), +91-532-227-1361(O); fax: +91-532-2545341.
E-mail address:dky@mnnit.ac.in.

2. Basic Coverage Criteria

Most of the developers often target 100% coverage though it is an admirable goal, but wrong type of coverage can lead to misleading predictions. A typical software development effort finds coverage in terms of the number of either statements or branches to be tested. Even with 100% statement or branch coverage, semantic bugs can still be present in the code, leaving developers with a false positive.

2.1. Calculation of Statement, Branch and Path Coverage: An Example

A small piece of 'C' source code:

```
int main() {
intp,q;
if((p+q)>100)
        printf("Large");
if(p>50)
        printf("PisLarge");        }
```

The flow chart for the above code is displayed in Fig. 1. To calculate statement coverage, one finds out the shortest number of paths following which all the nodes will be covered. For example in fig.1 by traversing through path 1A-2C-3D-E-4G-5H all nodes are covered. So, by traversing through only one path all the nodes 1,2,3,4 and 5 are covered, hence the statement coverage in this case is 1.

Branch Coverage (BC):

To calculate branch coverage, we need to find out the minimum number of paths which ensures covering of all the edges. As in fig.1, there is no single path which ensures coverage of all the edges in one chain. By following paths 1A-2C-3D-E-4G-5H edges A, C, D, E, G and H are getting covered except B and F. For complete coverage of these we can follow 1A-2B-E-4F. By these two paths it has been ensured that it traverses through all the edges. Hence, branch coverage is 2.

Path Coverage (PC):

Path coverage ensures covering of all paths from beginning to end. All such possible paths are-1A-2B-E-4F, 1A-2B-E-4G-5H, 1A-2C-3D-E-4G-5H and 1A-2C-3D-E-4F.So path coverage is 4.100% path coverage results in 100% statement coverage. 100% branch/decision coverage results in 100% statement coverage. 100% path coverage results 100% branch/decision coverage. Branch coverage and decision coverage are same.

3. Statistical Analysis and Working with 'gcov'

This analysis has been done by the tool "gcov" [3][9][10] which is an open source distribution in Linux environment. The analysis has been performed for more than 100 programs but due to limitation of space only one application has been presented. Further source code of the same 'C' program has been shown, thereafter the code has been compiled and executed in Linux environment with several test cases until the coverage reaches to the maximum. The compilation and execution of a program coded in 'C' and 'C++' is illustrated below. The name of the source code is "Test.c" which is a C Program.

```
For compilation:    [ram@localhost Project]$gcc -fprofile-arcs -ftest-coverage Test.c
For execution:  [ram@localhost Project]$ ./a.out
```

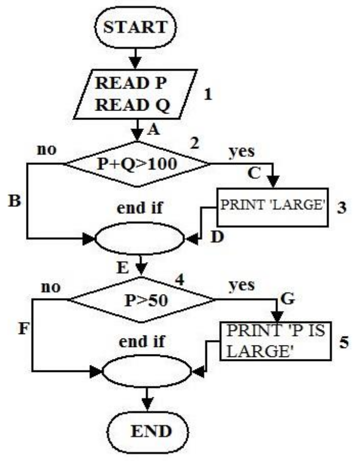
Then test cases are given as per the requirement of the program and finally we can see the coverage details by giving the command gcov Test.c –b which creates a Test.c.gcov file. These commands are not the part of the paper instead only results obtained after execution have been presented and analyzed.

3.1. Analysis of Code Coverage through a ‘C’ Application

Analysis with Recursive balanced Quick Sort

```
void qsort(inta[],int,int);
void partition(int a[],int,int,int *);
void print(int *,int);
void swap(int *,int *);
void main() {
int a[30],i,n; flushall();
printf(" enter how many data u want :"); scanf("%d",&n);
printf(" enter the data :"); for(i=0;i<n;i++)
{ printf(" %d .",i);
scanf("%d",&a[i]); } qsort(a,0,n-1);
printf(" SORTED DATA:");
print(a,n-1);}
void qsort(int a[],intlb,intub){
int j; if(ub>lb){
partition(a,lb,ub,&j); qsort(a,lb,j-2);
qsort(a,j+2,ub);} }
void partition(int a[],intlb,intub,int *j){
int mid=(lb+ub)/2,temp,up, down,pivot; pivot=a[mid];
up=ub;down=lb;
while(down<up){
while( a[down] <= pivot && down <= ub )
{ down++;
if(a[down]<a[down-1]&&down>lb)
swap(&a[down],&a[down-1]);}
while(a[up]>pivot){ up--;
if(a[up]>a[up+1]&&up<ub)
swap(&a[up],&a[up+1]);}
if(down<up){swap(&a[down],&a[up]);
if(a[down]<a[down-1]&&down>0)
swap(&a[down],&a[down-1]);
if(a[up]>a[up+1]&&up<ub)
swap(&a[up],&a[up+1]);} }
for(inti=0;i<ub-1;i++) if(a[i]>a[i+1])
swap(&a[i],&a[i+1]); *j=up;}void print(int a[],int n){
for(inti=0;i<=n;i++)
printf("%d.%d",i,a[i]); }
void swap(int *p,int *q){
intt;t=*p;          *p=*q;*q=t; }
```

After execution on several test cases different outputs got generated that has been shown in Fig. 1 and analysis of the program with all test cases together is being presented in Fig. 2.



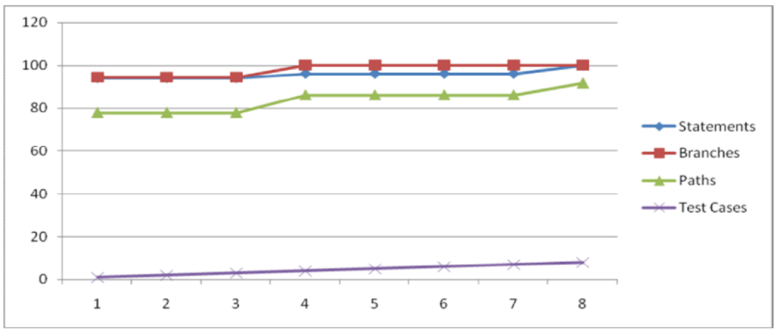


Fig. 1: Flow Chart of above C source code

Fig. 2: All test cases together for BQS

4. Another Approach: Achieving Path Coverage with the Help of Cyclomatic Complexity

The cyclomatic complexity [1] [5] of a program belongs to the difficulty in understanding the source code that is the count of the number of linearly independent paths present in the source code. It can be also obtained from control flow graph [6]. Thus, if the source code contains no decision points such as IF statements or FOR loops, the complexity will be 1, because there will be only one single path through the code. If the code contains a single IF statement having one condition then there will be two paths through the code depending on the two Boolean values of the condition.

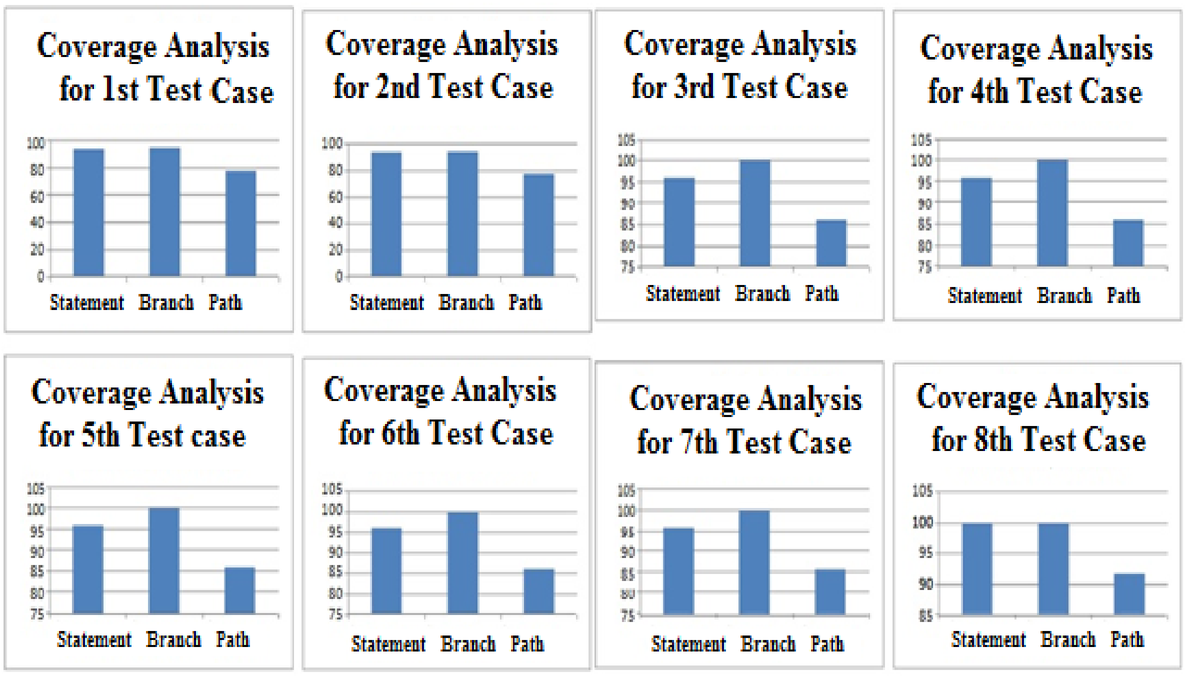


Fig. 3: Graphical representation of coverage for BQS

To achieve entire branch coverage let M is an upper bound for the number of test cases that are required which will also be a lower bound for the number of paths through control flow graph (CFG).Assuming every test case takes only one path, and then the number of test cases needed to achieve complete path coverage is equal to the number of paths in the graph that can actually be considered. But some paths might be impossible, so even if the number of paths through the CFG is an obvious upper bound for the number of test cases needed for path coverage but the number of possible paths may be sometimes less than M.

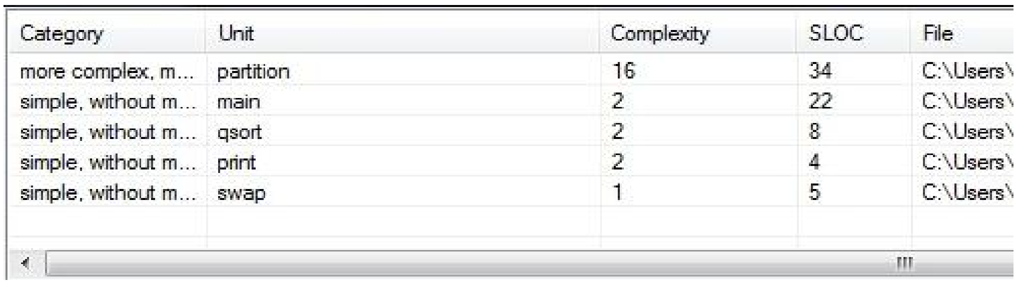
Branch coverage>= cyclomatic complexity [1] [5] >= number of paths.

It may be also concluded that number of test cases required achieving branch coverage <=cyclomatic complexity <= number of test cases required to achieve path coverage.

5. Correlation of Statistical Analysis and Cyclomatic Complexity

We found that the maximum number of distinct test cases to achieve path coverage may be equal or higher than the number of cyclomatic complexity. Similarly the maximum number of distinct test cases to achieve branch coverage may be equal or less than the number of cyclomatic complexity. We have measured the cyclomatic complexity of above application with the help of a tool named CCM which is a plug-in for Visual Studio 2008.

The snapshot has been shown in Fig. 4 and finally we will compare whether the number of test cases required for achieving maximum coverage are equal to the number of cyclomatic complexity or not.The table 1 provides a relationship for above application among cyclomatic complexity, number of test cases executed and the percentages of statement, branch and path coverage.



Category	Unit	Complexity	SLOC	File
more complex, m...	partition	16	34	C:\Users\
simple, without m...	main	2	22	C:\Users\
simple, without m...	qsort	2	8	C:\Users\
simple, without m...	print	2	4	C:\Users\
simple, without m...	swap	1	5	C:\Users\

Fig. 4: Snapshot of CCM

6. Conclusion and Future Work

From the table 1, it can be concluded that for any program the number of test cases required to achieve maximum coverage will not be more than the cyclomatic complexity of the program. It will be either equal to or less than the number of cyclomatic complexity. Although in this paper we have shown the detailed analysis for only one application butwe have applied this technique with number of applications. It is very important and useful to design the test case in such a way, so that it can achieve maximum coverage of all types rather than emphasizing over designing a tool for the coverage of the code. As, if the test cases are not suitable to execute the different linearly independent paths, no matter whatever the tool is, the coverage will not be maximal. It is the test suite that can execute the all linearly independent paths not the tool. A random test case generator can be designed in future which can produce the distinct test cases with the help of some optimization techniques that covers all the paths in the program in order to achieve the highest coverage.

Table 1: Cyclomatic Complexity Vs Coverage (%)

Application Name	CC	No. of Test Cases Executed	Statement Coverage (%)	Branch Coverage (%)	Path Coverage (%)
Balanced Quick Sort	16 +2 +2 +1	8	100	100	91.67

7. References

[1] Thomas J. McCabe, A Complexity Measure IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. SE-2, NO.4, DECEMBER 1976

[2] Steve Cornett ,A software testing technique. Copyright © Bullseye Testing Technology 1996-2011.

[3] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[4] Glenford J. Myers (2004). The Art of Software Testing, 2nd edition. Wiley. ISBN 0-471-46912-2

[5] Harrison (October 1984). "Applying Mccabe's complexity measure to multiple-exit programs". Software: Practice and Experience (J Wiley & Sons)

[6] Jiantao Pan, Carnegie M,Software Testing

[7] H. D. MiUls, "Mathematical foundations for structured programming,"Federal System Division, IBM Corp., Gaithersburg, MD,FSC 72-6012, 1972.

[8] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, Rex Black, Foundations of Software Testing. ISTQB Certification

[9] gcov—a Test Coverage Program. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[10] LCOV - the LTP GCOV extension. http://ltp.sourceforge.net/coverage/lcov.php