International Journal of Current Engineering and Technology
E-ISSN 2277 – 4106, P-ISSN 2347 - 5161
©2014 INPRESSCO®, All Rights Reserved
Available at http://inpressco.com/category/ijcet

General Article

# Contingent study of Black Box and White Box Testing Techniques

Neetu Dhingra[A] and Mayank[A*]

[A]Department of Information Technology, BLS Institute of Technology Management, Nh-10, Bahadurgarh, Haryana (India)

Accepted 20 Sept 2014, Available online 01 Oct 2014, **Vol.4, No.5 (Oct 2014)**

***Abstract***

*A high reliability and performance can be given to software by applying suitable testing techniques to them. This paper includes the comparison between black box and white box testing techniques in order make the software reliable and highly efficient. A contingent study is made over the testing techniques to get the solution to follow a better testing technique at better place. The paper includes the example to specify the techniques properly and to get the better results.*

***Keywords:*** *Black box, White box, Equivalence class, Cyclomatic Complexity*

## Introduction

Software testing is as old as the hills in the history of digital computers. The testing of software is an important means of assessing the software to determine its quality. Since testing typically consumes 40 - 50% of development efforts, and consumes more effort for systems that require higher levels of reliability, it is a significant part of the software engineering. Modern software systems must be extremely reliable and correct. Automatic methods for ensuring software correctness range from static techniques, such as (software) model checking or static analysis, to dynamic techniques, such as testing. All these techniques have strengths and weaknesses: model checking (with abstraction) is automatic, exhaustive, but may suffer from scalability issues. Static analysis, on the other hand, scales to very large programs but may give too many spurious warnings, while testing alone may miss important errors, since it is inherently incomplete (Trivedi, 2012). We can define software testing as a process or a series of processes, design to make sure that computer code does what it was actually design to do and it doesn't do anything unintended.(Khan, 2011). Software testing can be defining that as a process of executing a program with the intent of finding errors. So, testing means that one inspects behaviour of a program on a finite set of test cases (a set of inputs, execution prerequisites, and presumed outcomes developed for a particular target, like as to employ a particular program path or to verify compliance with a specific requirement, for which valued inputs always prevail. In operation, the entire set of test cases is examined as infinite, therefore conceptually there are too many test cases even for the uncomplicated programs. In this case, testing could demand months and months to execute. So, how will we be able to select the most proper set of test cases? In practice, various approaches are used for that, and some of them are corresponded with risk analysis, while others with test engineering proficiency. Testing is an action implement for assessing software quality and for improving it. Hence, the goal of testing is systematic detection of different classes of errors (error can be defined as a human action that produces an incorrect result, in a minimum amount of time and with a minimum amount of effort. It is a process of accessing the functionality and correctness of a software by analysis. The major motive of testing can be assurance of quality, estimation of reliability, validation and verification. Software testing is an elemental constituent of software quality assurance and represents a review of specification, design and coding. The major target of software testing is to affirm the quality of software system by systematically testing the software in carefully controlled circumstances, another aim is to recognize the completeness and correctness of the software, and ultimately it reveals undiscovered errors. Testing is basically a task of locating errors. It may be:

1).Positive testing
2).Negative testing

### Positive testing

– Operate application is should be operated.
– Does it behave normally?
– Use proper variety of legal test data, including data values at the boundaries to test if it fails.
– Check out the actual test result with the expected.
– Are results correct?
– Does the application work correctly?

### Negative testing

– Test for abnormal operations.
– Does the system fail/crash?
– Testing for illegal or abnormal data.

*Corresponding author: **Mayank**

– Deliberately seeks to make things go wrong and to discover/ detect.
– Does an application do what it should not?
– Does it fail to do what it should?
–

**Principles of Testing**

To make software testing effective and efficient we follow certain principles:

1. Testing should be based on the user requirements.
2. Testing time and resources are limited.
3. Exhaustive testing is impossible.
4. Use effective resources to test.
5. Testing planning should be done early.
6. Testing should be begin in small and progress in large.
7. Testing should be conducted by a different testing team or external team.
8. All tests should be according to customer requirements.
9. Assign the best person for testing.
10. Test should be planned to show software defects and not their absence.
11. Prepare the test reports including test cases and test results to summarize the result of testing.
12. Advance test planning is must and should be updated timely.
13. Testing activities should start as early as possible in the software or system development life cycle and should be focused on defined objectives.
14. Defect clustering refers to a small number of modules contain most of the defects discovered during before executing testing, or are responsible for the most operational failures.

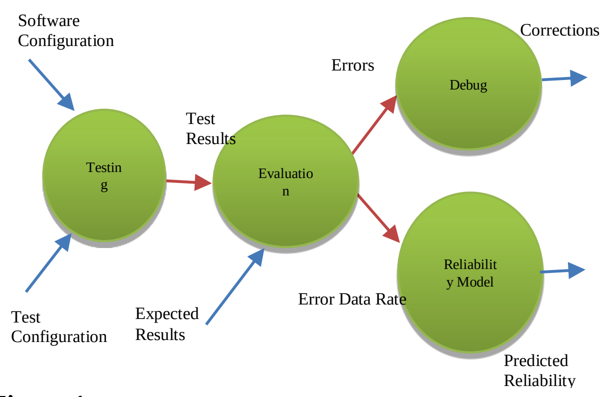


**Figure 1**

Types of Testing

Software testing is involved in each stage of software life cycle, but the way of testing conducted at each stage of software development is different in nature and it has different objectives (Dondeti, 2012).

A software testing Strategy should be flexible enough to promote a customized testing approach at same time it must be right enough. Strategy is generally developed by project managers, software engineer and testing specialist (Abhijit A. Sawant, 2012).

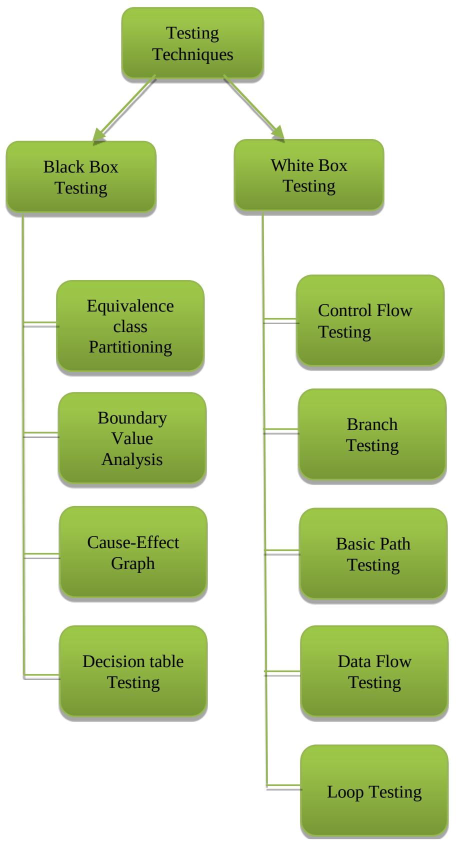


**Figure 2**

**Black Box Testing Technique**

Black Box Testing is used when code of the module is not available. In such situations appropriate priorities can be given to different test cases, so that the quality of software is not compromised, if testing is to be stopped prematurely(Harsh Bhasin, 2014).It is a technique of testing without having any internal working knowledge of the application. It only examines the basic aspects of the system and to verify that whether it is according to customer/client requirements or not.

a) **Equivalence class Partitioning:** It can reduce the number of test cases, as it split the input data of a software unit into partition of data from which test cases can be derived. This technique divides the input domain of a program onto equivalence classes. It is set of valid or invalid states for input conditions, and can be defined as:
1).An input condition specifies a range →one valid and two invalid equivalence classes are defined.
2). an input condition needs a describe value → one valid and two invalid
Equivalence classes are defined. An input condition describes a member of a set one valid and one invalid → equivalence class are defined;
3).An input condition is in the form of Boolean one→ valid and one invalid equivalence class are defined.

b) **Boundary Value Analysis:** It focuses more on testing at boundaries, or where the utmost boundary values are

select. It includes minimum, maximum, just inside/outside boundaries, error values and representative values.
In this technique one can form the test cases in the following way:

1. An input condition describes a range bounded by values a and b test cases should be made with values just above and just below a and b, respectively.
2. An input condition specifies various values → test cases should be produced to exercise the minimum and maximum numbers;
3. Rules 1 and 2 apply to output conditions only if internal program data structures have prescribed boundaries, produce test cases to exercise that data structure at its boundary.

**c). Decision table testing:** it is a testing technique, in which we take condition stub and action stub. It is a good way to deal with different combination of inputs with their associated output. Decision tables are very much helpful in test design technique. It helps the testing team to search the effects of combination of different inputs and other software that correctly implement business rules.

**Table 1**

| | |
|---|---|
| **Conditions Stub** | **Conditions Entry** |
| **Actions Stub** | **Actions Entry** |

**d).Cause-Effect Graph**: It is a testing technique, in which testing begins by creating a graph and establishing the relation between the effect and its causes. Identity, negation, logic OR. One uses this technique when one wants to translate a policy or procedure specified in a natural language into programming language. This technique means:

Input conditions and actions are listed for a function an identifier is allocated for each one of them cause-effect graph is created this graph is changed into a decision table the rules of this table are modified to test cases.

**White Box Testing Technique:** It is the detailed investigation of internal logic and structure of the code. In this testing it is necessary for a tester to have full knowledge of source code (Mohd. Ehmer Khan, 2012)

a) **Control Flow Testing**: Control flow graphs or program graphs that represent the control flow of programs are widely used in the analysis of software and have been studied for many year (GOLD, 2010) it is a structural testing strategy that uses the program control flow as a model control flow and favours more but simpler paths over fewer but complicated path. Control flow testing applies to almost all software and is effective for most software. Testing strategy that uses the program's control flow as a model control flow testing favour more but simpler paths over complicated but fewer paths. Now, we will define various coverage methods:
**Statement Coverage:** It is a measure of the percentage of statements that have been executed by test cases. Less than 100% statement coverage means that not all lines of code have been executed we can achieve statement coverage by identifying cyclomatic number and executing this minimum set of test cases. An advantage of statement coverage is that it is greatly able to isolate the portion of program, which could not be executed.

**Branch Coverage:** A stronger logic coverage criterion is known as branch coverage or decision coverage. It is measures of the percentage of the decision point of the program have been evaluated as both true and false in test cases. Examples of branch coverage-DO WHILE statements, IF statements and multiway GOTO statements. Branch coverage is usually shown to satisfy statement coverage. By 100% branch coverage we mean that every control flow graph is traversed.

**Condition Coverage:** A criterion which is stronger than decision coverage is condition coverage. It is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true and false outcome in test cases.

b) **Branch Testing** Branch testing has the objective to test every option (true or false) on every control statement which also includes compound decision.(when the second decision depends upon the first decision).In branch testing, test cases are designed to exercise control flow branches or decision points in a unit. All branches inside the branch are tested at least once.

c) **Basis Path Testing:** Basis path testing allows the test case designer to produce a logical complexity measure of procedural design and then uses this measure as an approach for outlining a basic set of execution paths. Basic path testing makes sure that each independent path through the code is taken in a predetermined order.

1).Flow Graph Notation
2).Cyclomatic Complexity
3).Deriving Test Cases
4).Graph Matrices

d) **Data Flow Testing:** Data flow testing is another type of white box testing which looks at how data moves inside a program. In data flow testing the control flow graph is annoted with the information about how the program variables are defined and used. We can also define data flow testing as testing techniques which is based on the observation that values associated with variables can effect program execution. Data flow testing picks enough paths to assure that:

1. Every data object has been initialized prior to its use.
2. All defined objects have been used at least once.
Some of the major points of data flow testing are:
1).All data flow anomalies are resolved.
2).Ignore the integration problems by doing all data flow operation on a variable within the same routine.
3).When possible use explicit (rather than implicit) declaration of data.

e) **Loop Testing:** Loop testing is another type of white box testing which exclusively focuses on the validity of

loop construct. Loops are easy to test unless dependencies exist between the loops or among the loop and the code it contain. There are four types of loops:
 Simple Loop
 Nested Loop
 Concatenated Loop
 Unstructured Loop

Working of black box and white box testing

## Example of Black Box Testing Techniques

**Table 2**

| Marks obtained | Grade |
|---|---|
| **80-100** | DISTINCTION |
| **60-79** | 1$^{ST}$ DIVISION |
| **50-59** | 2$^{ND}$ DIVISION |
| **40-49** | 3$^{RD}$ DIVISION |
| **0-39** | FAIL |

**C1: 0≤MATH≤100C2:0≤COMPUTER≤100**
**C3:0≤ENGLISH≤100**
**PER%=(MATH+COMPUTER+ENGLISH/300)*100**
**C4:80≤PER%≤100C5:60≤PER%≤79**
**C6:50≤PER%≤59C7:40≤PER%≤49**
**C8: 0≤PER%≤39**

### 1). Boundary value Analysis (BVA)

It yields (4n+1) test cases.
N=3 (no of variables)
(4*3+1)

**Table 3**

| Test Cases | Math | Computer | English | %age | Expected Output |
|---|---|---|---|---|---|
| **1** | 0 | 98 | 98 | 65.33333 | 1st div |
| **2** | 100 | 1 | 1 | 34 | fail |
| **3** | 99 | 1 | 1 | 33.66667 | fail |
| **4** | 100 | 5 | 5 | 36.66667 | fail |
| **5** | 98 | 2 | 2 | 34 | fail |
| **6** | 2 | 97 | 97 | 65.33333 | 1st div |
| **7** | 10 | 100 | 100 | 70 | 1st div |
| **8** | 100 | 0 | 0 | 33.33333 | fail |
| **9** | 99 | 2 | 1 | 34 | fail |
| **10** | 5 | 98 | 98 | 67 | 1st div |
| **11** | 1 | 100 | 100 | 67 | 1st div |
| **12** | 99 | 98 | 98 | 98.33333 | distinction |
| **13** | 100 | 0 | 0 | 33.33333 | fail |
| **14** | 97 | 99 | 99 | 98.33333 | distinction |
| **15** | 0 | 100 | 100 | 66.66667 | 1st div |

=13 TEST CASES

We now draw the table which shows those 13 test cases. Note: 8 and 13 test cases are redundant.

### 2). Equivalence Class Testing

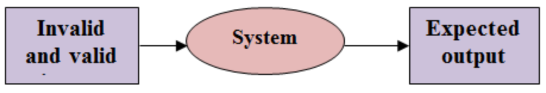


**Figure 3**

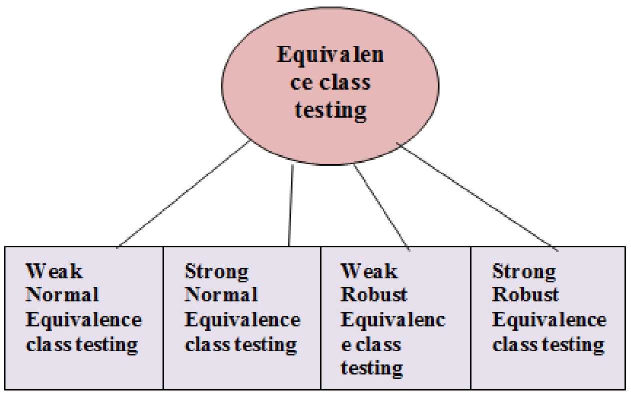

Equivalence class testing
Weak Normal Equivalence class testing
Strong Normal Equivalence class testing
Weak Robust Equivalence class testing
Strong Robust Equivalence class testing

**Figure 4**

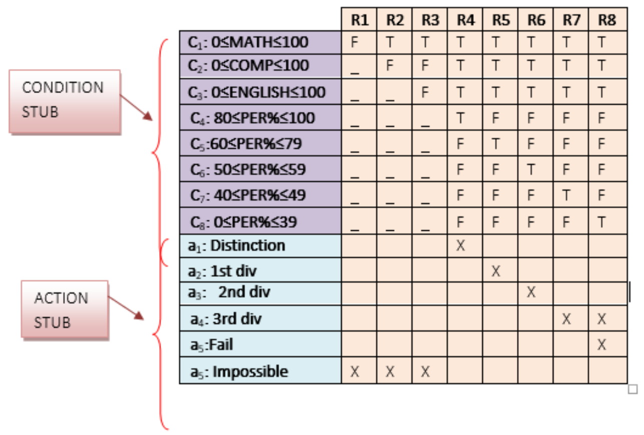
**Table 4**

| Weak normal equivalence class testing | | | | | |
|---|---|---|---|---|---|
| **Test Cases** | **Math** | **Computer** | **English** | **%age** | **Expected Output** |
| **1** | 32 | 33 | 35 | 33 | Fail |
| **2** | 68 | 65 | 62 | 65 | 1st Division |
| **3** | 70 | 75 | 72 | 71 | Distinction |
| **4** | 40 | 42 | 41 | 40 | 3rd Division |
| **5** | 55 | 55 | 56 | 55 | 2nd Division |

**Table 6**

| Strong robust equivalence class testing | | | | | |
|---|---|---|---|---|---|
| **Test Cases** | **Math** | **Computer** | **English** | **%age** | **Expected Output** |
| **1** | -1 | 45 | 48 | _ | Invalid input of Math |
| **2** | 45 | -1 | 48 | _ | Invalid input of Computer |
| **3** | 45 | 48 | -1 | _ | Invalid input of English |
| **4** | -1 | -8 | 65 | _ | Invalid input math & comp |
| **5** | -1 | -8 | -9 | _ | All invalid input |

### 3).Decision Table Testing


CONDITION STUB
ACTION STUB

| | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 |
|---|---|---|---|---|---|---|---|---|
| $C_1$: 0≤MATH≤100 | F | T | T | T | T | T | T | T |
| $C_2$: 0≤COMP≤100 | _ | F | F | T | T | T | T | T |
| $C_3$: 0≤ENGLISH≤100 | _ | _ | F | T | T | T | T | T |
| $C_4$: 80≤PER%≤100 | _ | _ | _ | T | F | F | F | F |
| $C_5$:60≤PER%≤79 | _ | _ | _ | F | T | F | F | F |
| $C_6$: 50≤PER%≤59 | _ | _ | _ | F | F | T | F | F |
| $C_7$: 40≤PER%≤49 | _ | _ | _ | F | F | F | T | F |
| $C_8$: 0≤PER%≤39 | _ | _ | _ | F | F | F | F | T |
| $a_1$: Distinction | | | | X | | | | |
| $a_2$: 1st div | | | | | X | | | |
| $a_3$: 2nd div | | | | | | X | | |
| $a_4$: 3rd div | | | | | | | X | X |
| $a_5$:Fail | | | | | | | | X |
| $a_5$: Impossible | X | X | X | | | | | |

**Figure 5**

### 4). Causes and Effects

**Step 1: The causes are**

| | |
|---|---|
| C1: 0≤MATH≤100 | C2:0≤COMPUTER≤100 |
| C3:0≤ENGLISH≤100 | C4:80≤PER%≤100 |
| C5:60≤PER%≤79 | C6:50≤PER%≤59 |
| C7:40≤PER%≤49 | C8: 0≤PER%≤39 |

**Step 2: The effects are**

a1: Distinction
a2: 1st div
a3:  2nd div
a4: 3rd div
a5: Fail
a6: Impossible

**Step 3: Cause-Effect graph is**

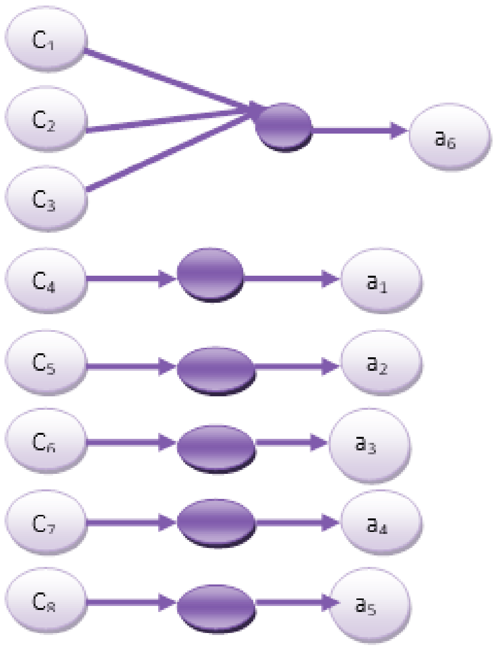


**Figure 6**

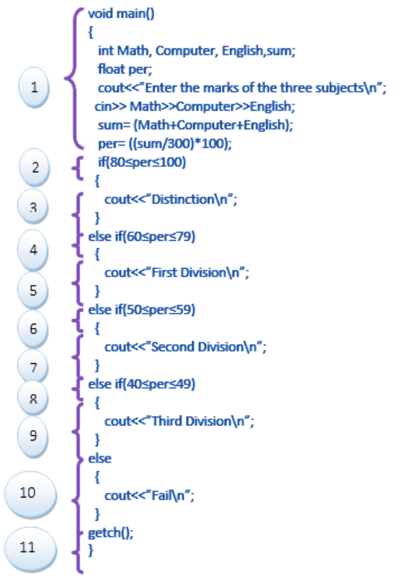
**White Box Testing Example**

**1). Cyclomatic Complexity**

Cyclomatic complexity is probably the most widely used complexity metric in Cyclomatic complexity is probably the most widely used complexity metric in software engineering. Defined by Thomas McCabe in 1976 and based on the control flow structure of a program. It is easy to understand, easy to calculate and it gives useful results. It's a measure of the structural complexity of a procedure. (Ayman Madi, 2013)

It is a Software matrix that provides a quantitative measure of the logical complexity of a program. It is computed in one of the three ways:

1).The Number of Regions belongs to the Cyclomatic Complexity.
2).Cylomatic complexity: E-N+2(E is the no. of edges and N s no. of Nodes)
3).Cylomatic complexity: P+1 (where P shows the no. of Predicate Nodes)



```
     void main()
     {
       int Math, Computer, English,sum;
       float per;
1      cout<<"Enter the marks of the three subjects\n";
      cin>> Math>>Computer>>English;
       sum= (Math+Computer+English);
       per= ((sum/300)*100);
2      if(80≤per≤100)
      {
3        cout<<"Distinction\n";
      }
     else if(60≤per≤79)
4     {
         cout<<"First Division\n";
5     }
     else if(50≤per≤59)
6     {
         cout<<"Second Division\n";
7     }
     else if(40≤per≤49)
8     {
         cout<<"Third Division\n";
9     }
     else
      {
10       cout<<"Fail\n";
      }
     getch();
11   }
```

**Figure 7**

**2). Basis Path Testing (Data Flow Graph)**

Basis path testing is one of the famous structural testingcriteria. It is a methodology which searches the program domain for suitable test data, such that after executing theprogram with the test data, a predefined path is reached (Yeresime Suresh, 2013)

Cyclomatic Complexity V(G) as follows:

V(G)= Enclosed Regions+1 =>  4+1 => 5
V(G)= E-N+2 =>  14-11+2 => 5
V(G)= P+1 =>  4+1 => 5   [ 2,4,6,8 are predicate nodes]

As V (G) =5 by all three methods. It mean it's a well written code, its testability is high and cost to maintain is low.

*Five independent paths are:*

Path 1: 1-2-3-11
Path 2: 1-2-4-5-11
Path 3: 1-2-4-6-7-11
Path 4: 1-2-4-6-8-9-11
Path 5: 1-2-4-6-8-10-11

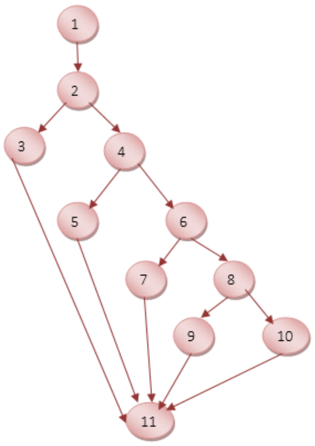
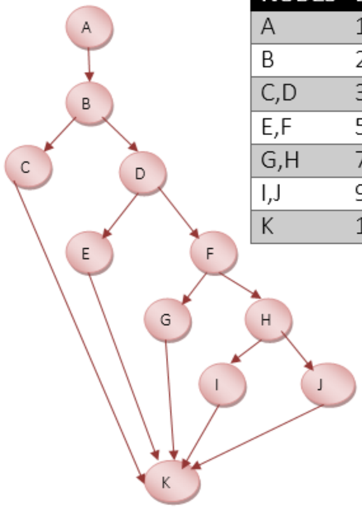


**Figure 8**

## DD Path Testing

From the flow graph, we can draw another graph that is known as decision to decision (DD) path graph, our main concentration now is on the decision nodes only.



| NODES | LINE |
|---|---|
| A | 1 |
| B | 2 |
| C,D | 3,4 |
| E,F | 5,6 |
| G,H | 7,8 |
| I,J | 9,10 |
| K | 11 |

**Figure 9**

*DD Paths:*

Path 1: A-B-C-K
Path 2: A-B-D-E-K
Path 3: A-B-D-F-G-K
Path 4: A-B-D-F-H-I-K
Path 5: A-B-D-F-H-J-K

## Graph Matrix

In this testing, we convert flow graph into a square matrix with one row and one column for every node in graph. The objective is to trace all links of the graph at least once.

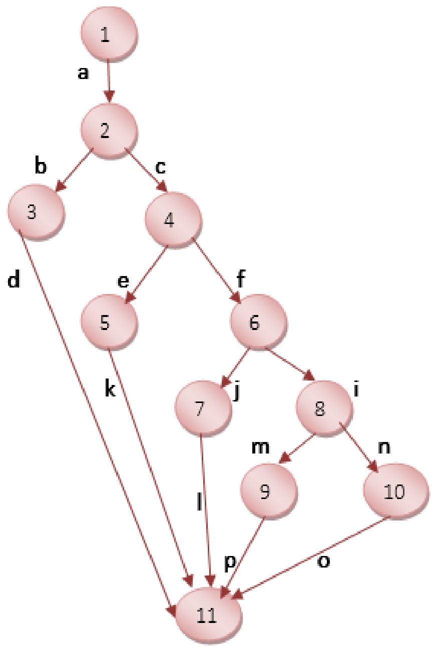
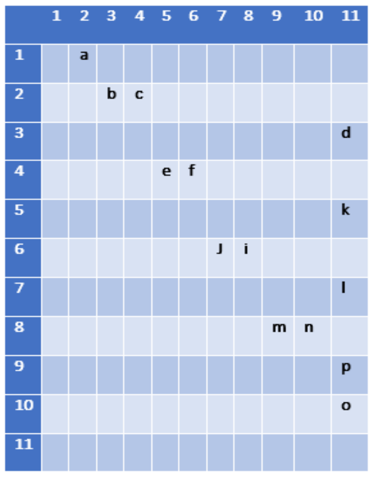


**Figure 10**



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | a | | | | | | | | | |
| 2 | | | b | c | | | | | | | |
| 3 | | | | | | | | | | | d |
| 4 | | | | | e | f | | | | | |
| 5 | | | | | | | | | | | k |
| 6 | | | | | | | J | i | | | |
| 7 | | | | | | | | | | | l |
| 8 | | | | | | | | | m | n | |
| 9 | | | | | | | | | | | p |
| 10 | | | | | | | | | | | o |
| 11 | | | | | | | | | | | |

**Figure 11**

## Comparison between black box and white box testing

**Table 7**

| SNO. | TESTING POINTS | BLACK BOX | WHITE BOX |
|---|---|---|---|
| 1 | Focus on functional requirements | YES | NO |
| 2 | Focus on procedural detail | NO | YES |
| 3 | Apply on later stages of testing | YES | NO |
| 4 | Apply on early stages of testing | NO | YES |
| 5 | Test the control structure of the program | NO | YES |
| 6 | Focus on to test the software interface | YES | NO |
| 7 | Test the logical path of the software | NO | YES |
| 8 | To check the internal data structure and storage | NO | YES |

## Conclusion

Software testing is often less formal and rigorous than it should, and a main reason for that is because we have struggled to define best practices, methodologies, principles, standards for optimal software testing. To perform testing effectively and efficiently, everyone involved with testing should be familiar with basic software testing goals, principles, limitations and concepts(Abhijit A. Sawant, 2012).As we compared to both, we find that black box testing is focus on functional requirements of the software and it includes the tests that are conducted at the software interface. While white box testing guarantee that all independent path within a module has been covered at least once and exercise internal data structure to ensure their validity. Hence, we conclude that white box is more efficient and reliable as it covers maximum error in the modules and provides an effective software product to customer as technology upgraded day by day.

## References

Abhijit A. Sawant, P. H. B. a. P. M. C., 2012. Software Testing Techniques and Strategies. International Journal of Engineering Research and Applications, 2(3), pp. 980-986.

Ayman Madi, O. K. Z. a. S. K., 2013. On the Improvement of Cyclomatic Complexity Metric. International Journal of Software Engineering and Its Applications, 7(2).

Dondeti, S. N. a. J., 2012. Black box and white box testing. International Journal of Embedded Systems and Applications, 2(2).

Gold, R., 2010. Control flowgraphs and code coverage. Int. J. Appl. Math. Comput. Sci., 20(4), pp. 739-749.

Harsh Bhasin, E. K., 2014. Black Box Testing based on Requirement Analysis and Design Specifications. International Journal of Computer Applications, 87(18), pp. 0975-8887.

Khan, M. E., 2011. Different Approaches to White Box Testing Technique for Finding Errors. International Journal of Software Engineering and Its Applications, 5(3), p. 14.

Mohd. Ehmer Khan, F. K., 2012. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. International Journal of Advanced Computer Science and Applications, 3(6).

Trivedi, S. H., 2012. Software Testing Techniques. International Journal of Advanced Research in Computer Science and Software Engineering, 2(10).

Yeresime Suresh, S. K. R., 2013. A Genetic Algorithm based Approach for Test Data Generationin Basis Path Testing. The International Journal of Soft Computing and Software Engineering, 3(3).