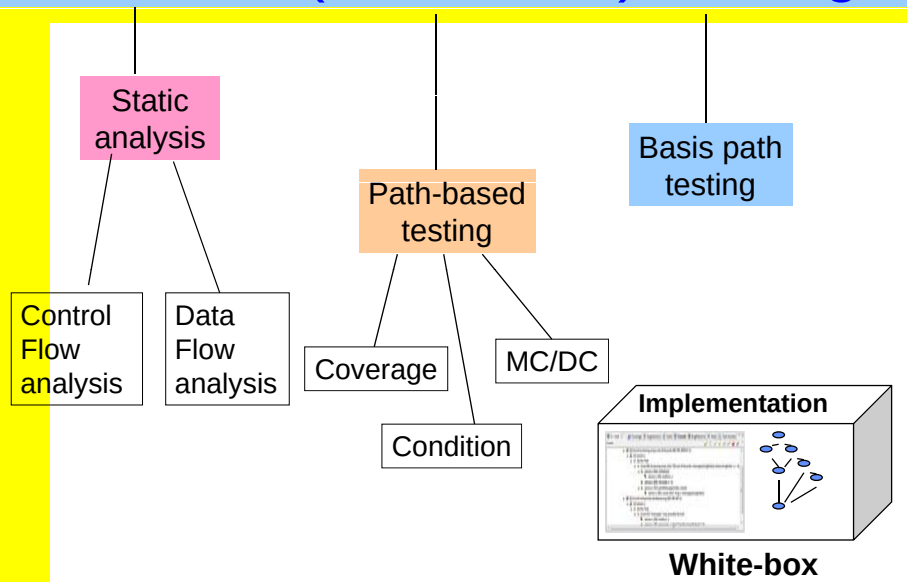
Course Structure

1. Software Quality Assurance
2. Testing Fundamentals
3. Code-based Techniques – Part I
4. Specification-based Techniques
5. Inspection Technique
6. Test Tools
7. Measuring Software Quality
8. TDD

Code-based (White Box) Testing



Learning Objectives

- Apply the following code-based testing techniques for **unit testing**:
  - Control flow analysis and testing
  - Dataflow analysis and testing
  - Path testing
  - Condition testing
  - Basis path testing
  - Domain testing
- Apply the following error-oriented testing techniques:
  - Error-based testing and error guessing
  - Mutation testing
  - Fault-based testing
- Use heuristics to test loops
- Understand difference between code-based & code coverage.

A True Story

▶ Introduction
Static Analysis
Path-based
Basis path

A man was crawling around on the sidewalk beneath a street light.

When asked by a policeman on what he was doing, he replied that he was looking for his car keys.

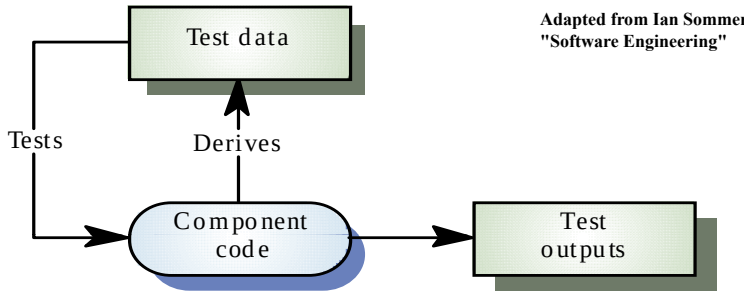
"Did you lose them here?" the policeman asked.

"No, I lost them in the parking lot, but the light is better here."

Moral of the story for tester?

White-box testing

- Introduction
- Static Analysis
- Path-based
- Basis path



Adapted from Ian Sommerville "Software Engineering"

Advantages

- force test team to reason carefully about the implementation (source code)
- reveal errors in "hidden" code (code added by the programmer and not required by the specification, e.g., input checking).

Disadvantage

Since the starting point for developing white-box test cases is the code itself, there is no way of finding requirements spec. not implemented in the code!!

Example: Program CountWord

- Introduction
- Static Analysis
- Path-based
- Basis path

Specification of Program CountWord:

Outputs the number of lines, the number of words, and the number of characters, given an input text of one or more lines.

Maximum number of lines is 20000.

1. What test cases would we use?
2. How many test cases?
3. How do we know we have used enough test cases?

Number of Test Cases for Testing CountWord?

- Introduction
- Static Analysis
- Path-based
- Basis path

Test case	1st Count
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
>10	

Example: Actual Program of CountWord

- Introduction
- Static Analysis
- Path-based
- Basis path


```
1   #define YES 1
2   #define NO 0
3   main()
4   {
5     int c, number_of_line, number_of_word, number_of_char, inword;
6     inword = NO;
7     number_of_line = 0;
8     number_of_word = 0;
9     number_of_char = 0;
10    c = getchar();
11      while (c != EOF) {
12          number_of_char = number_of_char + 1;
13          if (c == '\n')
14              number_of_line = number_of_line + 1;
15          if (c == ' ' || c == '\n' || c == '\t')
16              inword = NO;
17          else if (inword == NO) {
18              inword = YES;
19              number_of_word = number_of_word + 1;
20          }
21          c = getchar();
22      }
23      printf("%d \n", number_of_line);
24      printf("%d \n", number_of_word);
25      printf("%d \n", number_of_char);
26    }
```

Number of Test Cases for Testing CountWord?

Test case	1st Count	2nd count
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
>10		

Static Analysis for Fault Detection

Programs are analysed without being executed.

- Try to identify fault-prone code,
- Discover anomalous circumstances (e.g., unused variables), and
- Generate test data to cover specific elements (e.g., all statements, all branches) of the program's structure

- Static analysis may be performed **automatically** with tools such as parsers, data flow analysers, and syntax analysers, or **manually**, such as inspection and walkthrough.

Static Analysis

⌘ Static analysis can also be used:

- Understand the source code and answer:
  - From where in the source code a certain function is called?
  - Where a certain variable is defined and what is its type?
  - Which global data types exist?
- reveal bad coding practice in the source code (e.g., use A, B rather meaningful names)
- check the compliance with coding standards, and
- compute software metrics, which give numbers for:
  - Size of the program (e.g., 100KLOC)
  - Complexity of the program (e.g., V=15)
  - Maintainability (e.g., V=5)
  - Software quality (e.g., 2 defect/KLOC)

Control Flow Analysis

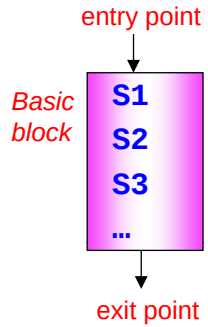
- A program's control flow relation shows program elements according to their execution order.
- A program element can be a **decision, a single statement,** or **a block of statements**.
- If element 2 can be executed immediately after element 1, then (1, 2) is in the control flow relation of the program. For example, if 1 refers to the decision **`(x<q)`** and 2 refers to the assignment statement in

```
1        if (x < q) then
2          p=a-q
```

then (1, 2) is in the control flow relation.

Drawing a Control Flow Graph

- A **basic block** is a sequence of consecutive statements with a single entry and a single exit point.
- Control always enters a basic block at its entry point and exits from its exit point.



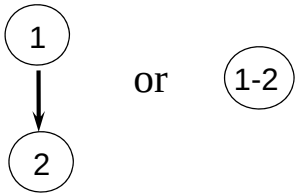
A **control flow graph (**or **flow graph**) G is defined as a finite set N of nodes and a finite set E of edges.

An edge (i, j) in E connects 2 nodes n_i and n_j in N. We write G=(N,E) to denote a flow graph G with nodes given by N and edges by E.

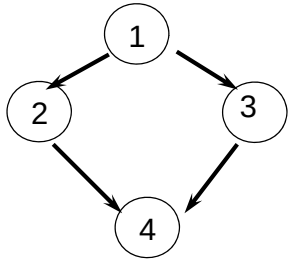
In a flow graph, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

Control Flow Graph (1)


```
1  A=A+1
2  B=B+1
```




```
1  IF (A>B) then
2       A=A+1
3  else A=A-1
4  B=B+1
```

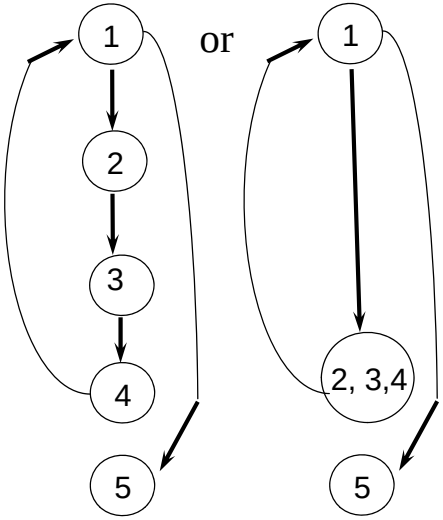


Note the statements, conditions, and paths

A **path** is a sequence of branches from the module's entry to the exit.

Control Flow Graph (2)

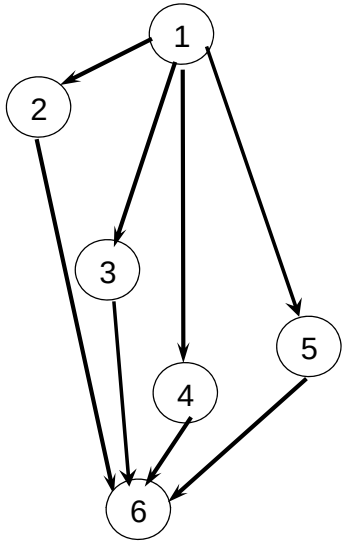

```
1  While (A>B) do
2     B=B+1
3     Write(A-B)
4  endWhile
5  Write(B)
```



*Example **Paths**:*
(1, 2, 3, 4, 1, 5)
(1, 2, 3, 4, 1, 2, 3, 4, 1, 5)

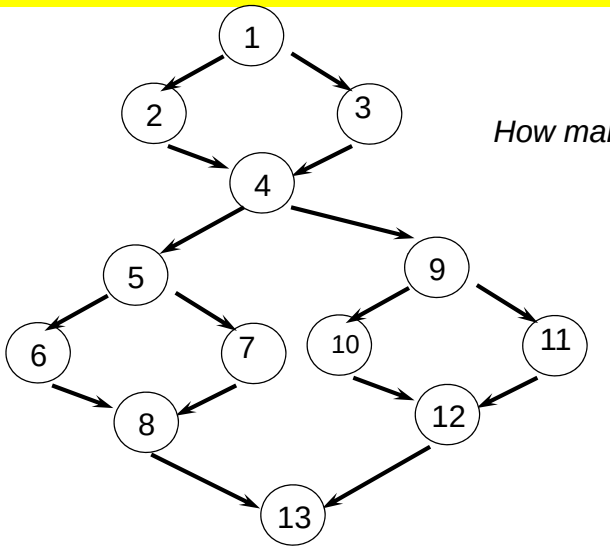
Control Flow Graph (3)


```
1  Switch (day) {
2       Monday, Tuesday, Wed, Thur :
3       Fri: println ("9-6"); break;
4       Sat: println ("9-noon"); break;
5       Sun: println ("home"); break;
   }
6  println (name);
```



Example

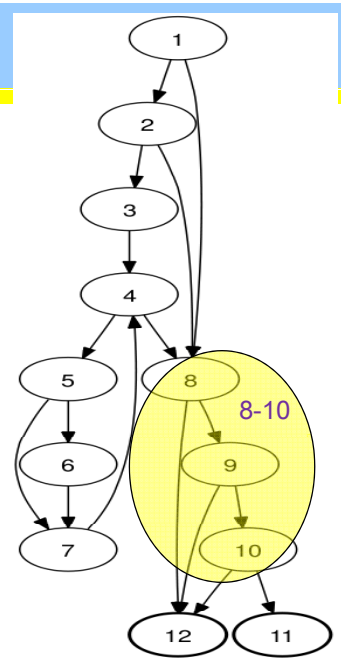
Introduction
Static Analysis
▸ Control flow
Data flow
Path-based
Basis path



How many paths?

Another Example

```
public boolean validateIdentifier(String s) {
        char achar;
/* 1 */     boolean validId = false;
/* 1 */     if (s.length() > 0) {
/* 2 */         achar = s.charAt(0);
/* 2 */         validId = validS(achar);
/* 2 */         if (s.length() > 1) {
/* 3 */             achar = s.charAt(1);
/* 3 */             int i = 1;
/* 4 */             while (i < s.length() − 1) {
/* 5 */                 achar = s.charAt(i);
/* 5 */                 if (! validF(achar))
/* 6 */                     validId = false ;
/* 7 */                 i++;
                    }
                }
                }
/* 8, 9, 10 */  if (validId && (s.length() >= 1) && (s.length() < 6))
/* 11 */            return true;
                else
/* 12 */            return false ;
}
```



Draw the Control Flow Graph of CountWord

Introduction
Static Analysis
▸ Control flow
Data flow
Path-based
Basis path

Data Flow Analysis

Introduction
Static Analysis
Control flow
▸ Data flow
Path-based
Basis path

Fundamentally, key actions of any software is move data from one location to another.

Example:

- assigning a new value to a variable (e.g., **A**=10)
- computing a new value and store in a particular variable (e.g., **A**=B*12).

We want to know: which definition defines the value used in a certain statement (e.g., A=**B***12)

We look for dataflow anomaly (any flow condition that may indicate problems).

Example dataflow anomalies:

- ➔ defining a variable twice with no intervening reference,
- ➔ referencing a variable that is undefined, and
- ➔ undefining a variable that has not been referenced since its last definition.

Data Flow Analysis

d	*define*	*assign a value to a variable;* ***e.g., x:= y+a*** ***x is define***
r	*ref*	***c-use*** *: reference a variable in a* ***c****omputation or output;* ***e.g., y=x +4*** ***x is c-use*** *or* ***p-use****: reference a variable in a* ***p****redicate;* ***e.g., if (x<a)…*** ***x is p-use***
u	*undefine*	*release or de-allocate memory for the variable (e.g., a loop control variable typically loses its value after loop exit.)*

The data flow relation shows program elements according to their data access behavior. If element 2 **uses (refers to)** a data object that was potentially **defined** at element 1, then (1, 2) is in the dataflow relation.

Example of c-use (computational use)

A c-use path starts from a definition of the variable and ends at a statement in which it is involved in **computing**.

```
sum = 0;
prod = 0;
i = 1;
While (i <= n)
        {
            sum = sum + i;
            prod *= i;
            i = i +1;
        }
if (k == 0) print_results;
if (k == 1) compute;
```

sum is referenced (c-use)

i is referenced (c-use)

Example of p–use (predicate use)

A p-use path starts from a definition of the variable and ends in a statement in which it appears inside a predicate.

```
sum = 0;
prod = 0;
i = 1;
While (i <= n)
        {
            sum = sum + i;
            prod *= i;
            i = i +1;
        }
if (k == 0) print_results;
if (k == 1) compute;
```

i is referenced (p-use)

For simplicity, we will name both c-use and p-use 'reference'

Example Data Flow Anomaly


```
Begin
   A=1
   B=A*A-2
   While A<100 Do
   Begin
      B=A*A+2
      A=A+B
   End
End
```

variable	A	B
	u	u
	d	
	r	d
	r	
	r	d
	r,d	r
	u	u

Data flow sequence for A and B:

A: udrrrrdru **OK**

B: uddru **abnormal**

Data Flow Sequences

Introduction
Static Analysis
Control flow
Data flow
Path-based
Basis path

dd: variable is defined twice without an intervening use; a possible **problem**

du: variable is defined and killed; probably a **fault**

dr: OK

ud: variable is undefined, then defined; OK

uu: variable is killed twice; why? Can be a fault

ur: try to reference a variable after it has been killed; clearly a **fault**

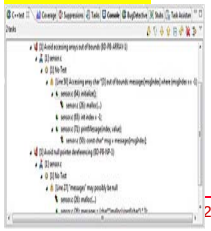
rr: variable is referenced twice; OK

Check for these anomalies: dd, du, ur

Static Analysis Tools

Introduction
Static Analysis
Path-based
Basis path

- Many static code analysis tools available (e.g., JustCode, Parasoft).
- They offer different depths of analysis, and some will only operate on a few programming languages.
- Most of them run on uncompiled source code and first translate to an intermediate language, which the analysis tool itself can read.
- They save time compared to doing static analysis by hand.



Many tools produce data that must be laboriously analyzed and processed; **staff requires skill and a lot of training**.

Path-based Testing: coverage

Introduction
Static Analysis
Path-based
Basis path

⌘ Make reference to the control flow graph, try to achieve certain **coverage**

$$\text{Coverage} = \frac{\text{Items executed at least once}}{\text{Total number of items}}$$

⌘ Coverage is a notion how completely testing has been done.

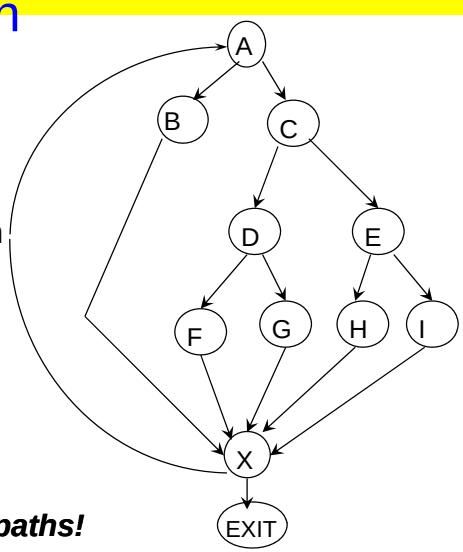
⌘ When we fail to achieve 100% coverage, we know that there are gaps in our testing.

⌘ **100% coverage is not "100% tested"!** Each coverage aspect is narrow; good coverage is necessary, but not sufficient to achieve good testing.

We have seen this Example Flow Graph

Introduction
Static Analysis
Path-based
Basis path

No more than 20 times



Too many paths!

Common Types of Coverage

Introduction
Static Analysis
Path-based
▸Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

1. **All statement coverage:** execute each statement at least once – give us confidence that every statement is capable of executing correctly.
2. **All branch (all decision) coverage:** execute each branch at least once
3. **All condition coverage:** execute every condition for **true** and **false** values.
4. **All path coverage:** execute each path at least once.

All path has higher defect-finding power than **all condition**, which in term has a higher defect-finding power than **all branch**.

Example

Introduction
Static Analysis
Path-based
▸Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

```
1  IF (A or B) then
2      Discount = 0.9
   end IF
3  Charge = Charge * Discount
```

1. How many test case needed to achieve 100% statement coverage?
2. How many test case needed to achieve 100% branch coverage?
3. Can you identify some weaknesses with branch testing?

Other Types of Coverage

Introduction
Static Analysis
Path-based
▸Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

- For the decision **(A or B)**, test case (A=T, B=F) and (FF) will cover the true and false of the decision outcome.
- But, the effect of B is not tested – these two test cases (TF, FF) cannot distinguish between the decision **(A or B)** and the decision **A**!

Consider the Decision

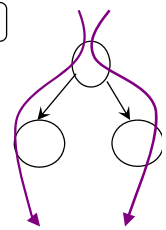
Introduction
Static Analysis
Path-based
▸Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

```
if ((age<25) and (sex=male) and (not married))
   then …..
```

decision

3 conditions => 8 combinations:

	(age<25)	(sex=male)	(not married)	decision
1	True	T	T	T
2	T	T	F	F
3	T	F	F	F
4	T	F	T	F
5	False	T	T	F
6	F	T	F	F
7	F	F	F	F
8	F	F	T	F



Decision Coverage ≠ condition coverage

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

A decision may consist of several conditions.

Trying a decision both ways (or executing both branches of the decision) does not imply every condition within the decision has been exercised for both FALSE and TRUE.

In the previous example, if we pick test cases 2 and 8, then every condition in the decision will have tried both True and False values. The condition coverage is then 100%.

Condition coverage requires that each condition in a decision take on all possible outcomes at least once, but does not require that the decision take on all possible outcomes at least once.

Condition/Decision Coverage

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

Use test cases that exercised the true and false cases of each individual condition within a decision point, **and** the true and false cases of the decision point.

Example

```
if ((a<10) and (b=-1)) then ….
```

- Test case 1: a=5, b=-1 exercise the true branch
- Test case 2: a=5, b=1 exercise the false branch

But condition (a<10) has not taken the false value.
Need one more test: a=11, b=any value
to achieve the condition/decision coverage.

But, Condition/decision coverage has limitation

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

For decision expression `((age<25) or (sex=male))`, test cases (TT) and (FF) would meet the condition/decision coverage.

But, these 2 tests does not distinguish the correct expression `((age<25) or (sex=male))` from

- `(age<25)`, or
- `(sex=male)`, or
- `((age<25) or (sex=male))`

We need a better coverage, called **MC/DC**

Example: for `((age<25) or (sex=male))`, test cases (TF), (FT), and (FF) achieve MC/DC.

Modified Condition/Decision Coverage (MC/DC)

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

Require

- Every point of entry and exit in the program has been invoked at least once
- Each decision takes on every possible outcome (true and false) at least once
- Every condition in a decision in the program has taken on all possible outcomes (true and false) at least once
- Each condition has been shown to affect that decision outcome *independency* (by varying just that decision while holding fixed all other possible conditions.)

Safety-critical software, like airborne software must comply with the DO-178B standard, certified by FAA, which requires the testing process to meet this code coverage criterion.

MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

- Condition/decision coverage does not guarantee the coverage of all conditions in the code because in many test cases, some conditions of a decision are masked by the other conditions.

- **Masking** refers to that specific inputs to a logic expression can hide the effect of other inputs to the expression.
- Examples
  - a false input to an **and** operator masks all other inputs, and
  - a true input to an **or** operator masks all other inputs.

MC/DC insures a much more complete coverage than decision coverage and condition/decision coverage.

MC/DC

- MC/DC coverage was designed for languages (Basic, FORTRAN) containing logical operators that do not short-circuit.
- For C, C++ and Java, MC/DC requires exactly the same test cases as condition/decision coverage.
- In general, a minimum of n+1 test case for a decision with n inputs.

Why MC/DC?

- People were concerned that compilers could and often generate extra code that added hidden functionality that might not be tested.
- Some examples:
  - array bounds checking,
  - exception handling,
  - error detection
  - libraries such as math libraries that would be linked in during the final build process.

How to identify test cases to achieve MC/DC?

Use the Unique-cause approach

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

1. Create a truth table for the logical expression (representing the decision).
2. The left-hand columns of the truth table list all possible input combinations for the decision.
3. The columns on the right indicate the possible **independence pairs** for each condition.
4. Test cases that provide MC/DC are selected by identifying pairs of rows where only one condition and the decision outcome change values between the two rows.
5. Any combination of the independence pairs (with a **minimum of one pair for each condition**) will yield the minimum tests for each logical expression.

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If (B and C) then ...

	B	C	r
1	T	T	T
2	T	F	F
3	F	T	F
4	F	F	F

	B	C	r	***B***	***C***
1	T	T	T		
2	T	F	F		
3	F	T	F		
4	F	F	F		

Steps:

1. Make a truth table.
2. Add additional 2 columns labeled with each input variable to record which rows will pair up with the existing row to show independence.
   In this case, we have 2 variables, so we have 4 rows in the truth table and 5 columns.

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If (B and C) then ...

	B	C	r	***B***	***C***
1	T	T	T	***3***	
2	T	F	F		
3	F	T	F	***1***	
4	F	F	F		

Steps:

3. For each variable, we find the rows where only the variable of interest changes and that change results in a change to the output (r).

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If (B and C) then ...

	B	C	r	***B***	***C***
1	T	T	T		***2***
2	T	F	F		***1***
3	F	T	F		
4	F	F	F		

Steps:

3. For each variable, we find the rows where only the variable of interest changes and that change results in a change to the output (r).

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If (B and C) then ...

	B	C	r	***B***	***C***
1	T	T	T	***3***	***2***
2	T	F	F		***1***
3	F	T	F	***1***	
4	F	F	F		

The final set of test cases achieving MC/DC: {1, 2, 3}

This set exercises all independence pairs for B and C!

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If ((A and D) or (C and E)) then ...

	A	D	C	E	q	***A***	***D***	***C***	***E***
1	T	T	T	T	T				
2	T	T	T	F	T	10	6		
3	T	T	F	T	T	11	7		
4	T	T	F	F	T	12	8		
5	T	F	T	T	T			7	6
6	T	F	T	F	F		2		5
7	T	F	F	T	F		3	5	
8	T	F	F	F	F		4		
9	F	T	T	T	T			11	10
10	F	T	T	F	F	2			9
11	F	T	F	T	F	3		9	
12	F	T	F	F	F	4			
13	F	F	T	T	T			15	14
14	F	F	T	F	F				13
15	F	F	F	T	F			13	
16	F	F	F	F	F				

There are 4 inputs.

Need (n+1) = 5 test cases.

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If ((A and D) or (C and E)) then …

	A	D	C	E	q	*A*	*D*	*C*	*E*
1	T	T	T	T	T				
2	T	T	T	F	T	10	6		
3	T	T	F	T	T	11	7		
4	T	T	F	F	T	12	8		
5	T	F	T	T	T			7	6
6	T	F	T	F	F		2		5
7	T	F	F	T	F		3	5	
8	T	F	F	F	F		4		
9	F	T	T	T	T			11	10
10	F	T	T	F	F	2			9
11	F	T	F	T	F	3		9	
12	F	T	F	F	F	4			
13	F	F	T	T	T			15	14
14	F	F	T	F	F				13
15	F	F	F	T	F			13	
16	F	F	F	F	F				

Test case:
{2, 5, 6, 7, 10}

Cover effects of
A : {10, 2}
D : {6, 2}
C : {7, 5}
E : {6, 5}

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If ((A and D) or (C and E)) then …

	A	D	C	E	q	*A*	*D*	*C*	*E*
1	T	T	T	T	T				
2	T	T	T	F	T	10	6		
3	T	T	F	T	T	11	7		
4	T	T	F	F	T	12	8		
5	T	F	T	T	T			7	6
6	T	F	T	F	F		2		5
7	T	F	F	T	F		3	5	
8	T	F	F	F	F		4		
9	F	T	T	T	T			11	10
10	F	T	T	F	F	2			9
11	F	T	F	T	F	3		9	
12	F	T	F	F	F	4			
13	F	F	T	T	T			15	14
14	F	F	T	F	F				13
15	F	F	F	T	F			13	
16	F	F	F	F	F				

Test case:
{2, 6, 9, 10, 11}

Cover effects of
A : {10, 2}
D : {6, 2}
C : {11, 9}
E : {10, 9}

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If ((A and D) or (C and E)) then …

	A	D	C	E	q	*A*	*D*	*C*	*E*
1	T	T	T	T	T				
2	T	T	T	F	T	10	6		
3	T	T	F	T	T	11	7		
4	T	T	F	F	T	12	8		
5	T	F	T	T	T			7	6
6	T	F	T	F	F		2		5
7	T	F	F	T	F		3	5	
8	T	F	F	F	F		4		
9	F	T	T	T	T			11	10
10	F	T	T	F	F	2			9
11	F	T	F	T	F	3		9	
12	F	T	F	F	F	4			
13	F	F	T	T	T			15	14
14	F	F	T	F	F				13
15	F	F	F	T	F			13	
16	F	F	F	F	F				

Test case:
{3, 7, 9, 10, 11}

Cover effects of
A : {11, 3}
D : {7, 3}
C : {11, 9}
E : {10, 9}

Example MC/DC

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

If ((A and D) or (C and E)) then …

	A	D	C	E	q	*A*	*D*	*C*	*E*
1	T	T	T	T	T				
2	T	T	T	F	T	10	6		
3	T	T	F	T	T	11	7		
4	T	T	F	F	T	12	8		
5	T	F	T	T	T			7	6
6	T	F	T	F	F		2		5
7	T	F	F	T	F		3	5	
8	T	F	F	F	F		4		
9	F	T	T	T	T			11	10
10	F	T	T	F	F	2			9
11	F	T	F	T	F	3		9	
12	F	T	F	F	F	4			
13	F	F	T	T	T			15	14
14	F	F	T	F	F				13
15	F	F	F	T	F			13	
16	F	F	F	F	F				

Any one of the following sets will achieve MC/DC:

{2, 5, 6, 7, 10}
{2, 6, 9, 10, 11}
{3, 7, 9, 10, 11}

Multiple Condition Coverage

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

Develop test cases to execute all possible combinations of condition.

In the previous example:

```
(age<25) and (sex=male) and (not married)
```

We need 8 test cases to test all 8 different combinations.

- For a decision with n inputs, multiple condition coverage requires 2^n tests.
- Example: the expression **`(A and B) or (A and C)`** has 3 inputs, but 4 conditions, because each occurrence of A is considered a unique condition.
- The maximum number of possible test cases is always 2^n, where n is the ***number of inputs***, not the number of conditions. In our example, $2^3 = 8$ test cases

Summary

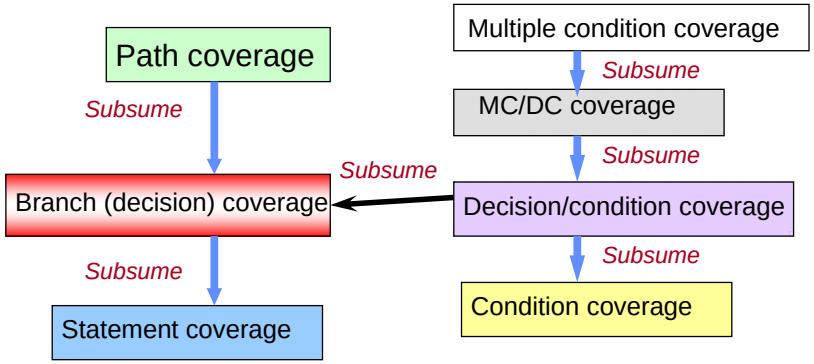
Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

	Statement coverage	Branch coverage	Condition coverage	Decision/ condition coverage	Multiple condition coverage
1. Each statement is executed at least once	Y	Y	Y	Y	Y
2. Each decision takes on all possible outcomes at least once	N	Y	N	Y	implicit
3. Each condition in a decision takes on all possible outcomes at least once	N	N	Y	Y	implicit
4. All possible combinations of condition outcomes in each decision occur at least once	N	N	N	N	Y

Subsumption Relation

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

Method X **subsumes** method Y if X tests at least what Y tests (possibly more).



When one test coverage metric subsumes another, a set of test cases that attains coverage in terms of the first coverage criterion also attains coverage wrt the subsumed criterion.

Subsumption Relation Example

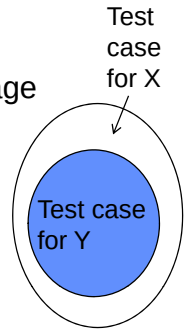
Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

Given a test set **T** = {tc1, tc2, tc3, tc4} (4 test cases)

If **T** achieves 100% path coverage, then **T** also achieves 100% branch coverage by the subsumption relation.

But, **T** may not achieve 100% condition coverage

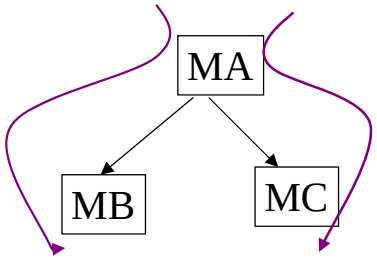
A coverage criterion X subsumes coverage criterion Y implies that X is ***stronger*** than Y, because all test cases required for Y are also required by X.



Call Coverage

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

- Measure whether we executed each function call.
- The hypothesis is that faults commonly occur in interfaces between modules.
- Also known as **call pair coverage** (try to achieve 100% call coverage (test all calls)



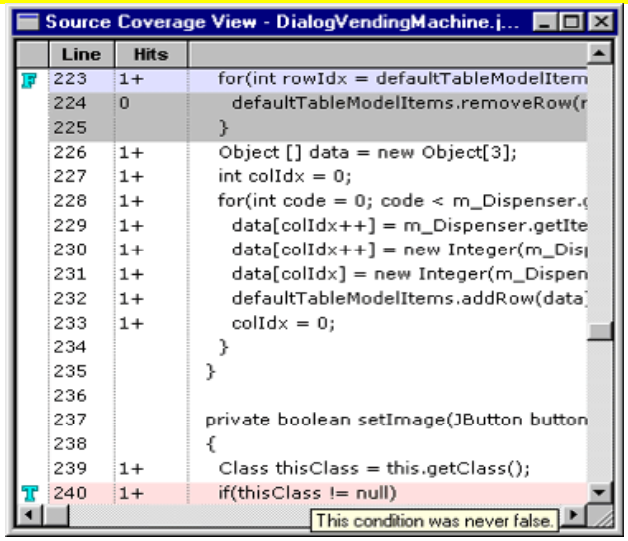
MA calls MB, and MA also calls MC

Test tool helps to compute the coverage

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool

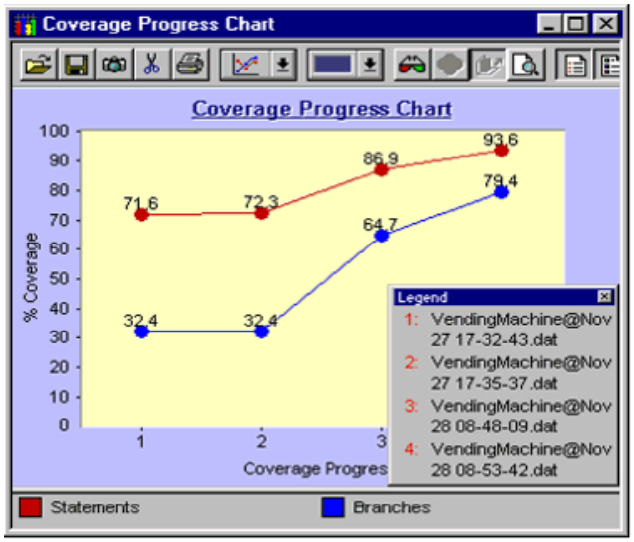
JCover for Java

Identify the statements executed during testing.



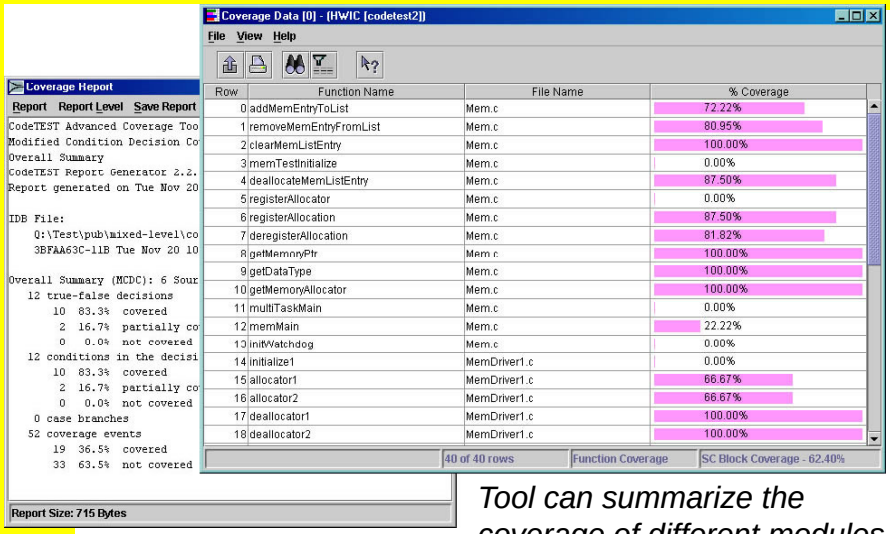
Coverage Tool: JCover

Introduction
Static Analysis
Path-based
Coverage
Condition/ decision
MC/DC
Multiple C
Subsumption
Call Coverage
Tool



As we run more test cases, the coverage increases

Coverage Tool: For C



Tool can summarize the coverage of different modules.

Question: How many Paths?


```
1  total:=0
2  n:=1
3   While (n<100) and not end_of_file do
4     begin
5           read(x);
6           if x<1 then
7                 total:=total*x
8           else  total:=total+x
9           n:=n+1
10    end;
11   write(sum);
```


Problems with Path-based Testing

- ☹ Number of unique paths usually very large. Example: a large embedded system may contain millions of program paths!
- ☹ Impractical to test each program path
- ☹ Testing all paths still can't find missing requirements
- ☹ Executing each path is no guarantee that the software satisfy requirements, because:
  - programmer may have written wrong function
  - software may have missing paths
  - a path may require specific data to show defects

Defect Detection depends on test input

Incorrect Version

```
1  read (a, b)
2  if (a>b) then
3  a=2*b
4  Write (a)
```

Correct Version

```
1  read (a, b)
2  if (a>b) then
3  a=2+b
4  Write (a)
```

The correct program

For input of a=5, b=2, both programs will give the correct output (i.e. a=4).

The defect will not be detected!!

But, for input of a=5, b=0, **Incorrect version** will give the wrong output (a=0 vs. a=2)! Then, we can detect the failure.

Why Can't Achieve 100% Coverage?

- ☹ Infeasible paths or conditions
- ☹ unreachable or redundant code (**dead code**, which cannot be executed under any circumstances)
- ☹ insufficient test cases
- ☹ Exception handling code
- ☹ Interrupt

Not practical to aim for 100% coverage!

Basis Path Testing

McCabe proposes the basis path method, which requires fewer test cases compared to all path testing.

This method constructs test cases that are guaranteed to executed every statement at least once.

Steps:

1. Use the design or code, draw the control flow graph
2. Determine the **cyclomatic number** or **complexity**, V, of the flow graph
3. Determine a **basis set** of independent paths
4. Prepare test cases that will force execution of each path in the basis set.

Cyclomatic Number for a *Graph*

From graph theory,

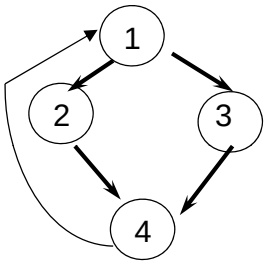
the cyclomatic number v(g) of a graph g with n vertices, e edges, and 1 connected component is v(g)=e-n+1

Theorem

In a strongly connected graph g, the cyclomatic number is equal to the max. number of linearly independent paths.

A graph is ***strongly connected*** if given 2 nodes a and b, there exists a path from a to b and a path from b to a.

The linearly independent paths is like a basis.



A strongly connected graph

Cyclomatic Number for a *Control flow Graph*

For an arbitrary program, its control flow graph is not strongly connected.

Example: we can start from the entry node and can get to any other nodes. But, we may not go from a node in the middle and back to the entry node.

How to make it strongly connected?

By adding 1 edge from the exit node to the entry node, we created a strongly connected graph.

Thus, V for a control flow graph is ***calculated slightly differently***.

McCabe's Cyclomatic Number, V

For a control flow graph,

V = e - n +2

e is the number of edges, n is the number of nodes

- Program complexity is dependent on the number of linearly independent paths in the program graph.
- V is the minimum number of paths which are necessary without overlapping to **cover** combinations of all edges.

Another Method to Compute V

Introduction

Static Analysis

Path-based

Basis path

▸ V

For **structured programs** (no GOTO):

V = number of binary decision nodes + 1,

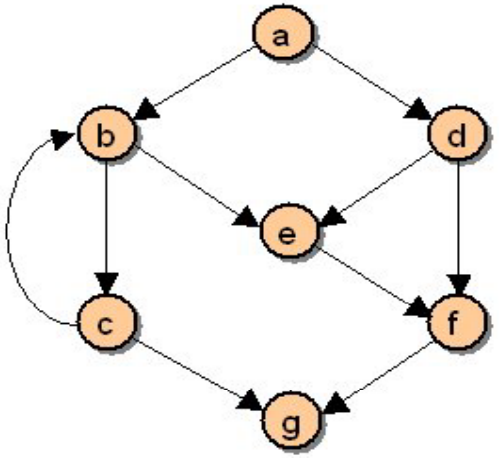
- IF statement is counted as 1 binary decision.
- 3-way CASE is counted as 2 binary decisions.
- n-way CASE is counted as (n-1) binary decisions.
- Loop is counted as 1 binary decision.

Summary: we can compute McCabe's V in 2 ways:

(1) V=e-n+2

(2) V=number of binary decision + 1

Example FG



e=10

n=7

V=e-n+2

=10-7+2

=**5**

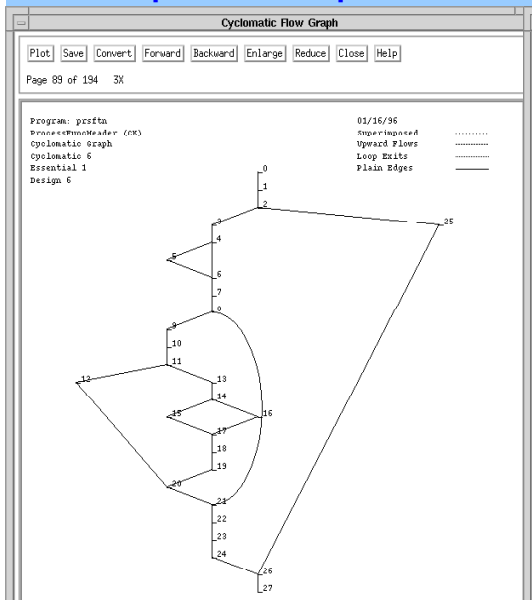
Binary decision = 4

V = 4 + 1

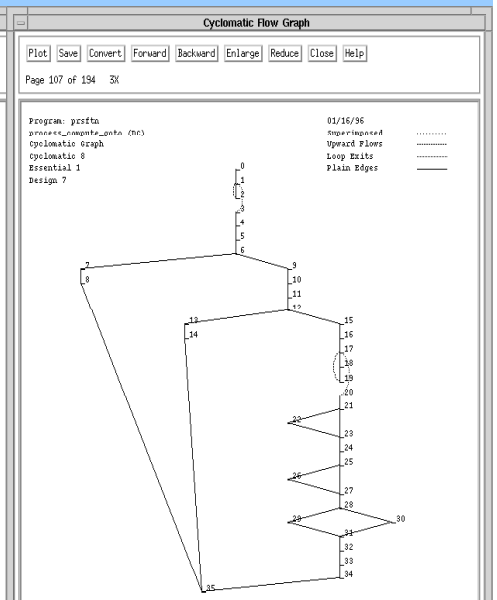
= **5**

Therefore, there are 5 basis paths.

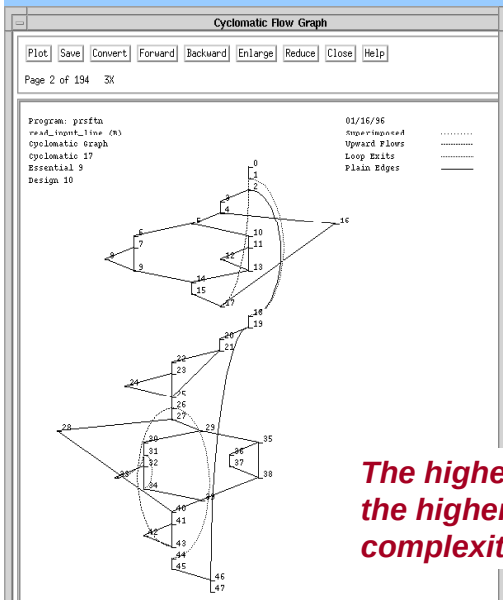
Example Flow Graph V=6



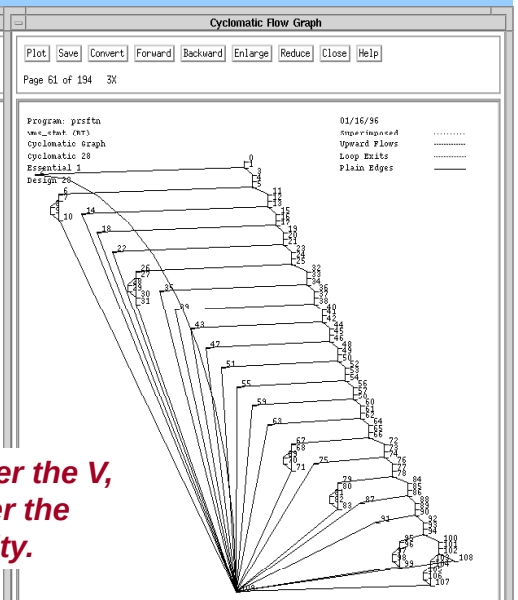
V=8



V=17



V=28



The higher the V, the higher the complexity.

Basis Path Testing

V is designed to indicate a program's testability and understandability (maintainability)

- McCabe uses V to find a minimum number of tests
- He suggests that a good upper limit for V is 10 (when V>10, we should re-design our code to reduce V)

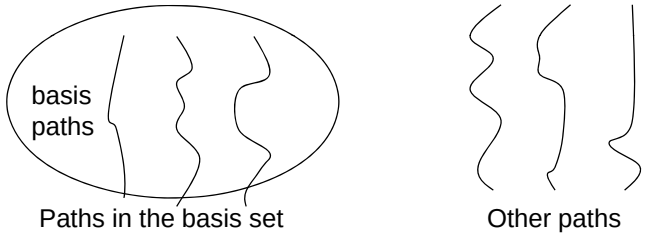
If the basis is 'OK', we could hope that everything that can be expressed in terms of the basis is also 'OK'.

Reference: http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/235/sttoc.htm

Basis set of Paths

Linearly independent paths (or Basis paths):

- no path in the **basis set** can be constructed from a combination of other paths in the basis set
- any path not in the basis set can be formed as a combination of paths in the basis set



An independent path (basis path) is any path through the code that exercises at least one new statement or a new condition.

For our example FG,

there are 5 basis paths:

Edge	abcbefg	abcg	abefg	adefg	adfg	abcbcg
ab	1	1	1			1
ad				1	1	
bc	1	1				2
cb	1					1
be	1		1			
de				1		
df					1	
ef	1		1	1		
cg		1				1
fg	1		1	1	1	

***Path* abcbcg = abcbefg + abcg – abefg**

Focus on the edges of each path. Then add or subtract the edges belong to these 3 paths ((abcbefg) + (abcg) - (abefg)), we get the edges for the path abcbcg.

Generating Basis Paths

1. Select an arbitrary path that represents a typical function, and not just an error or exceptional condition - call this the **baseline path**.
2. Select 2nd path by locating the first decision in the baseline path and flipping its results while holding the maximum number of the original decisions the same.
3. Set back the first decision. Flip the second decision in the baseline path while holding all other decisions to their baseline values.
4. Continue until every decision has been flipped.

Identifying Basis Paths

Assume we pick **abcbefg** as the **baseline path**.

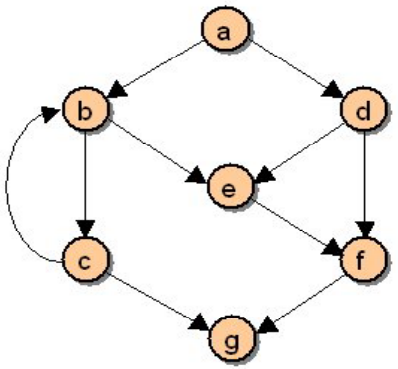
For the second path, we flip the first decision (node a), and get the path **adefg**

Next, we flip the second decision of the baseline path (node b), and get the path **abefg**.

Next, we flip the 3rd decision of the baseline path (node c) and get path **abcg**.

Now, no more decisions left in the baseline path. So, follow the second path.

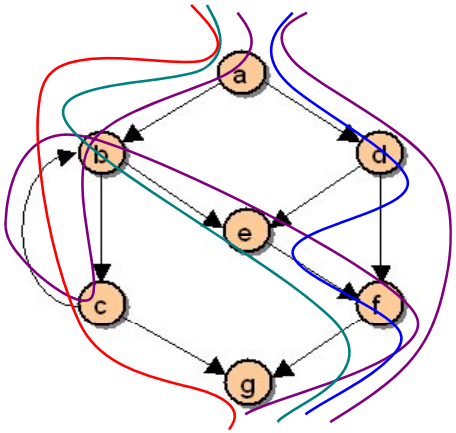
We flip node d, and get the path **adfg**.



Note that the 2nd to 5th paths try to follow as close to the baseline path as possible; this eases the design of test input.

Identifying Basis Paths

1st path: **abcbefg**
2nd path: **adefg**
3rd path: **abefg**
4th path: **abcg**
5th path: **adfg**

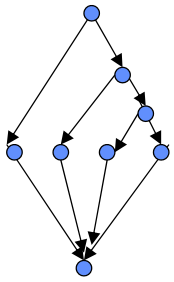


Criticism of Basis Path Testing

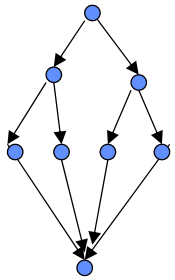
- ☹ only consider the flowgraph
- ☹ no consideration of the complex decision expression, different control flow statements (IF versus CASE versus loop), and nested structure (e.g., one IF inside another IF)
- ☹ using McCabe's method to generate basis paths can sometimes create infeasible paths!
- ☹ testing the set of basis paths is not sufficient. Need to use other testing technique, like EP, BV.

Question

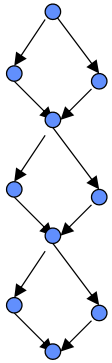
What is V for G_1, G_2, G_3?



G_1



G_2



G_3

G_3 seems to be more complex and should require more testing!

Review Exercise 1

For **Program CountWord**,

1. How many test cases are needed to achieve:

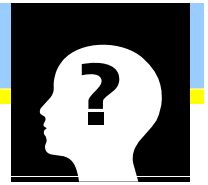
   100% statement coverage, 100% branch coverage,

   100% condition coverage?

2. Create test cases to exercise all basis paths.

   What is the achieved coverage of your test set in terms of statement, branch and condition coverage?

Review Exercise 2



For the **liability procedure,**

1. Draw its flowgraph.
2. How many test cases to achieve
   100% statement coverage? 100% branch coverage?
   100% path coverage?
3. Identify its basis paths.

```
1  procedure liability(age, sex, married, premium);
2  begin
3    premium := 500;
4    if ((age<25) and (sex=male) and (not married))
   then
5         premium := premium + 1500;
6    else (if (married or (sex = female)) then
7                premium := premium-200;
8          if ((age > 45) and (age < 65)) then
9                premium := premium-100;)
10 end;
```

- *Young single man pays a higher premium.*
- *Married or female pay a reduced rate*
- *Older people pay a reduced rate*