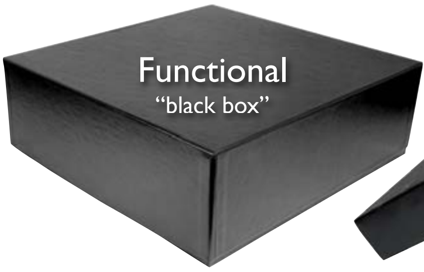
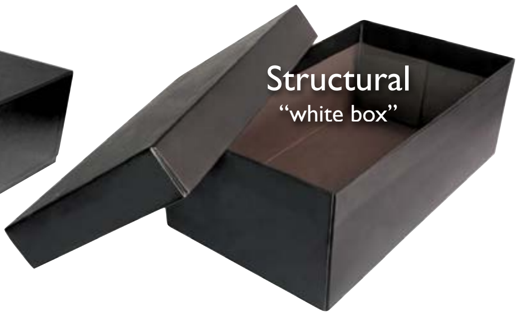
From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 14 and Pezze + Young, “Software Testing and Analysis”, Chapters 12-13

Testing Tactics



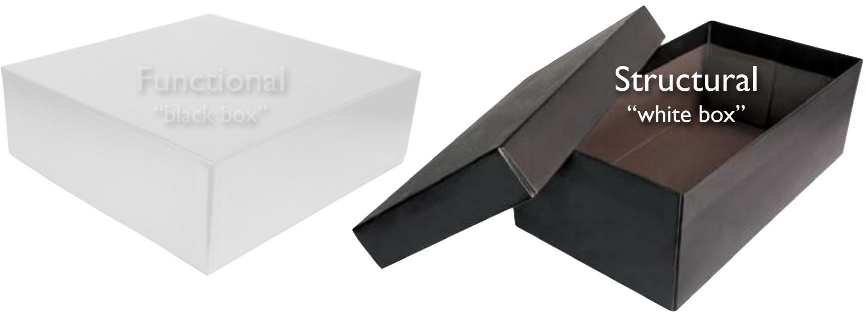


- Tests based on *spec*
- Test covers as much *specified* behavior as possible

- Tests based on *code*
- Test covers as much *implemented* behavior as possible

In contrast to *functional tests* (discussed the last time), *structural* tests are based on the code structure

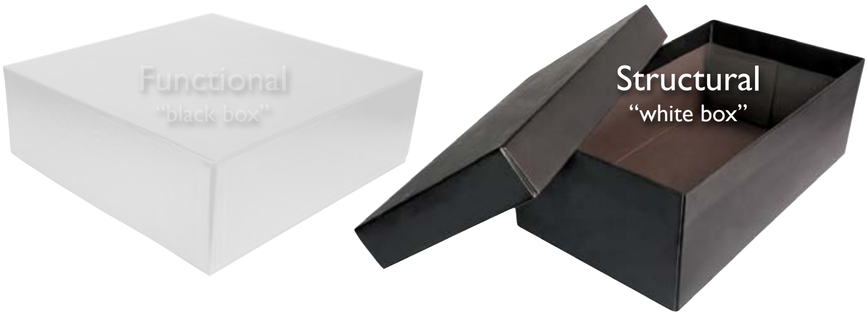
Why Structural?



- If a part of the program is never executed, a defect may loom in that part
  A “part” can be a statement, function, transition. condition…
- Attractive because automated

Structural tests are automated – and can be much more fine-grained than functional tests.

Why Structural?



- Complements functional tests
  Run functional tests first, then measure what is missing
- Can cover low-level details missed in high-level specification

Typically, both techniques are used.

A Challenge

```
class Roots {
    // Solve ax² + bx + c = 0
    public roots(double a, double b, double c)
    { … }

    // Result: values for x
    double root_one, root_two;
}
```

- Which values for *a, b, c* should we test?
  assuming a, b, c, were 32-bit integers, we'd have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

Recall this example from last lecture.

The Code

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }

    else if (q == 0) {
        // code for handling one root
    }

    else {
        // code for handling no roots
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Test this case

and this

and this!

If we know the code ("white box") and thus the structure, we can design test cases accordingly

The Test Cases

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        // code for handling one root
    }
    else {
        // code for handling no roots
    }
}
```

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(a, b, c) = (3, 4, 1)

(a, b, c) = (0, 0, 1)

(a, b, c) = (3, 2, 1)

Finding appropriate input values is a challenge in itself which may require external theory – but in this case, the external theory is just maths.

A Defect

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }
}
```

code must handle a = 0

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

(a, b, c) = (0, 0, 1)

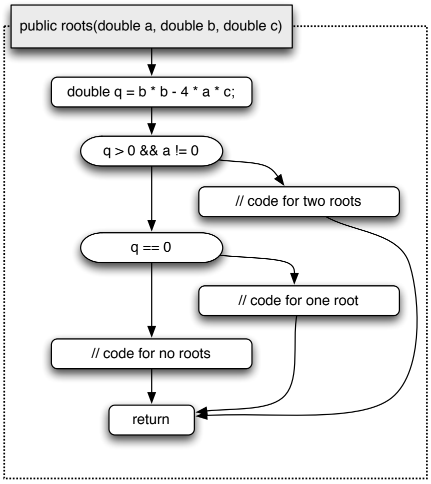
The test case that executes the q = 0 branch reveals a defect – the case a = 0

Expressing Structure

```
// Solve ax² + bx + c = 0
public roots(double a, double b, double c)
{
    double q = b * b - 4 * a * c;
    if (q > 0 && a ≠ 0) {
        // code for handling two roots
    }
    else if (q == 0) {
        x = (-b) / (2 * a);
    }
    else {
        // code for handling no roots
    }
}
```

What is relevant in her is the program structure – the failure occurs only if a specific condition is true and a specific branch is taken.

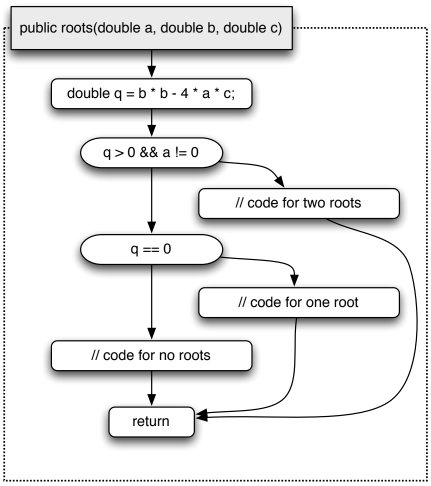
Control Flow Graph



- A *control flow graph* expresses paths of program execution
- *Nodes* are *basic blocks* – sequences of statements with one entry and one exit point
- *Edges* represent *control flow* – the possibility that the program execution proceeds from the end of one basic block to the beginning of another

To express structure, we turn the program into a *control flow graph*, where statements are represented as nodes, and edges show the possible control flow between statements.

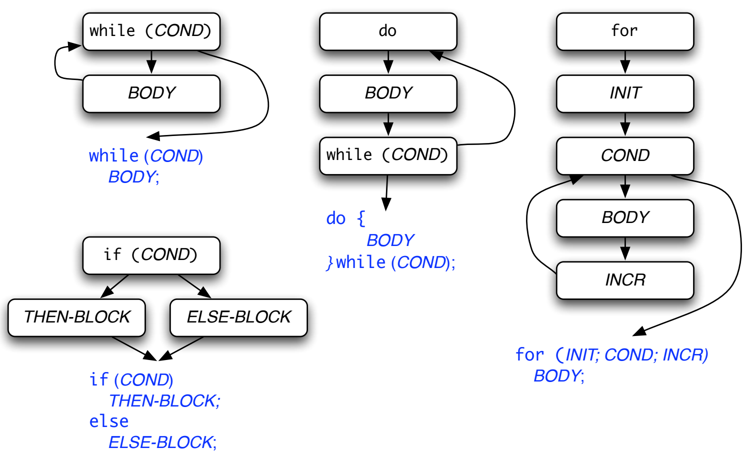
Structural Testing



- The CFG can serve as an *adequacy criterion* for test cases
- The more parts are covered (executed), the higher the chance of a test to uncover a defect
- "parts" can be: nodes, edges, paths, conditions…

To talk about structure, we turn the program into a *control flow graph*, where statements are represented as nodes, and edges show the possible control flow between statements.

Control Flow Patterns



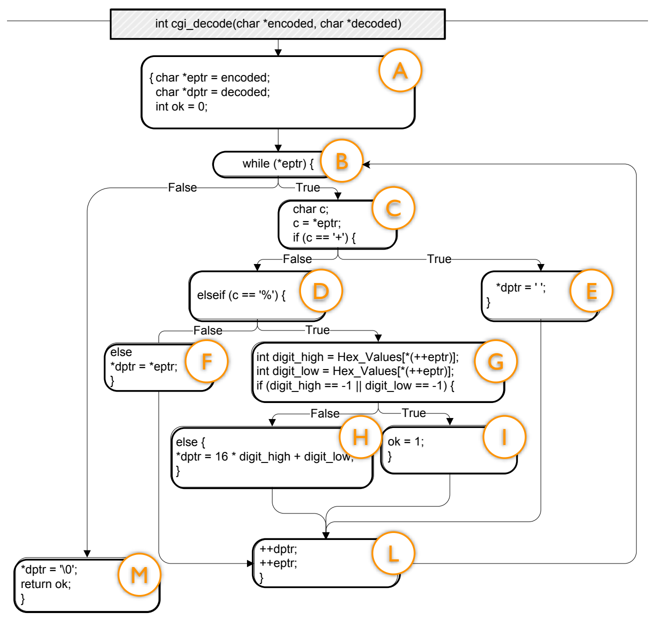
Every part of the program induces its own patterns in the CFG.

cgi_decode

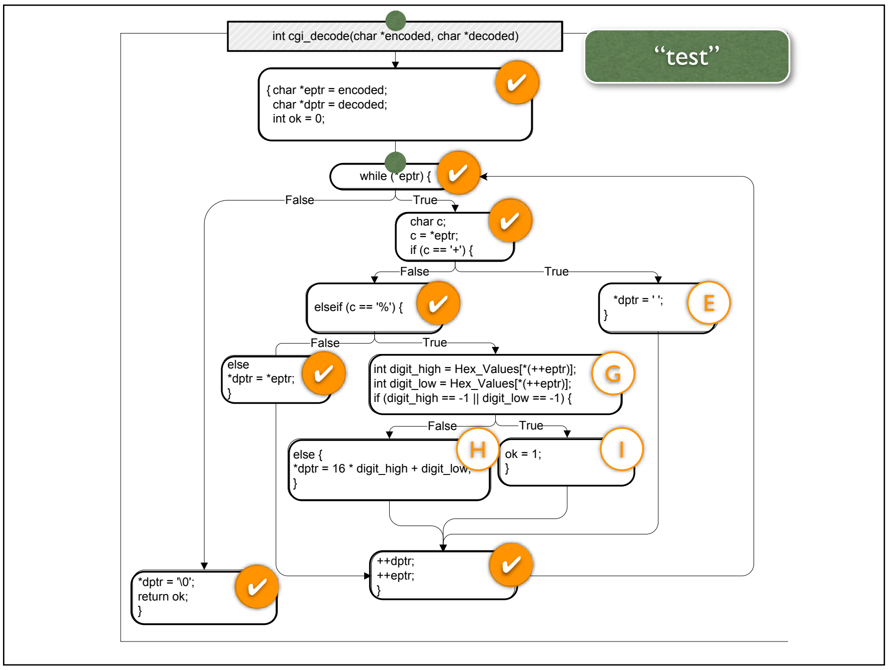
```
/**
  * @title cgi_decode
  * @desc
  * Translate a string from the CGI encoding to plain ascii text
  * '+' becomes space, %xx becomes byte with hex value xx,
  * other alphanumeric characters map to themselves
  *
  * returns 0 for success, positive for erroneous input
  * 1 = bad hexadecimal digit
  */

int cgi_decode(char *encoded, char *decoded)
{
    char *eptr = encoded;
    char *dptr = decoded;
    int ok = 0;
```

Here's an ongoing example. The function cgi_decode translates a CGI-encoded string (i.e., from a Web form) to a plain ASCII string, reversing the encoding applied by the common gateway interface (CGI) on common Web servers.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)

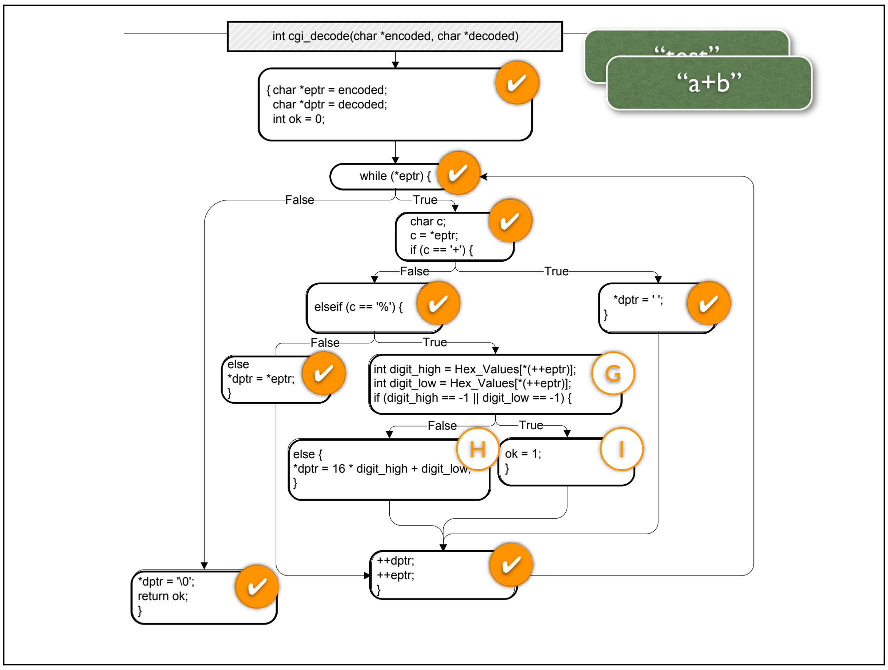
```
    while (*eptr)  /* loop to end of string ('\0' character) */
    {
        char c;
        c = *eptr;
        if (c == '+') {  /* '+' maps to blank */
            *dptr = ' ';
        } else if (c == '%') { /* '%xx' is hex for char xx */
            int digit_high = Hex_Values[*(++eptr)];
            int digit_low  = Hex_Values[*(++eptr)];
            if (digit_high == -1 || digit_low == -1)
                ok = 1; /* Bad return code */
            else
                *dptr = 16 * digit_high + digit_low;
        } else { /* All other characters map to themselves */
            *dptr = *eptr;
        }
        ++dptr; ++eptr;
    }

    *dptr = '\0';    /* Null terminator for string */
    return ok;
}
```



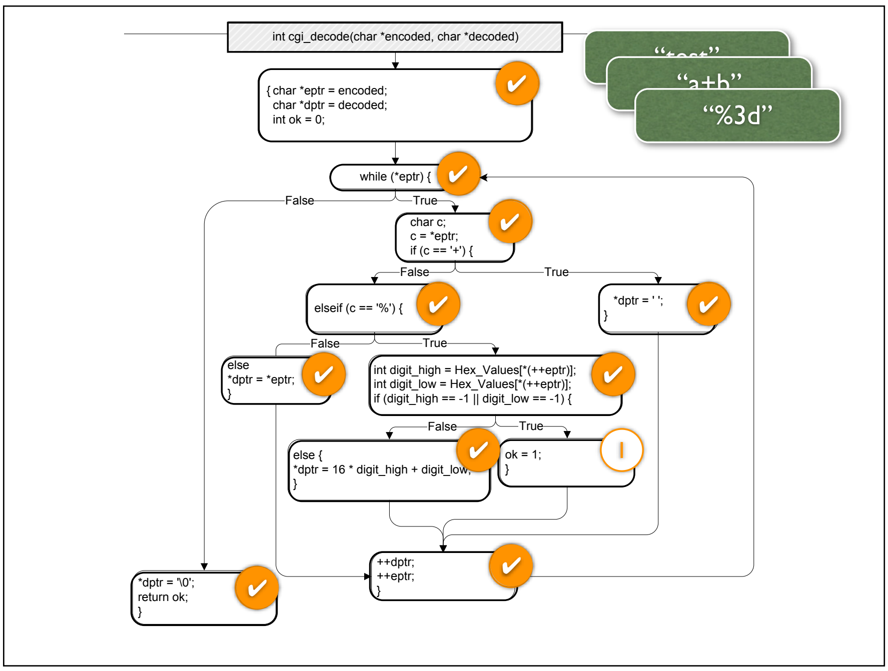
This is what cgi_decode looks as a CFG.
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)



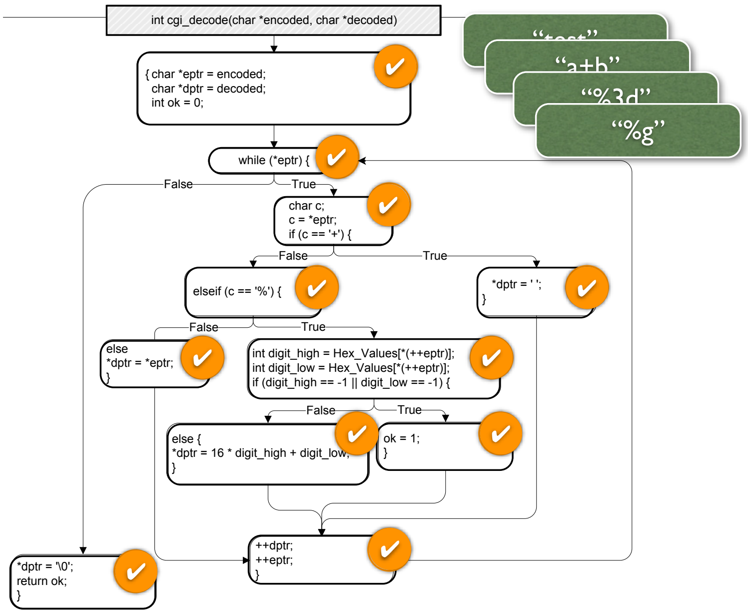
While the program is executed, one statement (or basic block) after the other is covered – i.e., executed at least once – but not all of them. Here, the input is "test"; checkmarks indicate executed blocks.



We'd like to test every statement, so we come up with more test cases.



We'd like to test every statement, so we come up with more test cases.



This is an interesting boundary test case, as it may cause non-deterministic behavior. Can you see why?

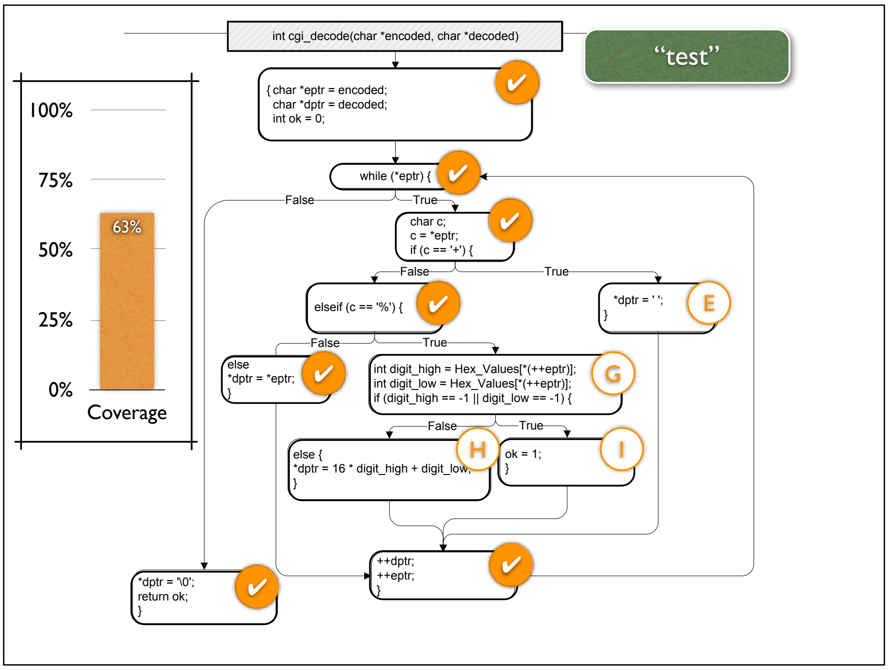
Test Adequacy Criteria

- How do we know a test suite is "good enough"?
- A *test adequacy criterion* is a predicate that is true or false for a pair ⟨*program, test suite*⟩
- Usually expressed in form of a rule – e.g., "all statements must be covered"

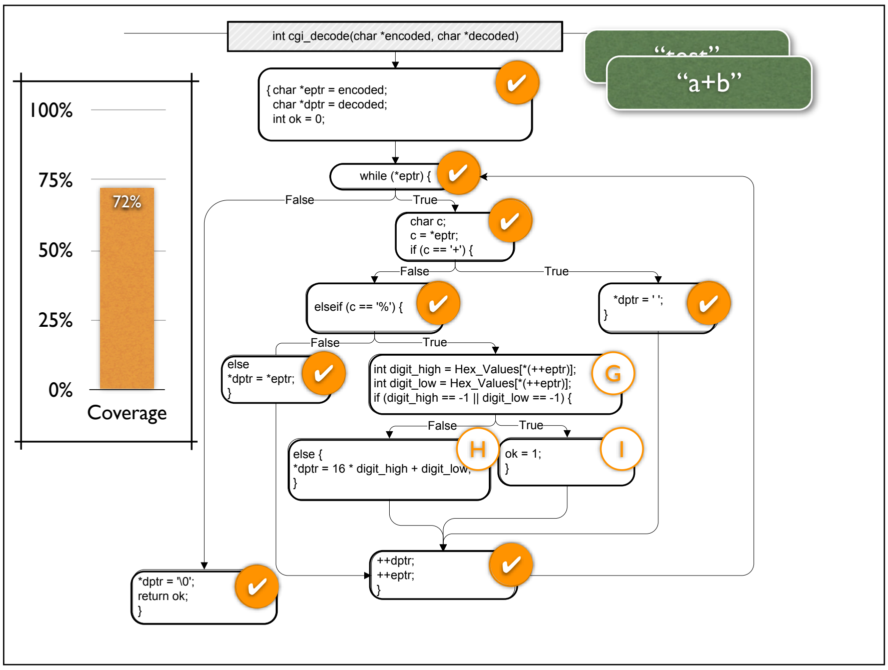
Statement Testing

- Adequacy criterion: each statement (or node in the CFG) must be executed at least once
- Rationale: a defect in a statement can only be revealed by executing the defect
- Coverage: $\dfrac{\text{\# executed statements}}{\text{\# statements}}$

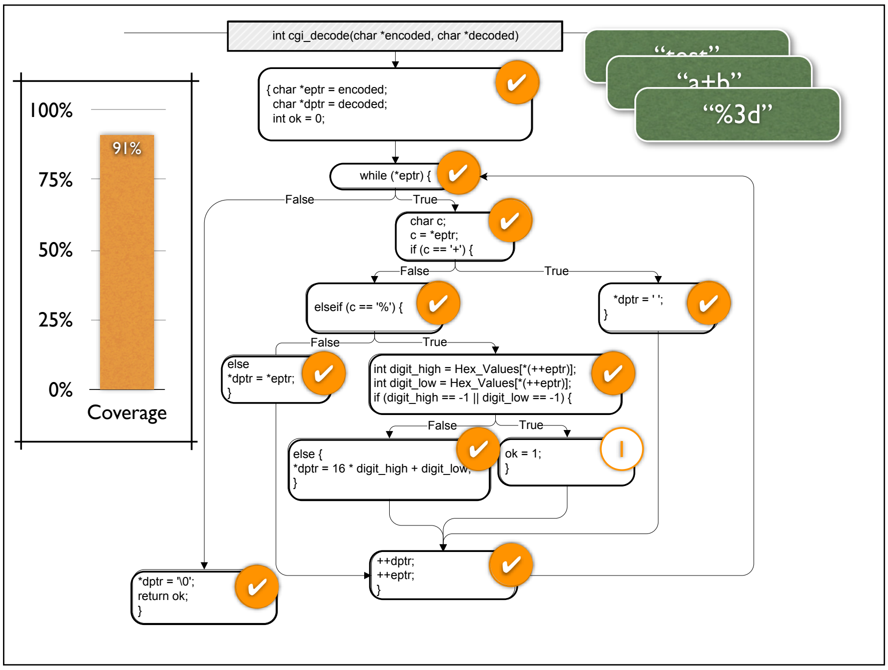
(from Pezze + Young, "Software Testing and Analysis", Chapter 12)

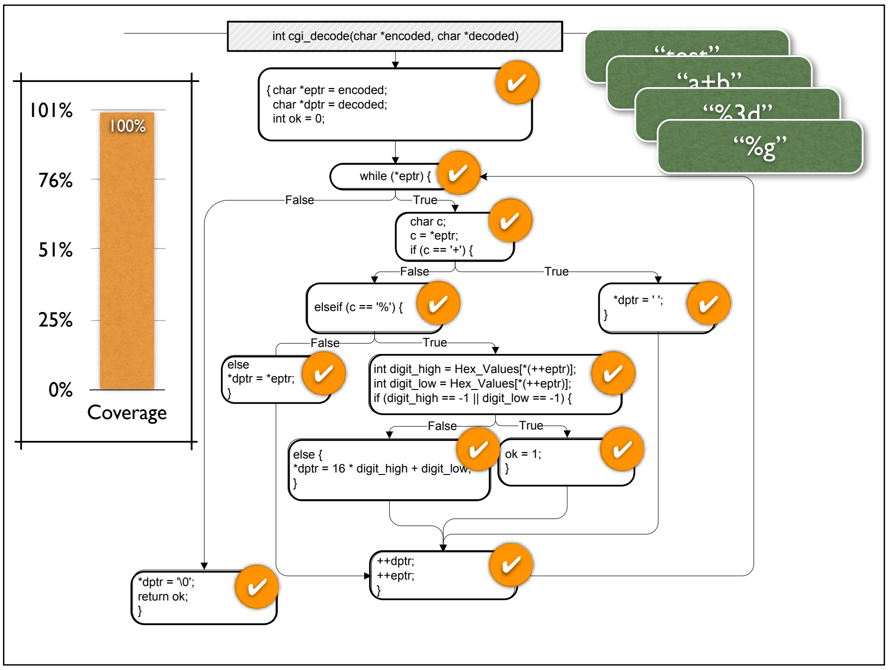


The initial coverage is 7/11 blocks = 63%. We could also count the statements instead (here: 14/20 = 70%), but conceptually, this makes no difference.



and the coverage increases with each additionally executed statement…





… until we reach 100% block coverage (which is 100% statement coverage, too).

Computing Coverage

- Coverage is computed automatically while the program executes
- Requires *instrumentation* at compile time
  With GCC, for instance, use options -ftest-coverage -fprofile-arcs
- After execution, *coverage tool* assesses and summarizes results
  With GCC, use "gcov *source-file*" to obtain readable .gcov file

For Java, use jcoverage or like tools.

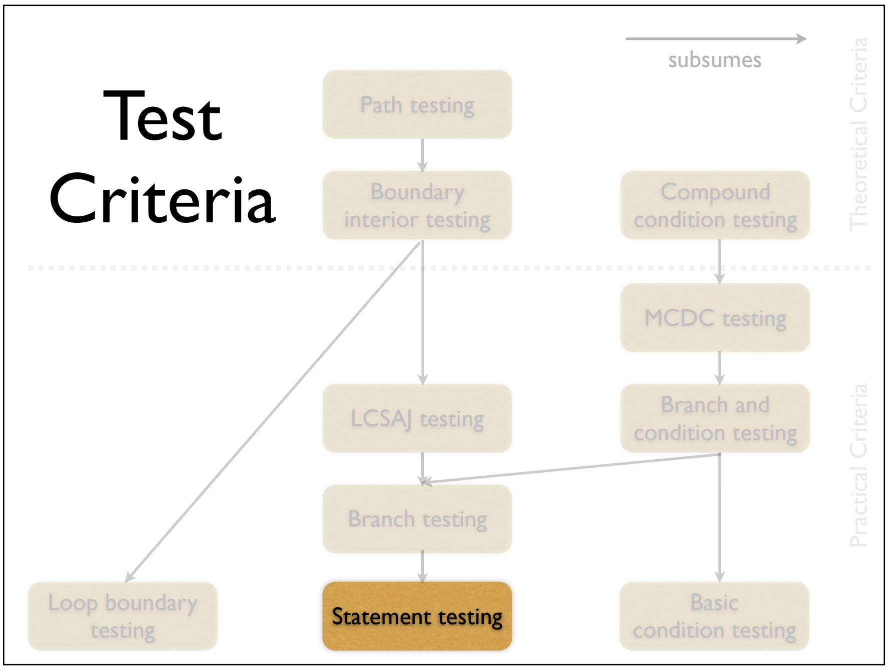
Pippin: cgi_encode — less — 80×24

```
    4:   18:    int ok = 0;
    -:   19:
   38:   20:    while (*eptr)  /* loop to end of string ('\0' character) */
    -:   21:    {
    -:   22:        char c;
   30:   23:        c = *eptr;
   30:   24:        if (c == '+') {  /* '+' maps to blank */
    1:   25:            *dptr = ' ';
   29:   26:        } else if (c == '%') { /* '%xx' is hex for char xx */
    3:   27:            int digit_high = Hex_Values[*(++eptr)];
    3:   28:            int digit_low  = Hex_Values[*(++eptr)];
    5:   29:            if (digit_high == -1 || digit_low == -1)
    2:   30:                ok = 1; /* Bad return code */
    -:   31:            else
    1:   32:                *dptr = 16 * digit_high + digit_low;
    -:   33:        } else { /* All other characters map to themselves */
   26:   34:            *dptr = *eptr;
    -:   35:        }
   30:   36:        ++dptr; ++eptr;
    -:   37:    }
    4:   38:    *dptr = '\0';    /* Null terminator for string */
    4:   39:    return ok;
    -:   40:}
(END)
```

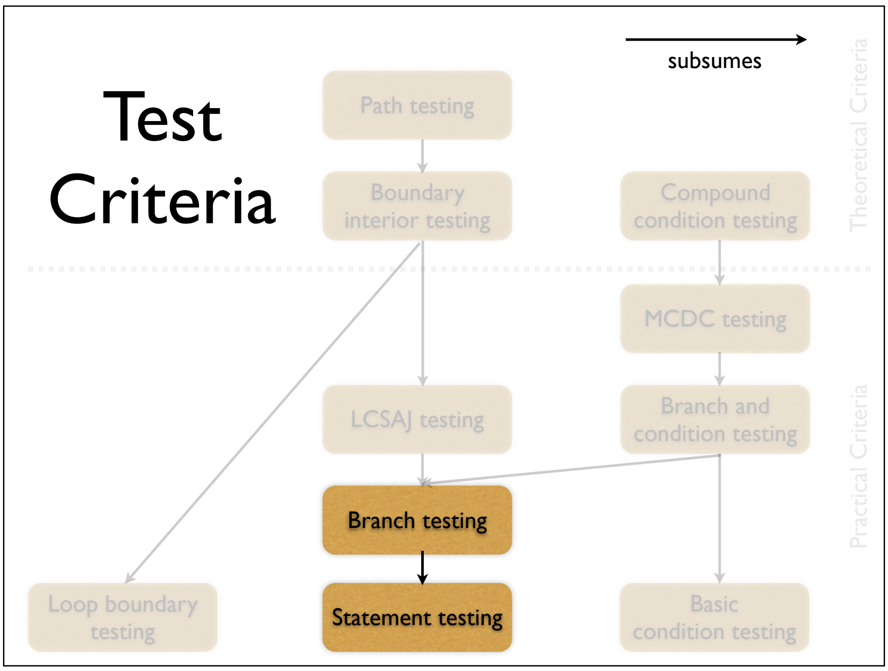
This is the output of the GCOV coverage tool for cgi_decode. Each statement (each line) is annotated with the number of executions so far. Zero executions is suspicious and would be marked by "#####"; the tag "–" stands for lines without executable code.

Demo

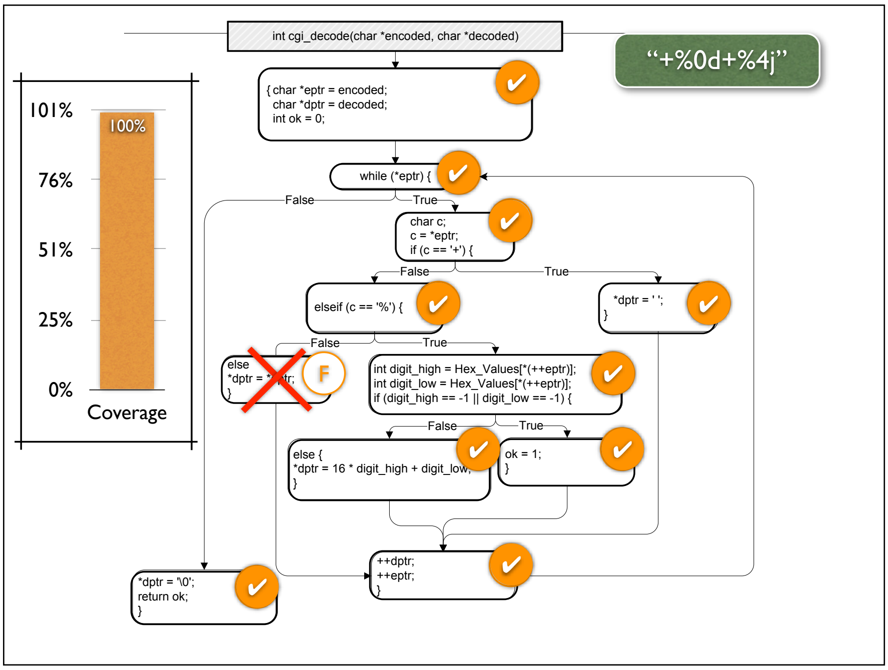
See the package “cgi_decode.zip” on the course page for instructions on how to do this yourself.



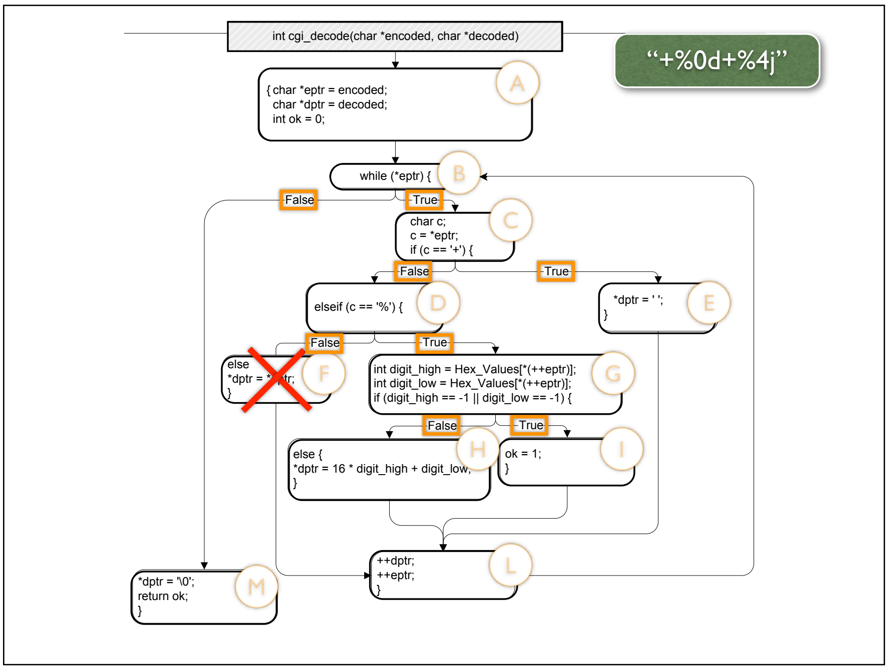
Statement testing is a simple criterion for assessing the adequacy of a test suite – but there are many more such criteria.



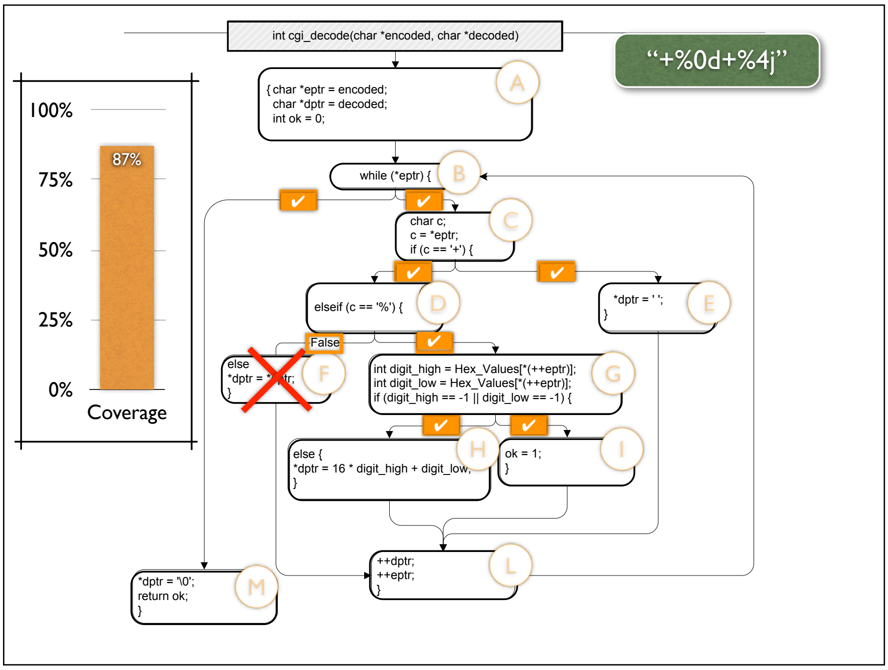
As an example, consider branch testing, which is a criterion that subsumes statement testing. In other words, if the branch testing criterion is satisfied by a pair ⟨program, test suite⟩, so is the statement testing criterion for the same pair.



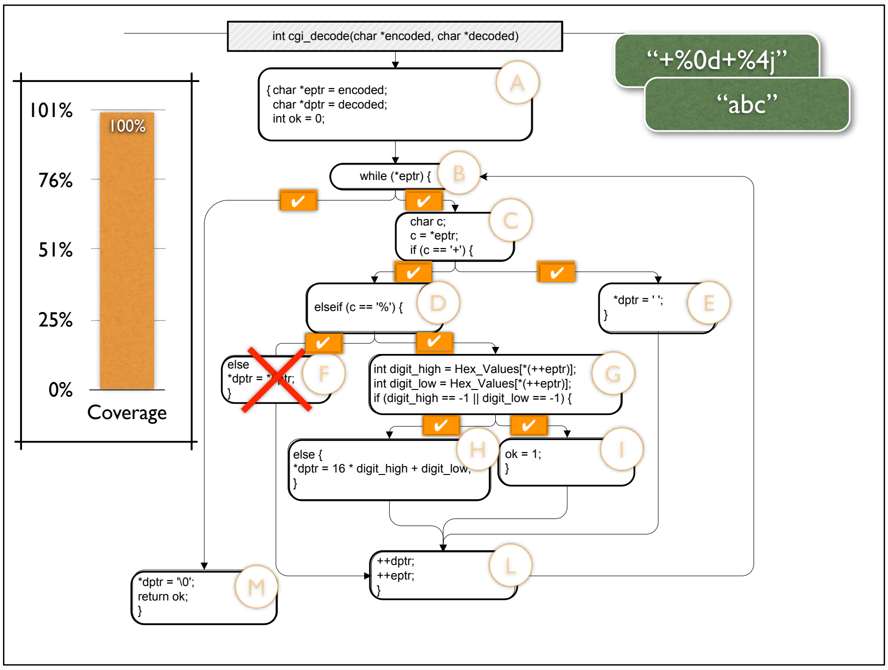
Why is branch testing useful? Assume block F were missing (= a defect). Then, we could achieve 100% statement coverage without ever triggering the defect.



If we focus on whether branches have been taken, though, we get a different picture.



Here, we'd find that the test case executes only 7 out of 8 branches, or 87%.



With another test case, we can cover this remaining branch – and find the defect.

Branch Testing

- Adequacy criterion: each branch in the CFG must be executed at least once
- Coverage: $\frac{\text{\# executed branches}}{\text{\# branches}}$
- Subsumes statement testing criterion
  because traversing all edges implies traversing all nodes
- Most widely used criterion in industry

(from Pezze + Young, "Software Testing and Analysis", Chapter 12)

Condition Testing

- Consider the defect
  `(digit_high == 1 || digit_low == -1)`
  `// should be -1`
- Branch adequacy criterion can be achieved by changing only `digit_low`
  i.e., the defective sub-expression may never determine the result
- Faulty sub-condition is never tested
  although we tested both outcomes of the branch

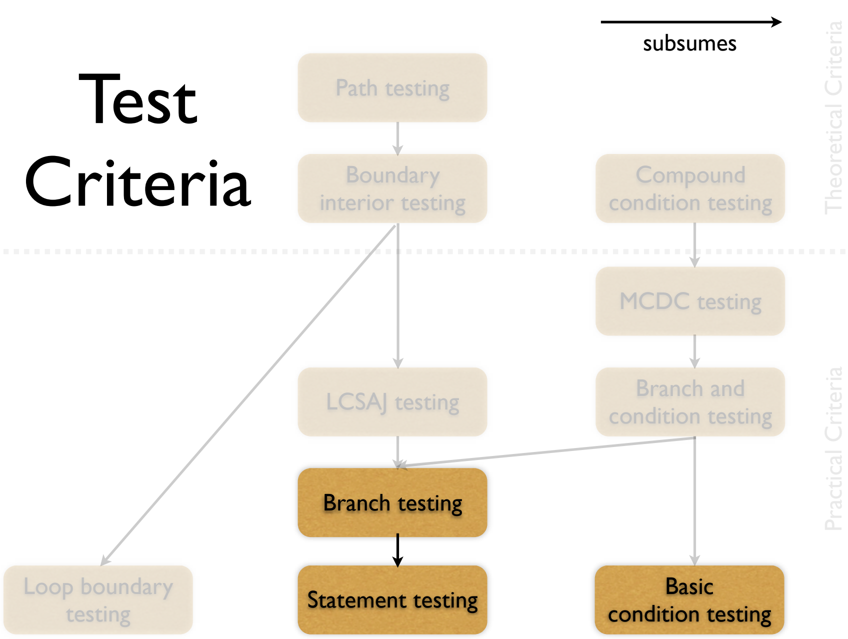
Condition Testing

- Key idea: also cover *individual conditions* in compound boolean expression
  e.g., both parts of `digit_high == 1 || digit_low == -1`

Condition Testing

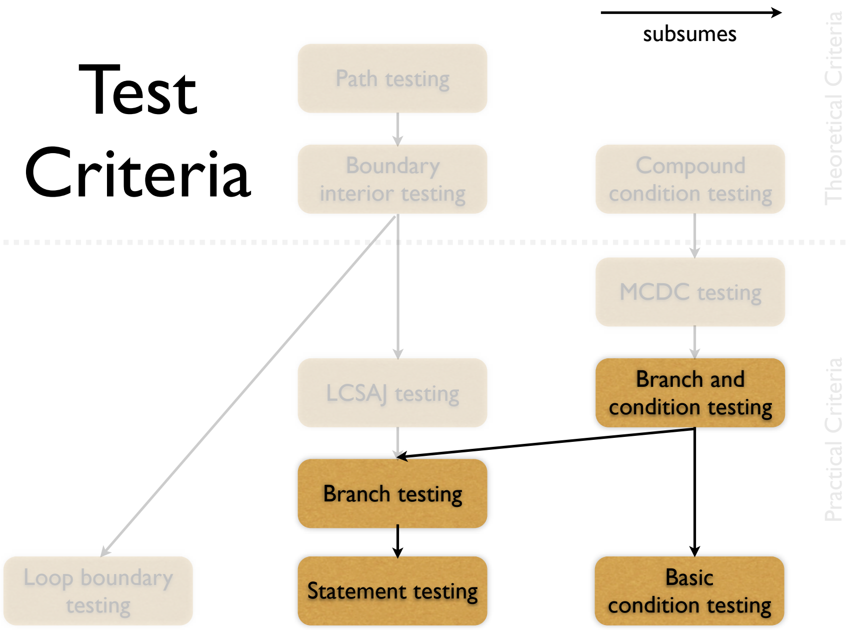
- Adequacy criterion: each basic condition must be evaluated at least once
- Coverage:
  $$\frac{\#\text{ truth values taken by all basic conditions}}{2 * \#\text{ basic conditions}}$$
- Example: "test+%9k%k9"
  100% basic condition coverage
  but only 87% branch coverage

Test Criteria



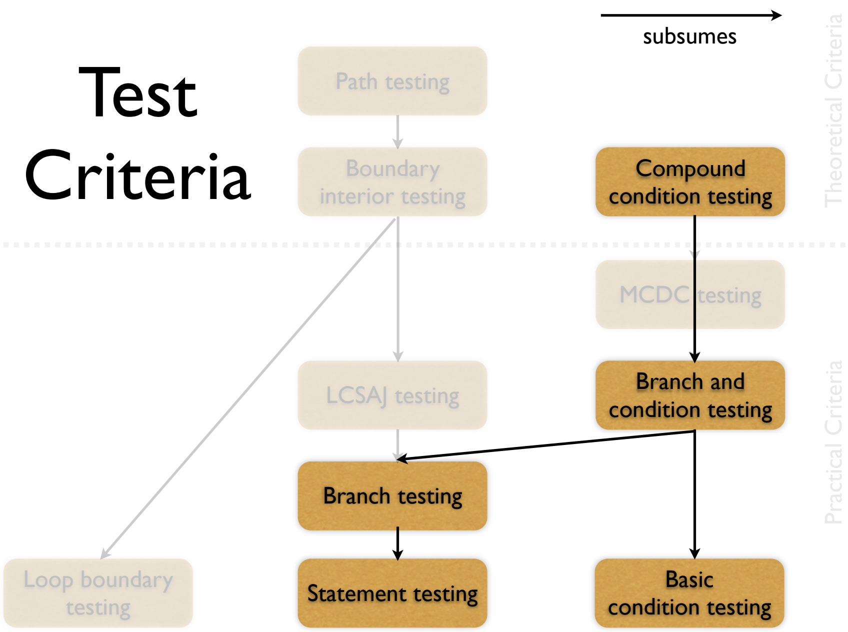
The basic condition criterion is not comparable with branch or statement coverage criteria – neither implies (subsumes) the other.

Test Criteria



The idea here is to cover both branch and condition testing – by covering all conditions and all decisions. That is, every sub-condition must be true and false, but the entire condition just as well.

Test Criteria

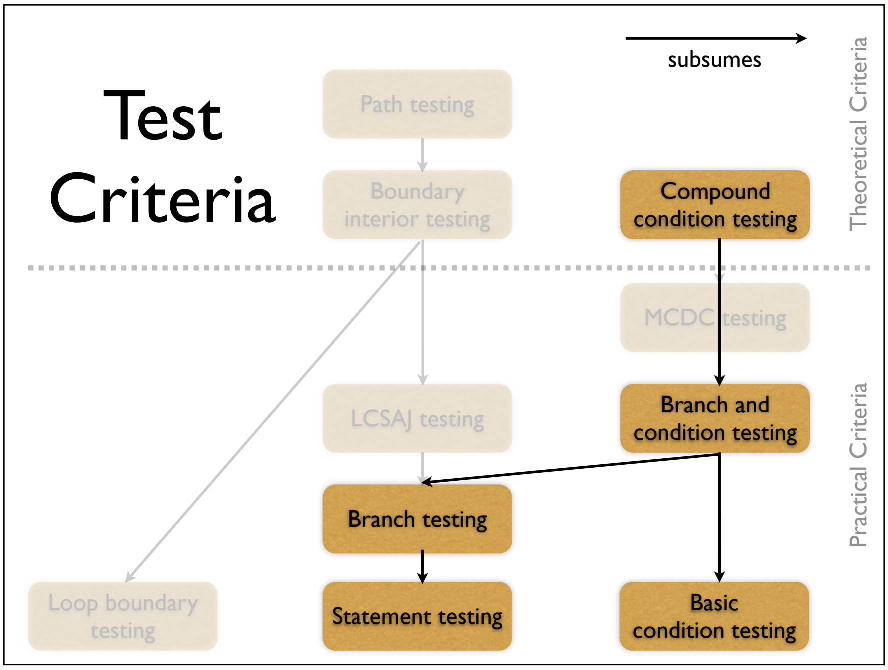


Another idea might be simply to test all possible combinations. This is called compound condition testing.

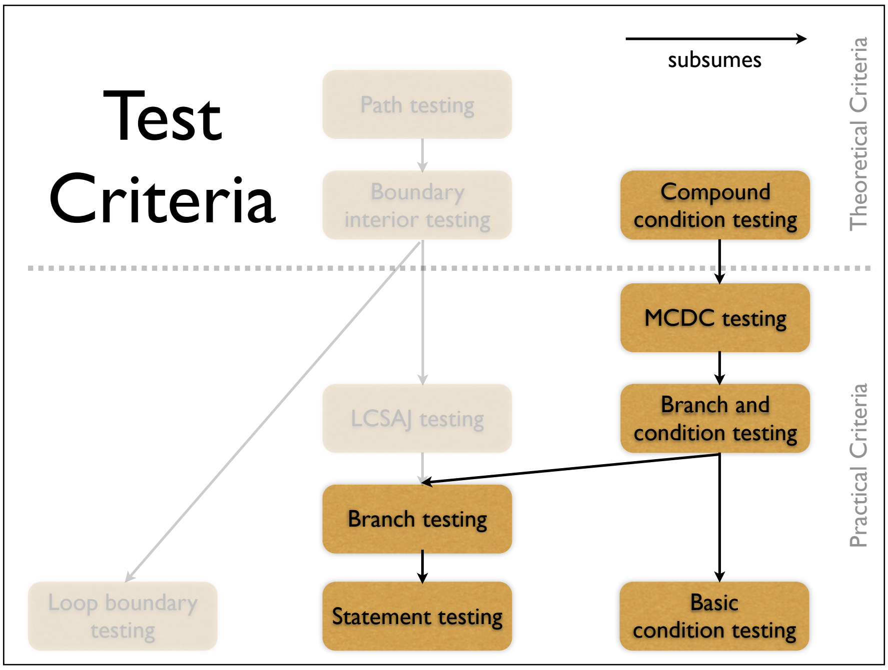
Compound Conditions

- Assume `(((a ∨ b) ∧ c) ∨ d) ∧ e)`
- We need 13 tests to cover all possible combinations
- In general case, we get a combinatorial explosion

Test Case	a	b	c	d	e
(1)	True	–	True	–	True
(2)	False	True	True	–	True
(3)	True	–	False	True	True
(4)	False	True	False	True	True
(5)	False	False	–	True	True
(6)	True	–	True	–	False
(7)	False	True	True	–	False
(8)	True	–	False	True	False
(9)	False	True	False	True	False
(10)	False	False	–	True	False
(11)	True	–	False	False	–
(12)	False	True	False	False	–
(13)	False	False	–	False	–



The combinatorial explosion is the reason why compound condition testing is a theoretical, rather than a practical criterion.



A possible compromise is MCDC or Modified Condition/Decision Coverage testing.

MCDC Testing

Modified Condition Decision Coverage

- Key idea: Test *important combinations* of conditions, avoiding exponential blowup
- A combination is "important" if each basic condition is shown to independently affect the outcome of each decision

MCDC Testing

Modified Condition Decision Coverage

- For each basic condition *C*, we need two test cases T_1 and T_2
- Values of all *evaluated* conditions except *C* are the same
- Compound condition as a whole evaluates to *True* for T_1 and *false* for T_2
- A good balance of thoroughness and test size (and therefore widely used)

MCDC Testing

Modified Condition Decision Coverage

- Assume `(((a ∨ b) ∧ c) ∨ d) ∧ e)`
- We need six tests to cover MCDC combinations

	a	b	c	d	e	Decision
(1)	True	–	True	–	True	True
(2)	False	True	True	–	True	True
(3)	True	–	False	True	True	True
(6)	True	–	True	–	False	False
(11)	True	–	False	False	–	False
(13)	False	False	–	False	–	False

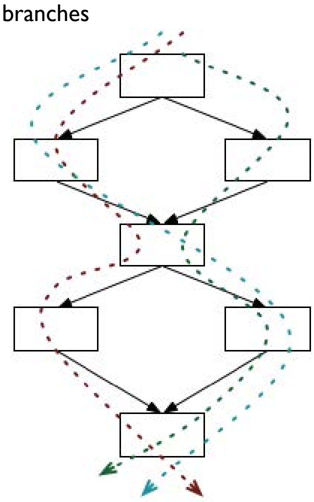
Underlined values independently affect the outcome of the decision. Note that the same test case can cover the values of several basic conditions. For example, test case (1) covers value True for the basic conditions a, c and e. Note also that this is not the only possible set of test cases to satisfy the criterion; a different selection of boolean combinations could be equally effective.

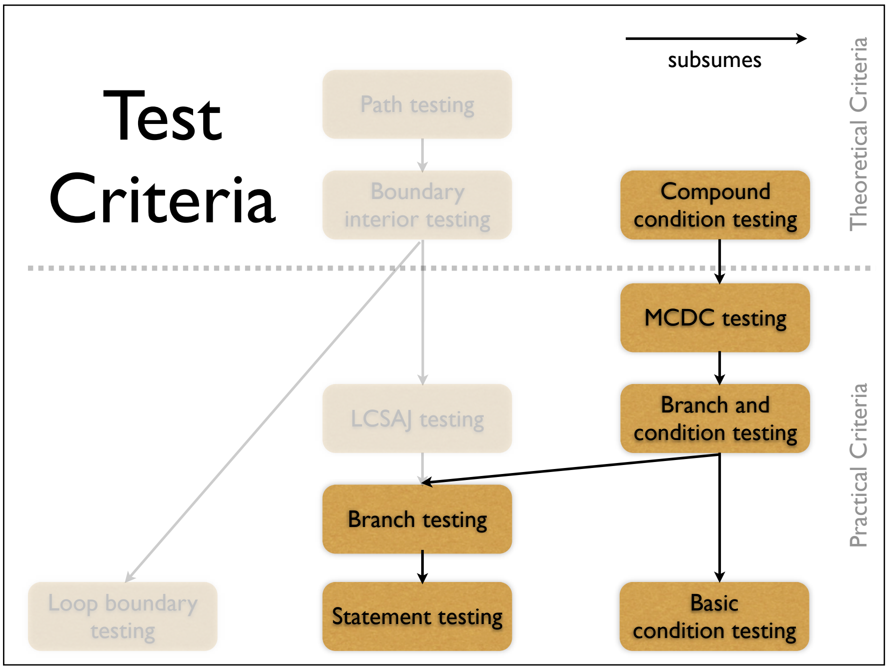
Path Testing

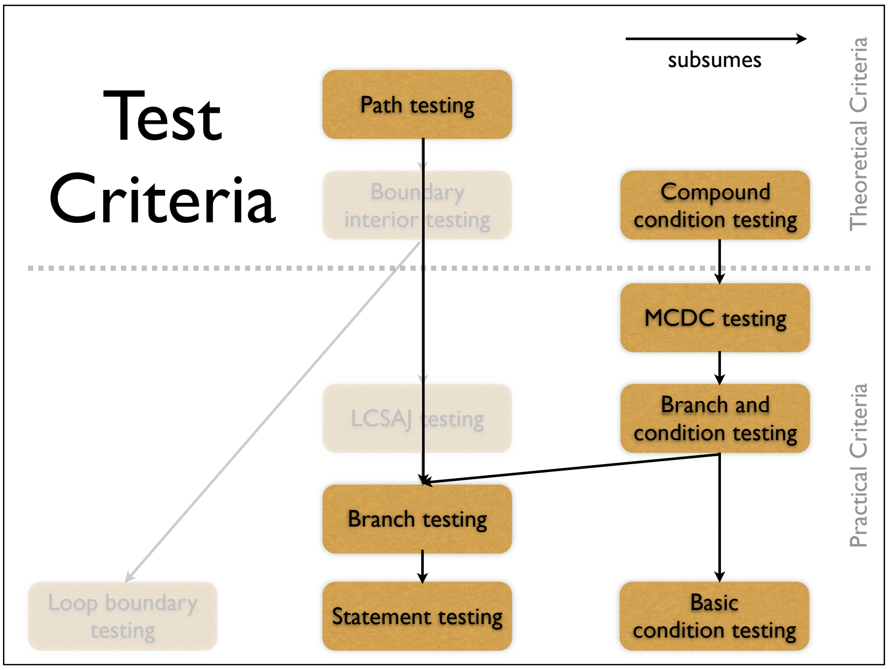
beyond individual branches

- Key idea: explore *sequences of branches* in control flow
- Many more paths than branches

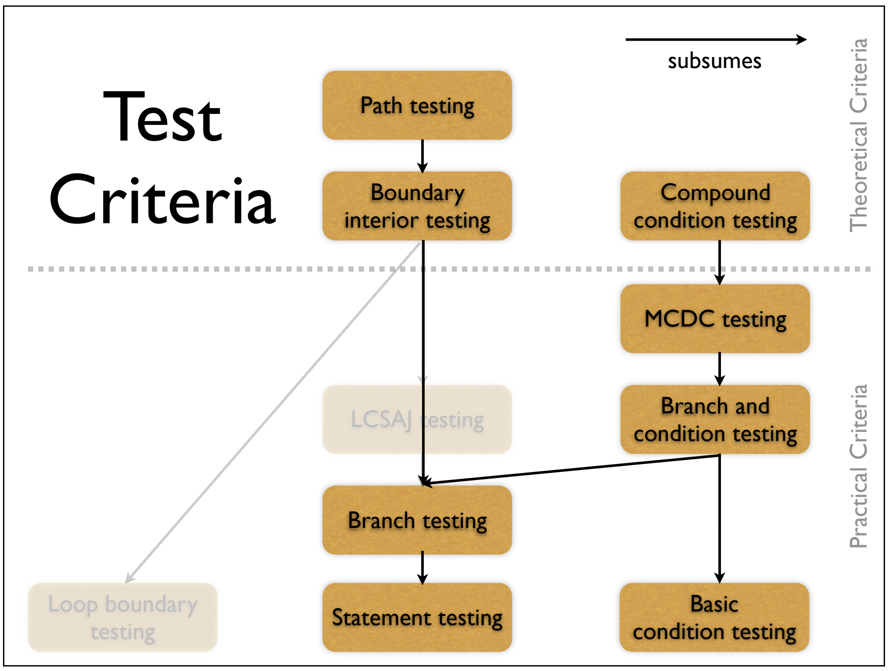
  calls for compromises







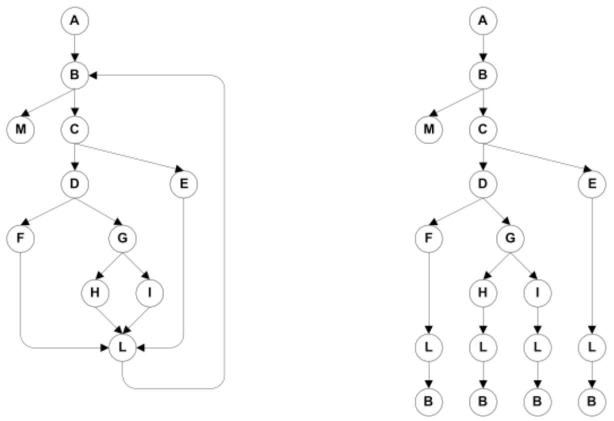
For one thing, there is general path testing, i.e. covering all paths in the program. Since loops are unbounded, this is generally not feasible and therefore just a theoretical criterion. Its advantage, though, is that it subsumes almost all criteria.



Boundary interior testing groups together paths that differ only in the subpath they follow when repeating the body of a loop. In other words, we follow each path in the CFG up to the first repeated node.

Boundary Interior Adequacy

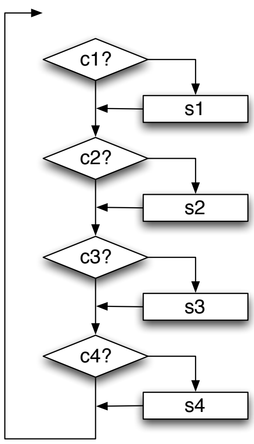
for cgi_decode



Original CFG   Paths to be covered

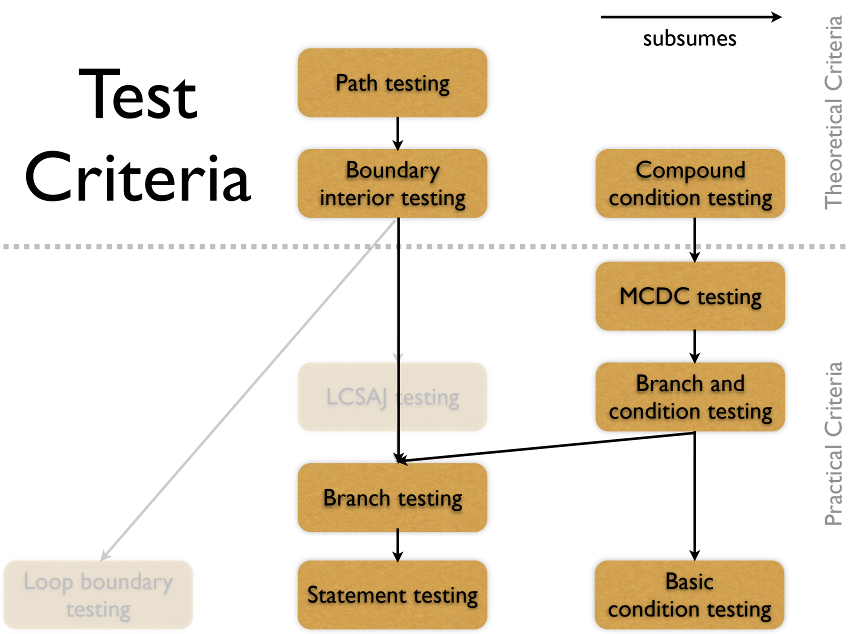
The graph at the right shows the paths that must be covered in the control flow graph at the left, using the “boundary interior” adequacy criiterion.

Issues



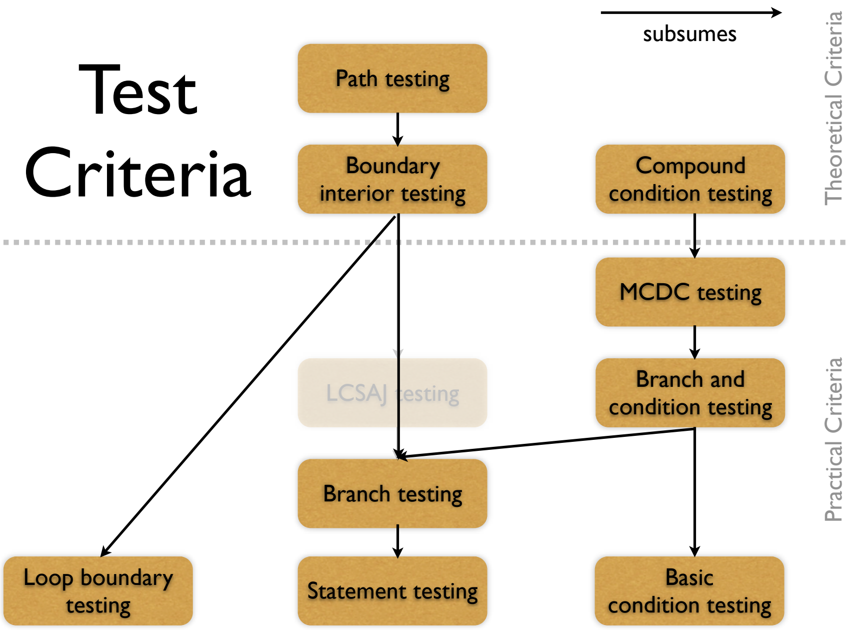
- The number of paths may still grow exponentially
  In this example, there are $2^4 = 16$ paths
- Forcing paths may be *infeasible*
  or even *impossible* if conditions are not independent

Test Criteria



Therefore, boundary interior testing belongs more to the “theoretical” criteria.

Test Criteria



Another alternative is loop boundary testing which forces constraints on how loops are to be executed. This is a practical criterion.

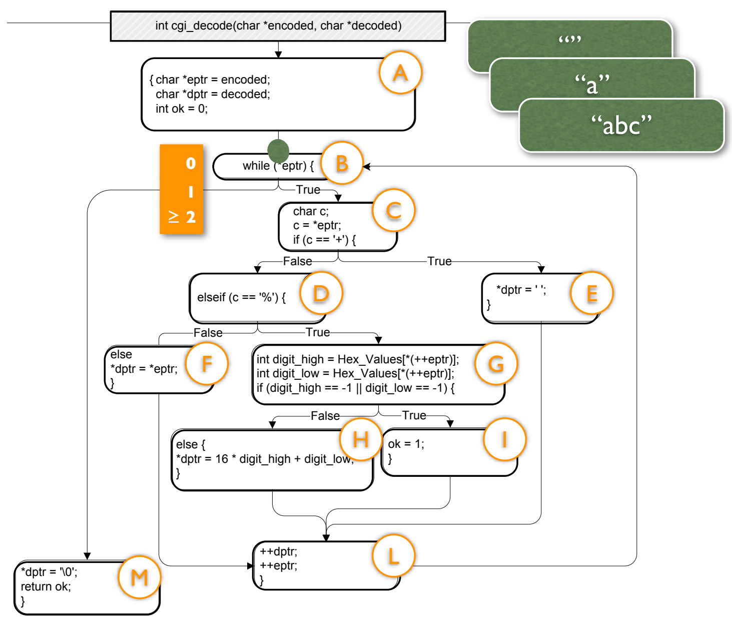
Loop Boundary Adequacy

A test suite satisfies the loop boundary adequacy criterion if for every loop *L*:

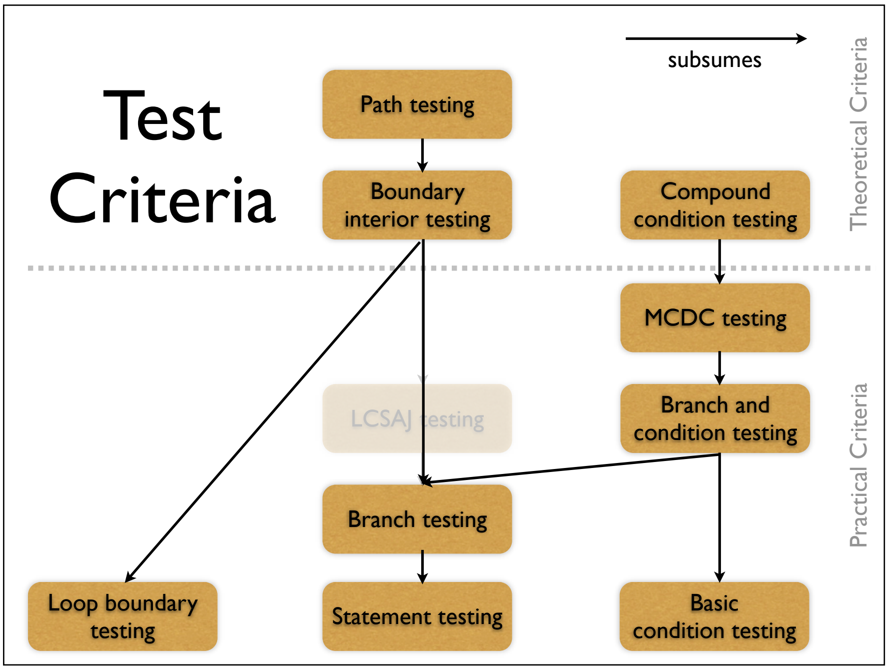
- *There is a test case which iterates L zero times*
- *There is a test case which iterates L once*
- *There is a test case which iterates L more than once*

Typically combined with other adequacy criteria such as MCDC

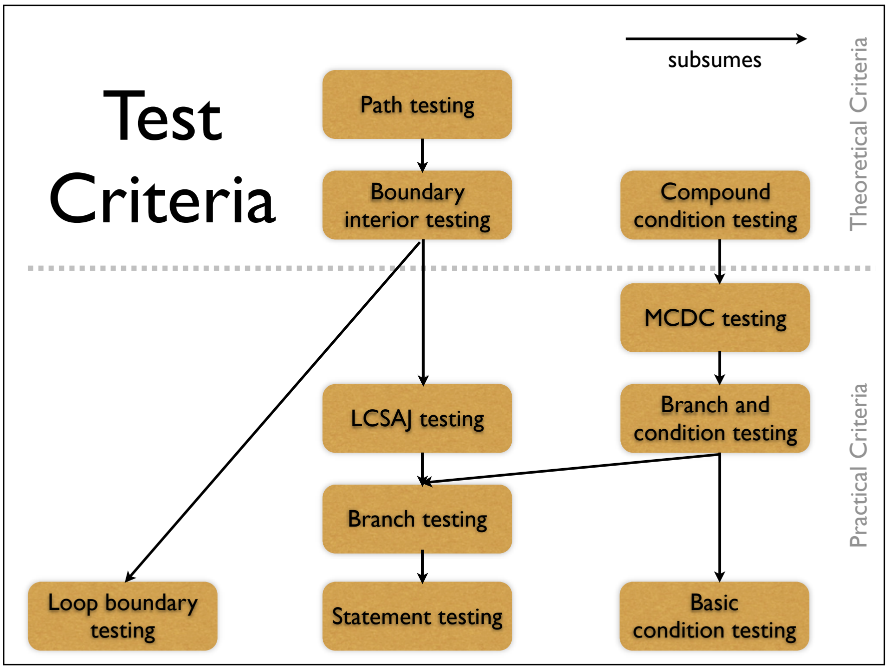
This is a variant of the boundary/interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths



With these three test cases, we obtain loop boundary adequacy for the cgi_decode main loop.



Another alternative is loop boundary testing which forces constraints on how loops are to be executed.



LCSAJ is a generalization over branch and statement coverage.

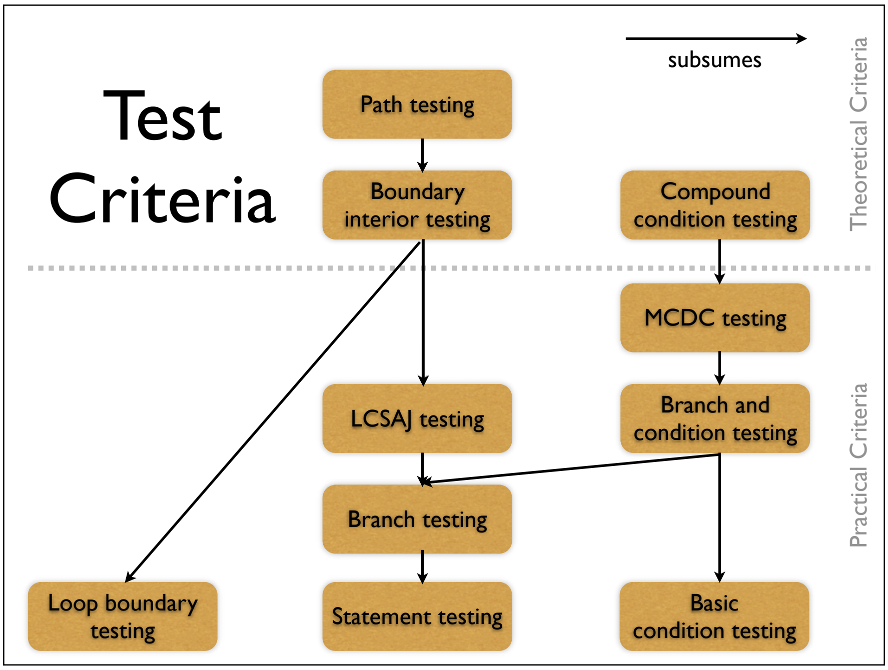
LCSAJ Adequacy

Testing all paths up to a fixed length

- LCSAJ = Linear Code Sequence And Jump
- A LCSAJ is a sequential subpath in the CFG starting and ending in a branch

LCSAJ length	corresponds to
1	statement coverage
2	branch coverage
n	coverage of *n* consecutive LCSAJs
∞	path coverage

Considering the exponential blowup in sequences of conditional statements (even when not in loops), we might choose to consider only sub-sequences of a given length. This is what LCSAJ gives us --- essentially considering full path coverage of (short) sequences of decisions.



And this is the summary of structural testing techniques.

Weyuker's Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

Established by a number of studies done by E. Weyuker at AT&T. "Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case".

Satisfying Criteria

Sometimes criteria may not be satisfiable:

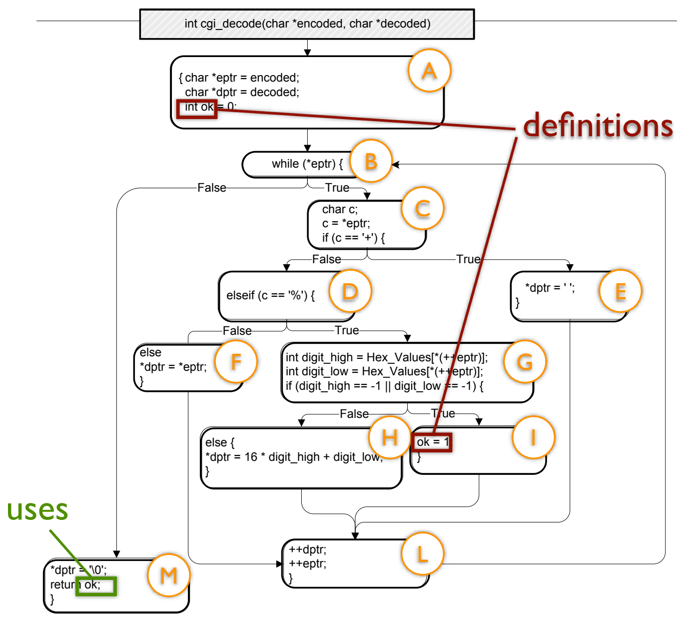
- *Statements* may not be executed because of *defensive programming* or *code reuse*
- *Conditions* may not be satisfiable because of *interdependent conditions*
- *Paths* may not be executable because of *interdependent decisions*

Satisfying Criteria

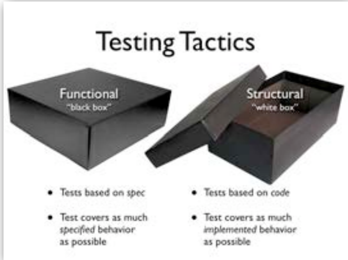
- Reaching specific code can be very hard!
- Even the best-designed, best-maintained systems may contain unreachable code
- A large amount of unreachable code/paths/conditions is a serious *maintainability problem*
- Solutions: allow coverage less than 100%, or require justification for exceptions

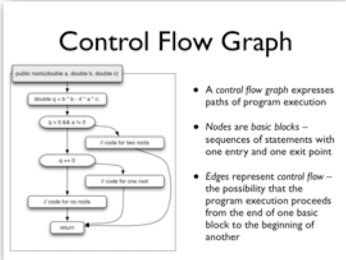
More Testing Criteria

- Object-oriented testing
  e,g,"Every transition in the object's FSM must be covered" or "Every method pair in the object's FSM must be covered"
- Interclass testing
  e.g,"Every interaction between two objects must be covered"
- Data flow testing
  e.g.,"Every definition-use pair of a variable must be covered"

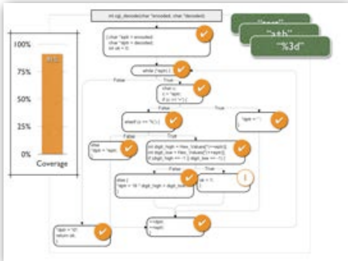


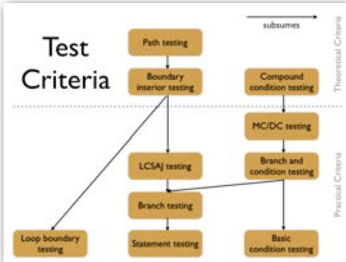
Data flow testing is based on the observation that computing the wrong value leads to a failure only when that value is subsequently used.
A typical data flow testing criterion is therefore that the tests must exercise every pair (definition, uses) of a variable (such as "ok" in this example).


Testing Tactics
Functional
"black box"
Structural
"white box"
• Tests based on spec
• Test covers as much specified behavior as possible
• Tests based on code
• Test covers as much implemented behavior as possible


Control Flow Graph
• A control flow graph expresses paths of program execution
• Nodes are basic blocks – sequences of statements with one entry and one exit point
• Edges represent control flow – the possibility that the program execution proceeds from the end of one basic block to the beginning of another

Summary


100%
75%
50%
25%
0%
Coverage
"%3d"


Test Criteria
subsumes
Path testing
Boundary interior testing
Compound condition testing
MC/DC testing
LCSAJ testing
Branch and condition testing
Branch testing
Loop boundary testing
Statement testing
Basic condition testing
Theoretical Criteria
Practical Criteria