Topics in Software Dynamic White-box Testing
Part 1: Control-flow Testing

[Reading assignment: Chapter 7, pp. 105-122 plus many things in slides that are not in the book …]

Control-Flow Testing

- **Control-flow testing** is a structural testing strategy that uses the program's control flow as a model.
- Control-flow testing techniques are based on judiciously selecting a set of test paths through the program.
- The set of paths chosen is used to achieve a certain measure of testing thoroughness.
  - *E.g.,* pick enough paths to assure that every source statement is executed as least once.

Motivation

- Control-flow testing is most applicable to new software for unit testing.
- Control-flow testing assumptions:
  - specifications are correct
  - data is defined and accessed properly
  - there are no bugs other than those that affect control flow
- Structured and OO languages reduce the number of control-flow bugs.

Control Flowgraphs

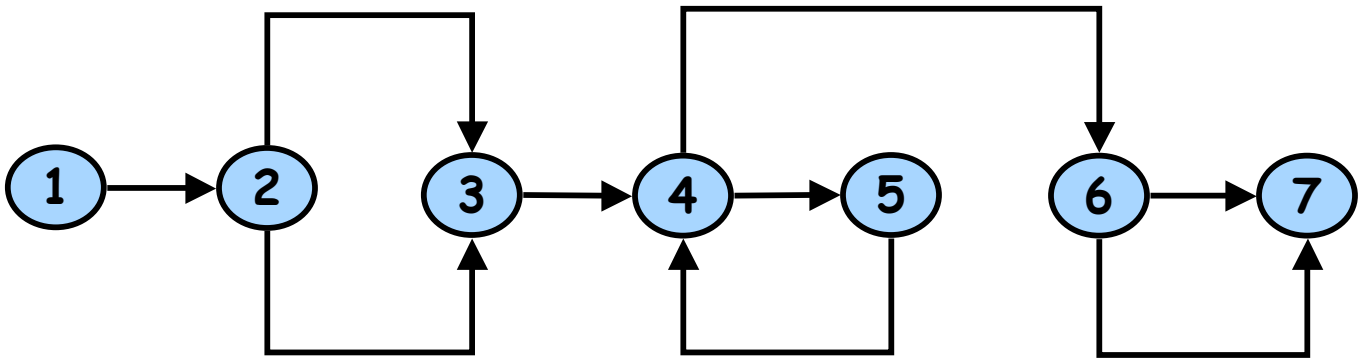
- The control flowgraph is a graphical representation of a program's control structure.

Flowgraphs Consist of Three Primitives

- A **decision** is a program point at which the control can diverge.
  - (*e.g.,* `if` and `case` statements).
- A **junction** is a program point where the control flow can merge.
  - (*e.g.,* `end if`, `end loop`, `goto label`)
- A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.,* straight-line code).
  - A process has one entry and one exit.
  - A program does not jump into or out of a process.

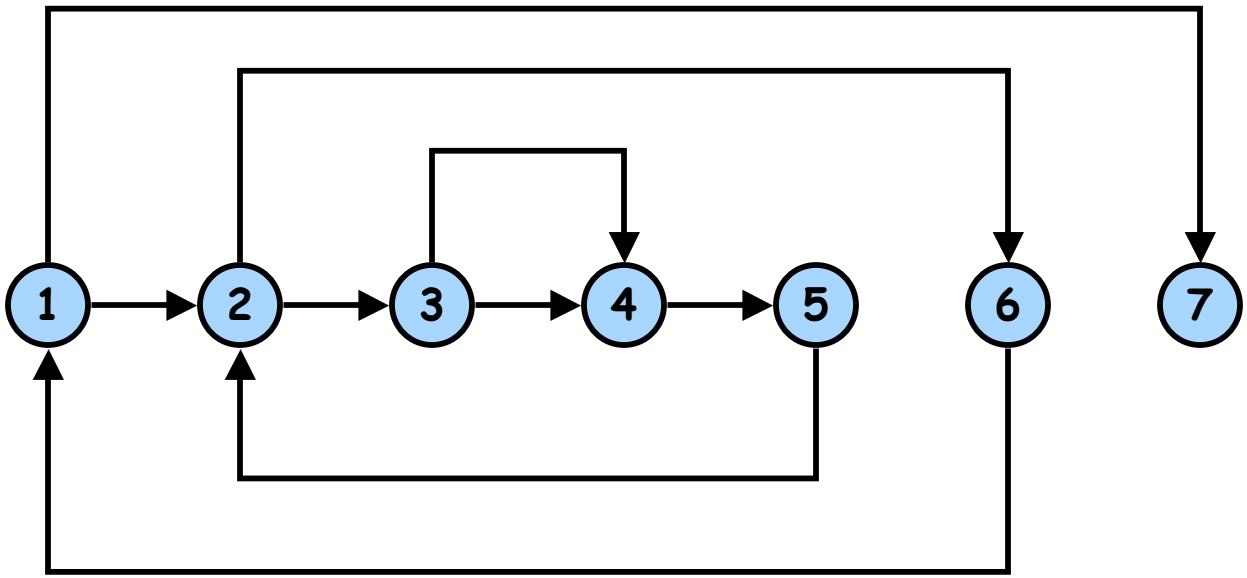
Exponentiation Algorithm

```
1   scanf("%d %d",&x, &y);
2   if (y < 0)
        pow = -y;
    else
        pow = y;
3   z = 1.0;
4   while (pow != 0) {
        z = z * x;
        pow = pow - 1;
5       }
6   if (y < 0)
        z = 1.0 / z;
7   printf ("%f",z);
```



Bubble Sort Algorithm

```
1  for (j=1; j<N; j++)  {
       last = N - j + 1;
2       for (k=1; k<last; k++)  {
3             if (list[k] > list[k+1])  {
                  temp = list[k];
                  list[k] = list[k+1];
                  list[k+1] = temp;
4             }
5      }
6 }
7 print("Done\n");
```



Paths

- A **path** through a program is a sequence of statements that starts at an entry, junction, or decision and ends at another (possible the same), junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Paths consist of **segments**.
- The smallest segment is a link. A **link** is a single process that lies between 2 nodes.

Paths (Cont'd)

- The **length** of a path is the number of links in a path.
- An **entry/exit path** or a **complete path** is a path that starts at a routine's entry and ends at the same routine's exit.

Paths (Cont'd)

- Complete paths are useful for testing because:
  - It is difficult to set up and execute paths that start at an arbitrary statement.
  - It is difficult to stop at an arbitrary statement without changing the code being tested.
  - We think of routines as input/output paths.

Path Selection Criteria

- There are many paths between the entry and exit points of a typical routine.
- Even a small routine can have a large number of paths.

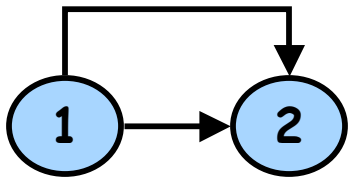
How do we define "complete" testing?

- 1) Exercise every path from entry to exit.
- 2) Exercise every statement at least once.
- 3) Exercise every branch (in each direction) at least once.
- Clearly, 1 implies 2 and 3
- However, 1 is impractical for most routines.
- Also, 2 is not equal to 3 in languages with `goto` statements.

Demonstration that 2 does not imply 3

Correct Code

```
1   if (x >= 0 ) {
      x = x + A;
      }
2   x = x + A
```



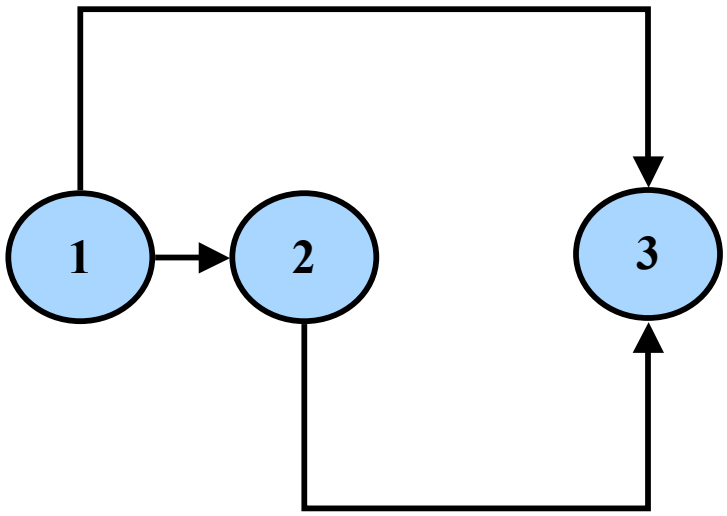
Buggy Code

```
1   if (x >= 0 ) {
    /* missing statement */
      }
2   x = x + A
```

- **2.Statement Coverage:**
  For x < 0 the program produces the correct result AND every statement has been executed.
- **3.Branch Coverage:**
  Would have found the bug! Therefore 2 does not imply 3.

Demonstration that 3 Does not Imply 2

```
1          if (x < 0) {
2              goto 200;
               x = x + A
           } else
               x = x + A
3  200:  x = x + A
```



- **<u>Branch Coverage:</u>** Does not exercise dead code. Therefore 3 does not imply 2.
- However, 3 implies 2 for programs written in a structured programming language without `goto` statements.

Control-flow Testing Criteria

- We have explored 3 testing criteria from an infinite set of strategies:
  - 1) Path Testing (P_∞):
    - 100% path coverage.
    - Execute all possible control flow paths through the program.

Control-flow Testing Criteria (Cont'd)

- 2) Statement Testing (P_1):
  - 100% statement coverage.
  - Execute all statements in a program at least once under some test.
- 3) Branch Testing (P_2):
  - 100% branch coverage.
  - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

- $$P_1 \leq P_2 \leq \ldots \leq P_\infty$$

Common Sense Strategies

- **Statement** and **branch** coverage have been used for over two decades as a minimum mandatory unit test requirement for new code developed at IBM and other companies.
- Insisting on statement and branch coverage is based on common sense rather than theory.

Common Sense Strategies (Cont'd)

- It makes sense to use **branch** coverage because software has a high density of conditional branches, loop, etc. (25% in most PLs)
- It is better to leave out untested code than to include it in a product release.

Quote

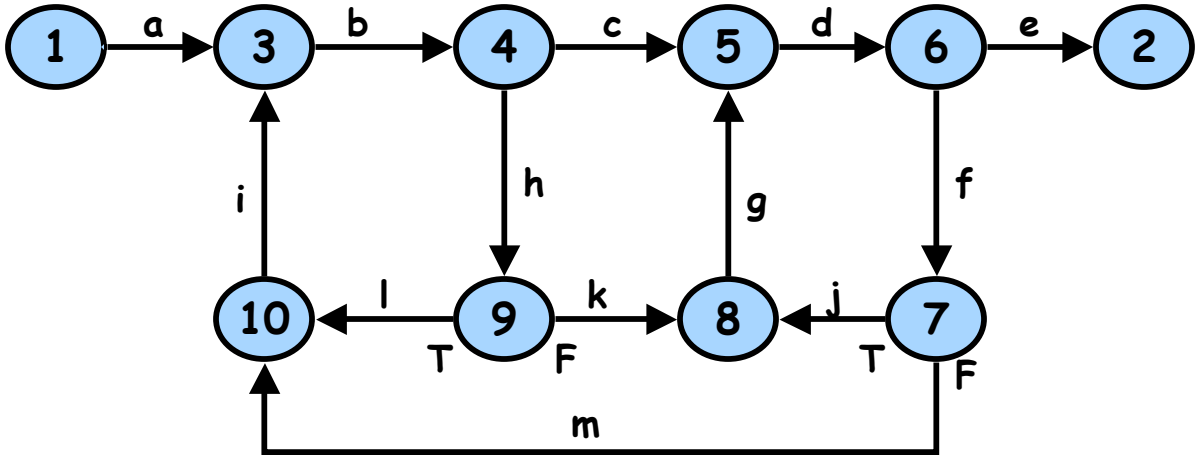
"The more we learn about testing, the more we realize that statement and branch coverage are minimum floors below which we dare not fall, rather that ceilings to which we should aspire."

- B. Beizer.

Which Paths?

- You must pick enough paths to achieve statement and branch coverage.
- **Question:** What is the fewest number of paths to achieve statement and branch coverage?
- **Answer:** Unask the question.
  - It is better to take many simple paths than a few complicated ones.
  - There is no harm in taking paths that will exercise the same code more than once.

Example of P1 and P2 Coverage



PATHS	DECISIONS				PROCESS LINKS											
	4	6	7	9	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*
abcde	*T*	*T*			*	*	*	*	*								
abhkgde	*F*	*T*		*F*	*	*		*	*		*	*			*		
abhlibcde	*TF*	*T*		*T*	*	*	*	*	*		*	*	*			*	
abcdfjgde	*T*	*TF*	*T*		*	*	*	*	*	*	*			*			
abcdfmibcde	*T*	*TF*	*F*		*	*	*	*	*	*		*	*				*

Branch and Statement Coverage

- **Question:** Does every decision have a *T* (true) and a *F* (false) in its column?
- **Answer:** Yes implies branch coverage.
- **Question:** Is every link covered at least once?
- **Answer:** Yes implies statement coverage.

Guidelines

- Select paths as small variations of previous paths.
- Try to change one thing in each path at a time.

Effectiveness of Control-flow Testing

- About 65% of all bugs can be caught in unit testing.
- Unit testing is dominated by control-flow testing methods.
- Statement and branch testing dominates control-flow testing.

Effectiveness of Control-flow Testing (Cont'd)

- Studies show that control-flow testing catches 50% of all bugs caught during unit testing.
  - About 33% of all bugs.
- Control-flow testing is more effective for unstructured code than for code that follows structured programming.
- Experienced programmers can bypass drawing flowgraphs by doing path selection on the source.

Limitations of Control-flow Testing

- Control-flow testing as a sole testing technique is limited:
  - Interface mismatches and mistakes are not caught.
  - Not all initialization mistakes are caught by control-flow testing.
  - Specification mistakes are not caught.

Test Outcomes

- The **outcome** of test is what we expect to happen as a result of the test.
- Test outcomes include anything we can observe in the computer's memory that should have (not) changed as a result of the test.
- Since we are not "kiddie testing" we must predict the outcome of the test as part of the test design process.

Testing Process

- run the test
- observe the actual outcome
- compare the actual outcome to the expected outcome.

Questions About Test Outcomes

- **Question:** If the predicted and actual outcomes match, can we say that the test has been passed?
- **Answer:** No! The desired outcome could have been achieved for the wrong reason. (coincidental correctness)

Questions About Test Outcomes

- **Question:** Assume that we ran a covering set of tests and achieved the desired outcomes for each case. Can we say that we've covered all branches?
- **Answer:** No! The desired outcome could have been reached by the wrong path!
  - Path instrumentation is necessary to confirm that the outcome was achieved by the intended path.

Path Instrumentation

- All instrumentation methods are a variation on a theme of an interpretive trace.
- An **interpretive trace program** executes every statement in order and records:
  - the intermediate values of all calculations
  - the statement labels traversed
  - ...

Path Instrumentation (Cont'd)

- If we run the tested routine under a trace, then we have all the information we need to confirm:
  - the outcome of the test
  - whether the outcome was achieved by the intended path.

Two Detailed Examples Of Control-flow Testing

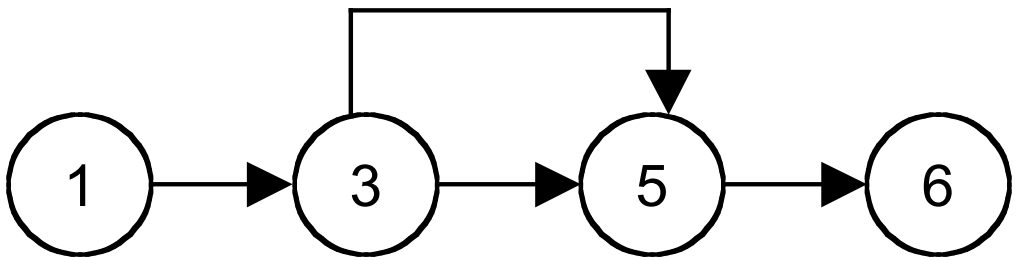
Using Control-flow Testing to Test Function ABS

- Consider the following function:

```
/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
1          int ABS(int x)
2          {
3          if (x < 0)
4              x = -x;
5          return x;
6          }
```

The Flowgraph for ABS

```
/* ABS
   This program function returns the absolute value of the integer
   passed to the function as a parameter.
    INPUT: An integer.
    OUTPUT: The absolute value if the input integer.
*/
1           int ABS(int x)
2           {
3           if (x < 0)
4               x = -x;
5           return x;
6           }
```

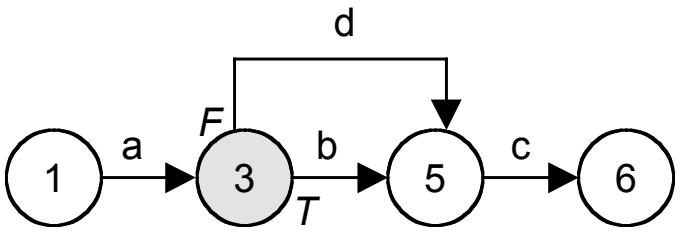


Test Cases to Satisfy **Path** Coverage for ABS

- Complete path testing of *ABS* is theoretically possible but not practical.
- *ABS* takes as its input any integer. There are many integers (depending on the maximum size of an integer for the language) that could be input to *ABS* making it impractical to test all possible inputs to ABS.

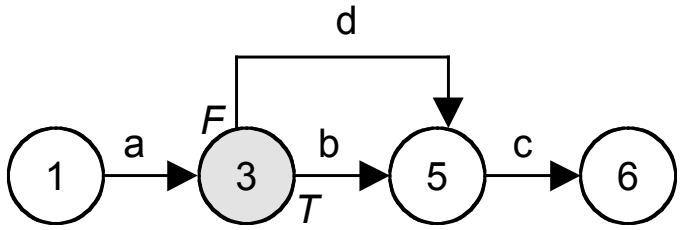
Test Cases to Satisfy **Statement Testing** Coverage for ABS

PATHS	PROCESS LINKS				TEST CASES	
	a	**b**	**c**	**d**	**INPUT**	**OUTPUT**
abc	√	√	√		A Negative Integer, x	-x
adc	√		√	√	A Positive Integer, x	x



Test Cases to Satisfy **Branch Testing** Coverage for ABS

PATHS	DECISIONS	TEST CASES	
		INPUT	**OUTPUT**
abc	T	A Negative Integer, x	-x
adc	F	A Positive Integer, x	x



Example: Using Control-flow Testing to Test Program COUNT

- Consider the following program:

```
/* COUNT
   This program counts the number of characters and lines in a text file
    INPUT: Text File
    OUTPUT: Number of characters and number of lines.
*/
1          main(int argc, char *argv[])
2          {
3          int numChars = 0;
4          int numLines = 0;
5          char chr;
6          FILE *fp = NULL;
7
```

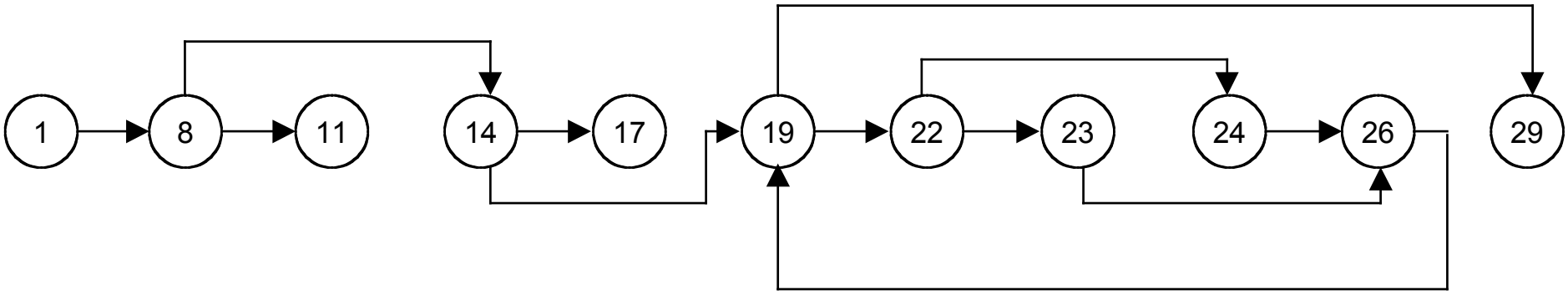
Program COUNT (Cont'd)

```
8          if (argc < 2)
9             {
10            printf("\nUsage: %s <filename>", argv[0]);
11            return (-1);
12            }
13         fp = fopen(argv[1], "r");
14         if (fp == NULL)
15            {
16            perror(argv[1]);      /* display error message */
17            return (-2);
18            }
```

Program COUNT (Cont'd)

```
19          while (!feof(fp))
20            {
21            chr = getc(fp);        /* read character */
22            if (chr == '\n')       /* if carriage return */
23                ++numLines;
24            else
25                ++numChars;
26            }
27        printf("\nNumber of characters = %d",
numChars);
28        printf("\nNumber of lines = %d", numLines);
29        }
```

The Flowgraph for COUNT

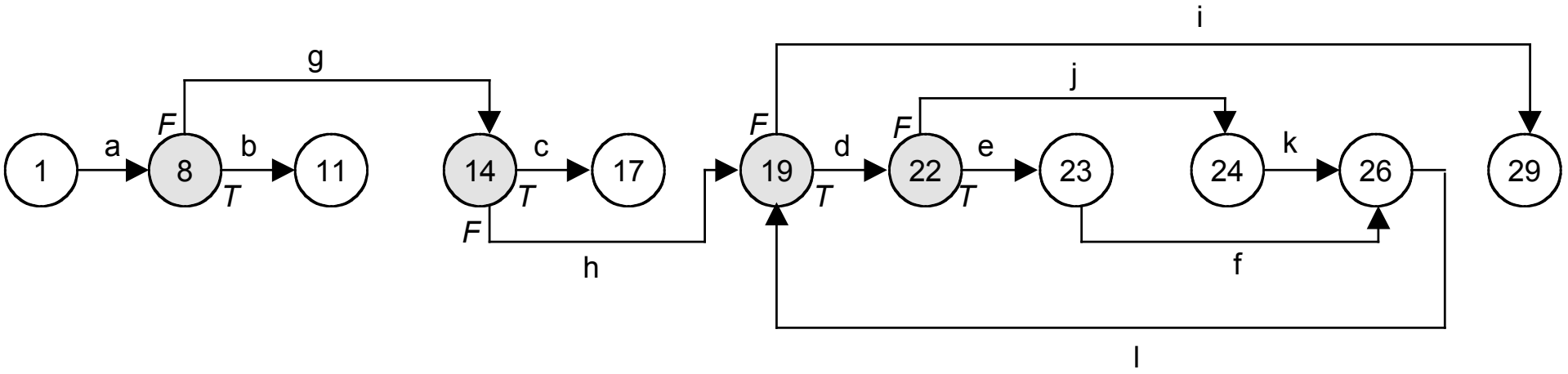


- The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.

Test Cases to Satisfy **Path** Coverage for COUNT

- Complete path testing of *COUNT* is impossible because there are an infinite number of distinct text files that may be used as inputs to *COUNT*.

Test Cases to Satisfy **Statement Testing** Coverage for COUNT



Test Cases to Satisfy **Statement Testing** Coverage for COUNT

PATHS	PROCESS LINKS												TEST CASES	
	a	b	c	d	e	f	g	h	i	j	k	l	INPUT	OUTPUT
ab	√	√											None	"Usage: COUNT <filename>"
agc	√		√				√						Invalid Input Filename	Error Message
aghdj kli	√			√			√	√	√	√	√	√	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghd efli	√			√	√	√	√	√	√			√	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Test Cases to Satisfy **Branch Testing** Coverage for COUNT

PATHS	DECISIONS				TEST CASES	
	8	14	19	22	INPUT	OUTPUT
ab	T				None	"Usage: COUNT <filename>"
agc	F	T			Invalid Input Filename	Error Message
aghdjkli	F	F	T, F	F	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghdefli	F	F	T, F	T	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

Summary

- The object of control-flow testing is to execute enough tests to assure that statement and branch coverage has been achieved.
- Select paths as deviation from the normal paths. Add paths as needed to achieve coverage.
- Use instrumentation (manual or using tools) to verify paths.
- Document all tests and expected test results.
- A test that reveals a bug has succeeded, not failed.

You now know …

- … control flow testing
- … statement coverage
- … branch coverage