COMP 3111
INTRODUCTION TO SOFTWARE ENGINEERING

Testing

Real Programmers need no Testing!

The TOP Five List

5) I want to get this done fast, testing is going to slow me down.
4) I started programming when I was 2. Don't insult me by testing my perfect code!
3) Testing is for incompetent programmers who cannot hack.
2) We are not HKU students, our code actually works!
1) "Most of the functions in Graph.java, as implemented, are one or two line functions that rely solely upon functions in HashMap or HashSet. I am assuming that these functions work perfectly, and thus there is really no need to test them."
– an excerpt from a student's e-mail

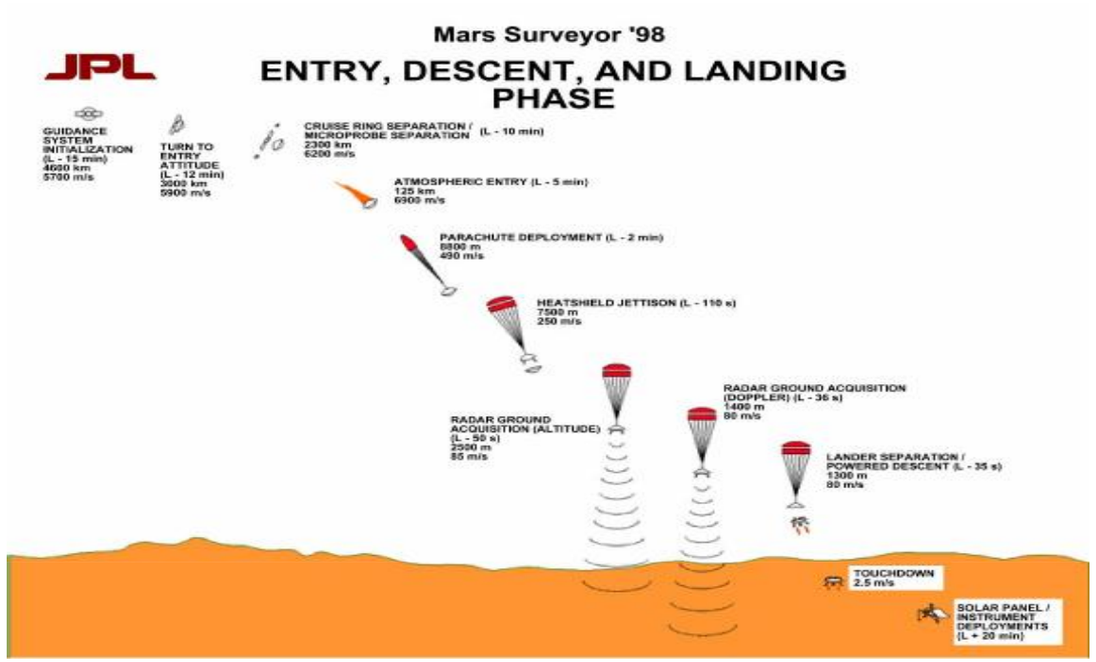
http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

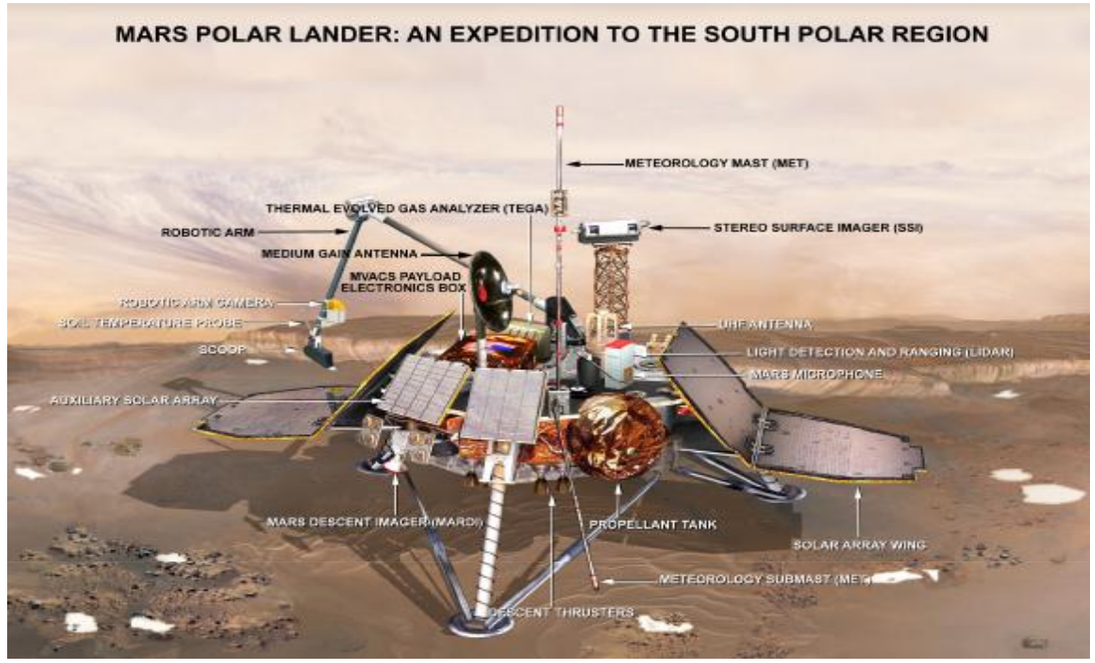
Ariane 5 rocket

- The rocket self-destructed 37 seconds after launch
- Reason: A control software bug that went undetected
  - Conversion from 64-bit floating point to 16-bit signed integer value had caused an exception
    - The floating point number was larger than 32767 (max 16-bit signed integer)
  - Efficiency considerations had led to the disabling of the exception handler.
  - Program crashed → rocket crashed
- Total Cost: over $1 billion

http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

Mars Polar Lander





- Sensor signal falsely indicated that the craft had touched down when it was 130-feet above the surface.
  - Then the descent engines to shut down prematurely
- The error was traced to a single bad line of software code.
- NASA investigation panels blame for the landers failure, "are well known as difficult parts of the software-engineering process,“

http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

Testing is for *every* system

Examples showed particularly costly errors

But every little error adds up

Insufficient software testing costs $22-60 billion per year in the U.S. [NIST Planning Report 02-3, 2002]

If your software is worth writing, it's worth writing right

http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

The Phases of Testing

Unit Testing

Is each module does what it suppose to do?

Integration Testing

Do you get the expected results when the parts are put together?

Validation Testing

Does the program satisfy the requirements

System Testing

Does it work within the overall system

http://www.cs.washington.edu/education/courses/403/09sp/calendar403.html.

THE REALITY OF TESTING

☞ **Testing *cannot* show the absence of software errors;**
it can only show that software errors are present!

☞ **It is impossible to completely test a nontrivial system.**
(Most systems will have bugs in them that either users uncover or that are *never uncovered*!)

☞ **Testing is not a decidable activity.**
(We cannot claim a system has no bugs just because all tests succeed.)

THE REALITY OF TESTING

☞ **Testing is a destructive activity.**
(We try to make the software fail.)

☞ **Thus, it is often difficult for software engineers to effectively test their own software.**
(They have no incentive to make their software fail!)

☞ **Testing is not a job for novices.**

☞ **Good testing has the opposite nature of good software engineering**
(i.e., good software engineering → try to ensure that the software does not fail
good testing → try to make the software fail).

IS EXHAUSTIVE TESTING FEASIBLE?

- Who needs a testing plan.
- Why not just try it with all inputs and see if it works?

```
int proc1 (int x, int y, int z)
    // requires: 1 ≤ x,y,z ≤ 1000
    // effects: computes some f(x,y,z)
```

How many runs are needed to *exhaustively* test this program?

IS EXHAUSTIVE TESTING FEASIBLE?

- Who needs a testing plan.
- Why not just try it with all inputs and see if it works?

```
int proc1 (int x, int y, int z)
    // requires: 1 ≤ x,y,z ≤ 1000
    // effects: computes some f(x,y,z)
```

How many runs are needed to *exhaustively* test this program?

Conclusion

It is imperative to have a plan for testing if we want to uncover as many defects as possible in the shortest possible time.

PLAN TESTS

Goal: To design a set of tests that has the **highest likelihood** of uncovering defects with the **minimum amount of time and effort.**

WHY? Tests cost money and take time to develop, perform and evaluate (up to 40% of project effort is often devoted to testing).

A test plan specifies:

1. A testing strategy → the criteria and goals of testing.
   - What kinds of tests to perform and how to perform them.
   - The required level of test and code coverage.
   - The percentage of tests that should execute with a specific result.
2. An estimate of resources required: human/system.
3. A schedule for the testing: when to run which tests.

DESIGN TESTS: TEST CASE

A *test case* is one way of testing the system.

Specifies: *what* to test, under what *conditions*, *how* to test and *expected result*.

White Box Tests:

Verify component logic based on data/control structures.

Test cases based on *knowledge of the internal workings* of a component.

➨ **Availability of source code is required.**

Black Box Tests:

Verify component functionality based on the inputs and outputs.

Test cases based on *knowledge of specified functionality* of a component.

➨ **Availability of source code is not required!**

Regression Tests: *selective* White Box and Black Box re-testing to ensure that no new defects are introduced after a change is made.

DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: To design *selective tests* to ensure that we have executed/ exercised all:

1. independent paths in the code at least once.

   ☞ **Basis Path Testing**

2. logical decisions on their true and false sides.

   ☞ **Condition Testing**

3. loops at their boundaries and within their bounds.

   ☞ **Loop Testing**

4. internal data structures to ensure their validity.

   ☞ **Data Flow Testing**

WHITE BOX TESTING: BASIS PATH TESTING

Goal: To exercise each independent path at least once.

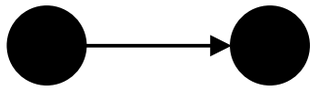
Overview

1. From the code, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the flow graph.
3. Determine a basis set of linearly independent paths based on the cyclomatic complexity.
4. Prepare test cases that force the execution of each path in the basis set.

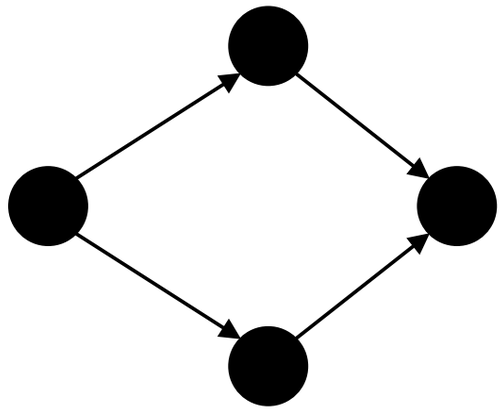
WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

1. From the code, draw a corresponding flow graph.

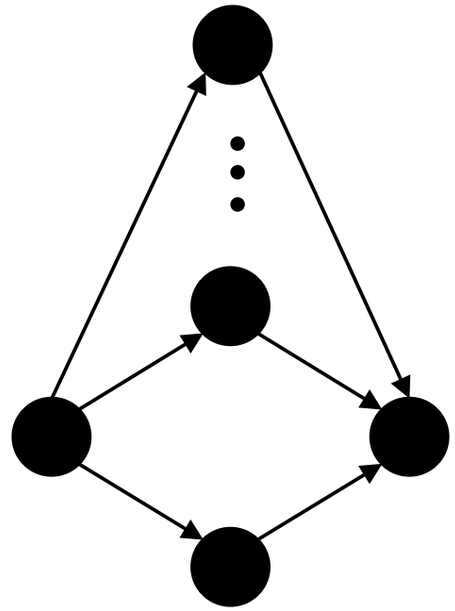
– Each circle represents one or more non-branching source code statements.



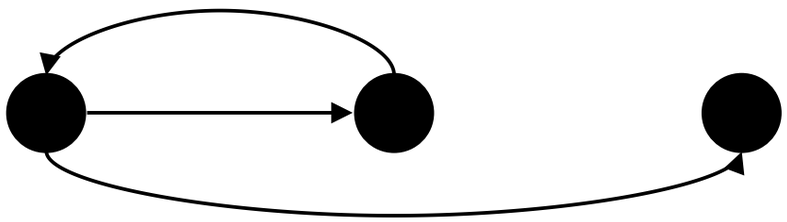
Sequence



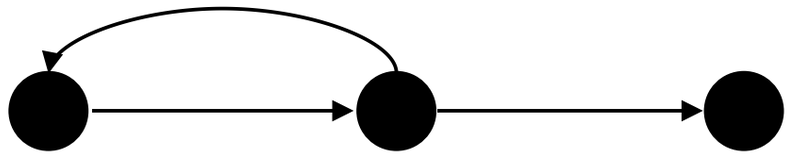
If-then-else



Case



Do-while



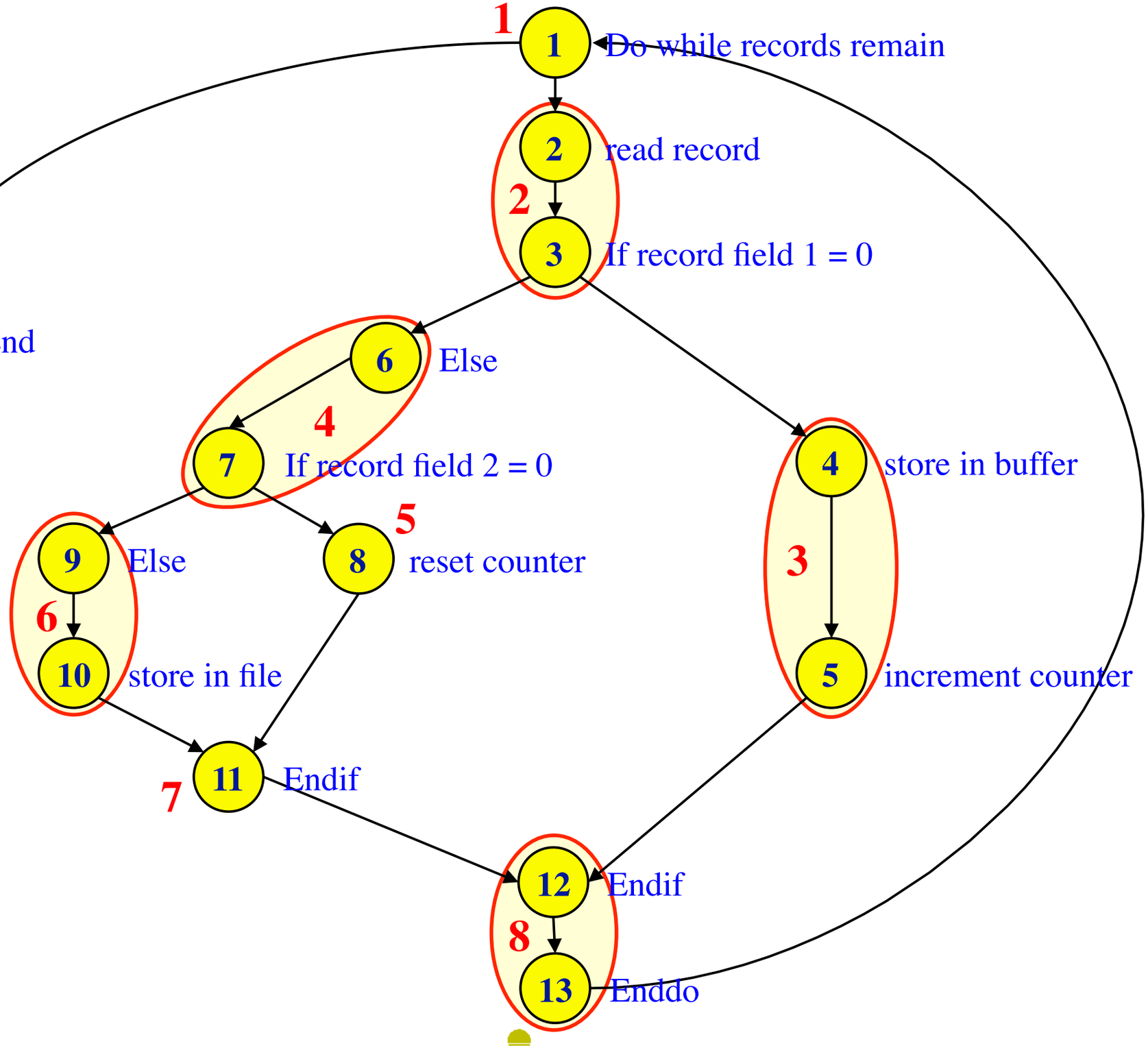
Do-until

EXAMPLE BASIS PATH TESTING: FLOW GRAPH

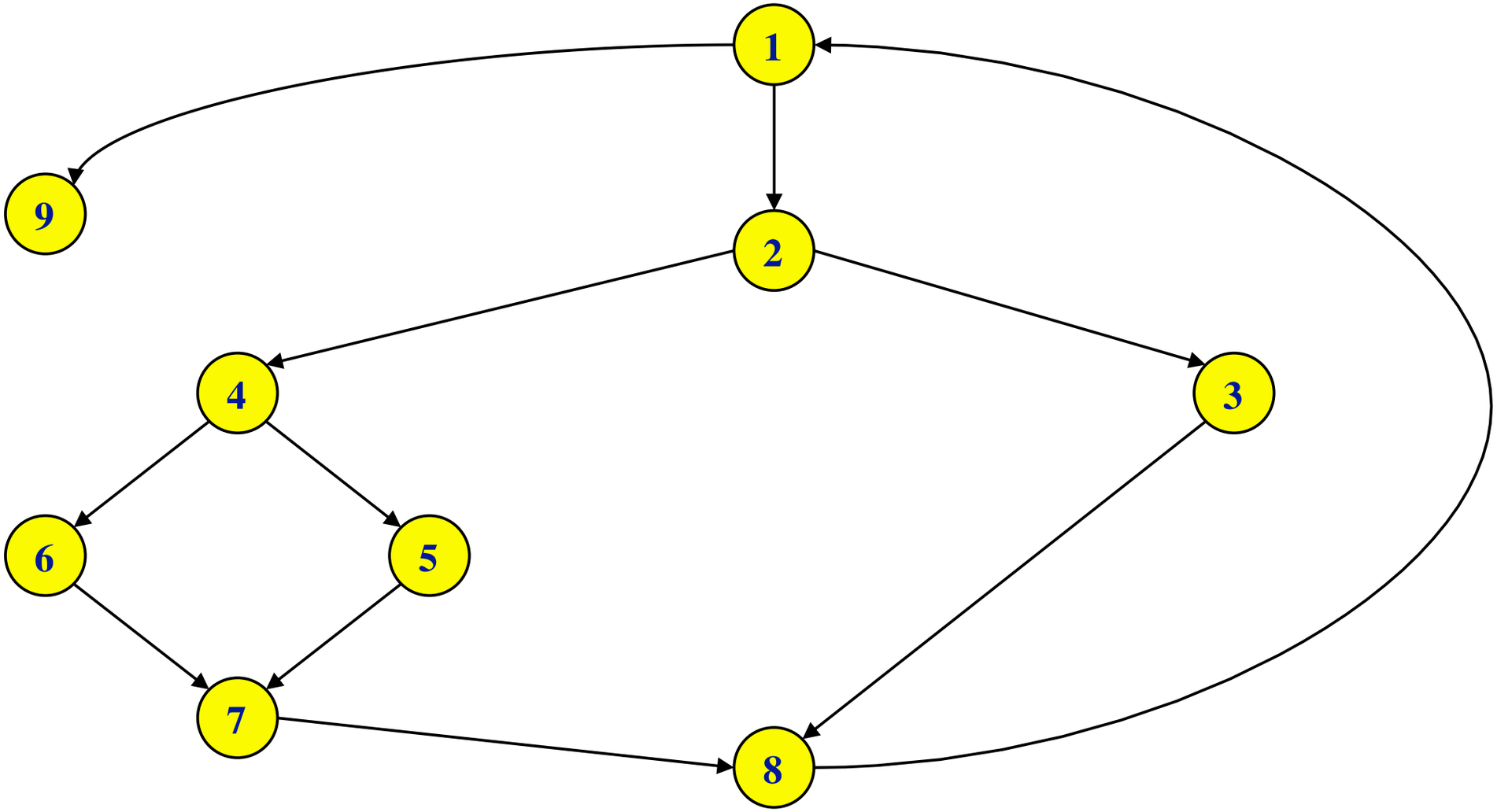
```
      Procedure: process records
1.          Do while records remain
2.                read record
3.                If record field 1 = 0
4.                      store in buffer
5.                      increment counter
6.                Else
7.                      If record field 2 = 0
8.                            reset counter
9.                      Else
10.                           store in file
11.                     Endif
12.               Endif
13.         Enddo
14.   End
```

EXAMPLE BASIS PATH TESTING: FLOW GRAPH

```
Procedure: process records
1.      Do while records remain
2.          read record
3.          If record field 1 = 0
4.              store in buffer
5.              increment counter
6.          Else
7.              If record field 2 = 0
8.                  reset counter
9.              Else
10.                 store in file
11.             Endif
12.         Endif
13.     Enddo
14. End
```



EXAMPLE BASIS PATH TESTING: FLOW GRAPH



Flow graph node to program statement mapping:

Node	Statement	Node	Statement	Node	Statement
1.	1	4.	6, 7	7.	11
2.	2, 3	5.	8	8.	12, 13
3.	4, 5	6.	9, 10	9.	14

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

2. Determine the cyclomatic complexity of the flow graph.

cyclomatic complexity: A quantitative measure of the logical complexity of the code.

Cyclomatic complexity provides an upper bound on the number of paths that need to be tested in the code.

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

2. Determine the cyclomatic complexity of the flow graph.

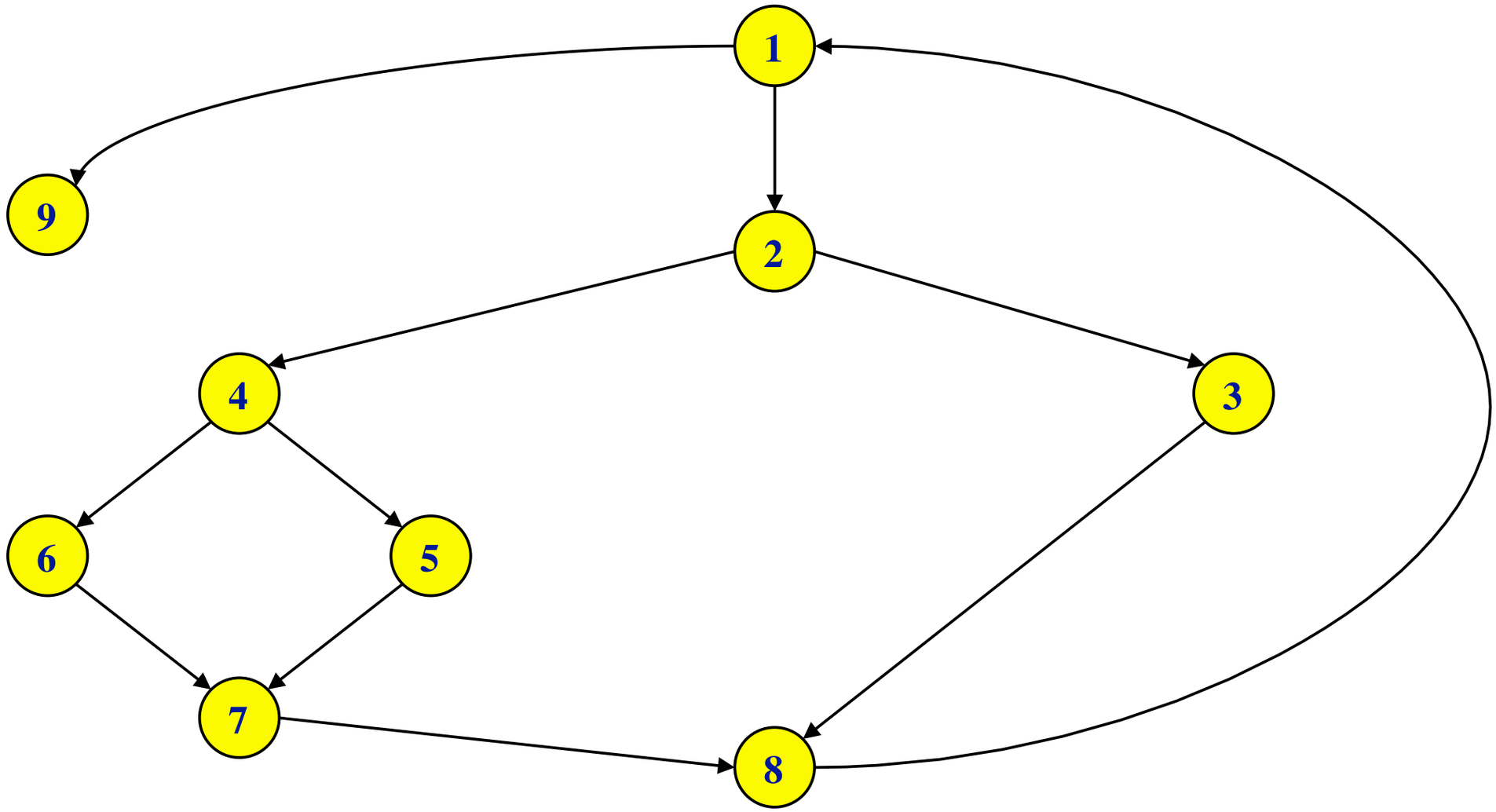
cyclomatic complexity: A quantitative measure of the logical complexity of the code.

Cyclomatic complexity provides an upper bound on the number of paths that need to be tested in the code.

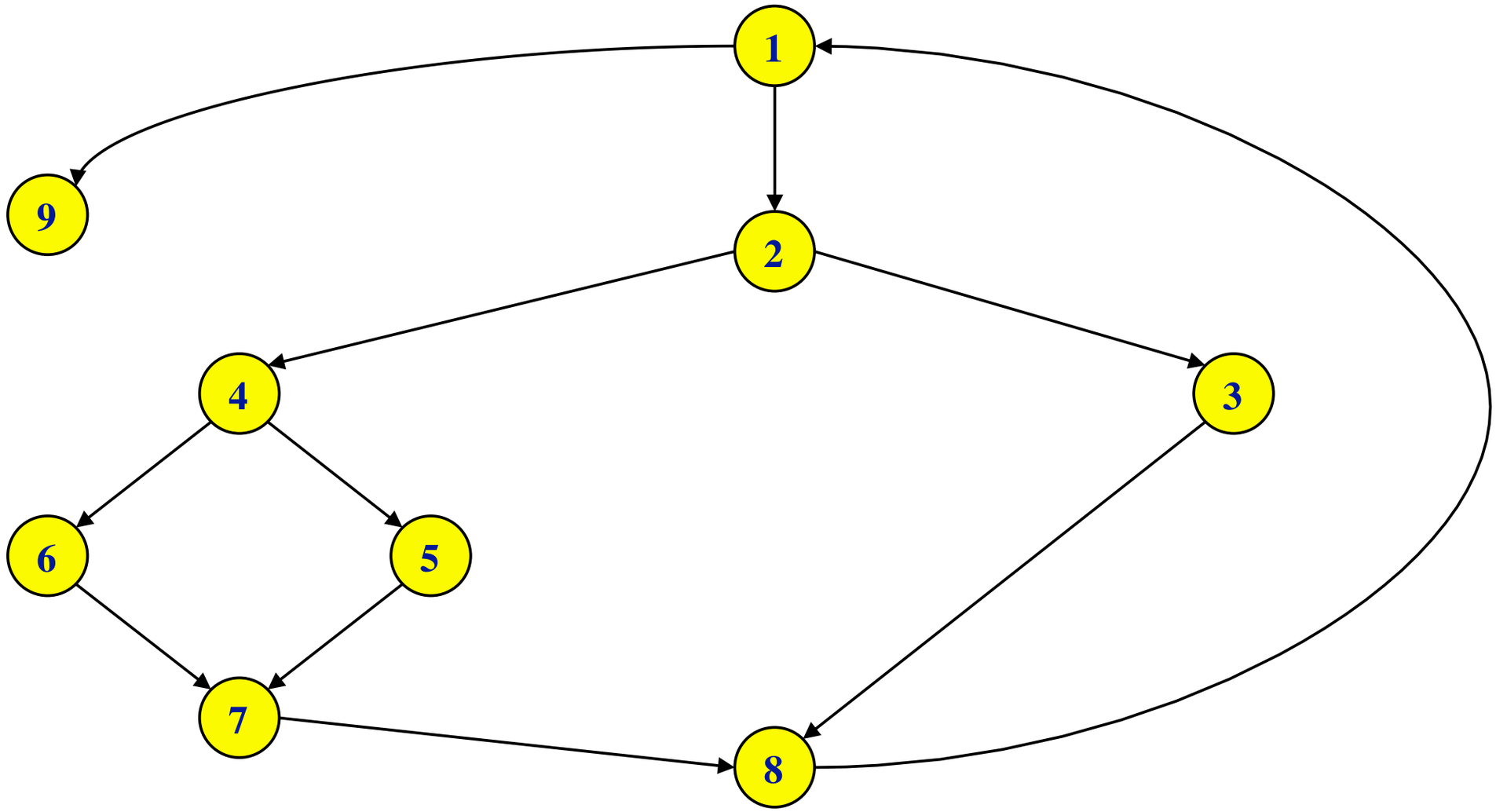
Ways to compute cyclomatic complexity V(G):

- V(G) = the number of regions (areas bounded by nodes and edges—area outside the graph is also a region)
- V(G) = the number of edges - the number of nodes + 2
- V(G) = the number of (simple) predicate nodes + 1

EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY

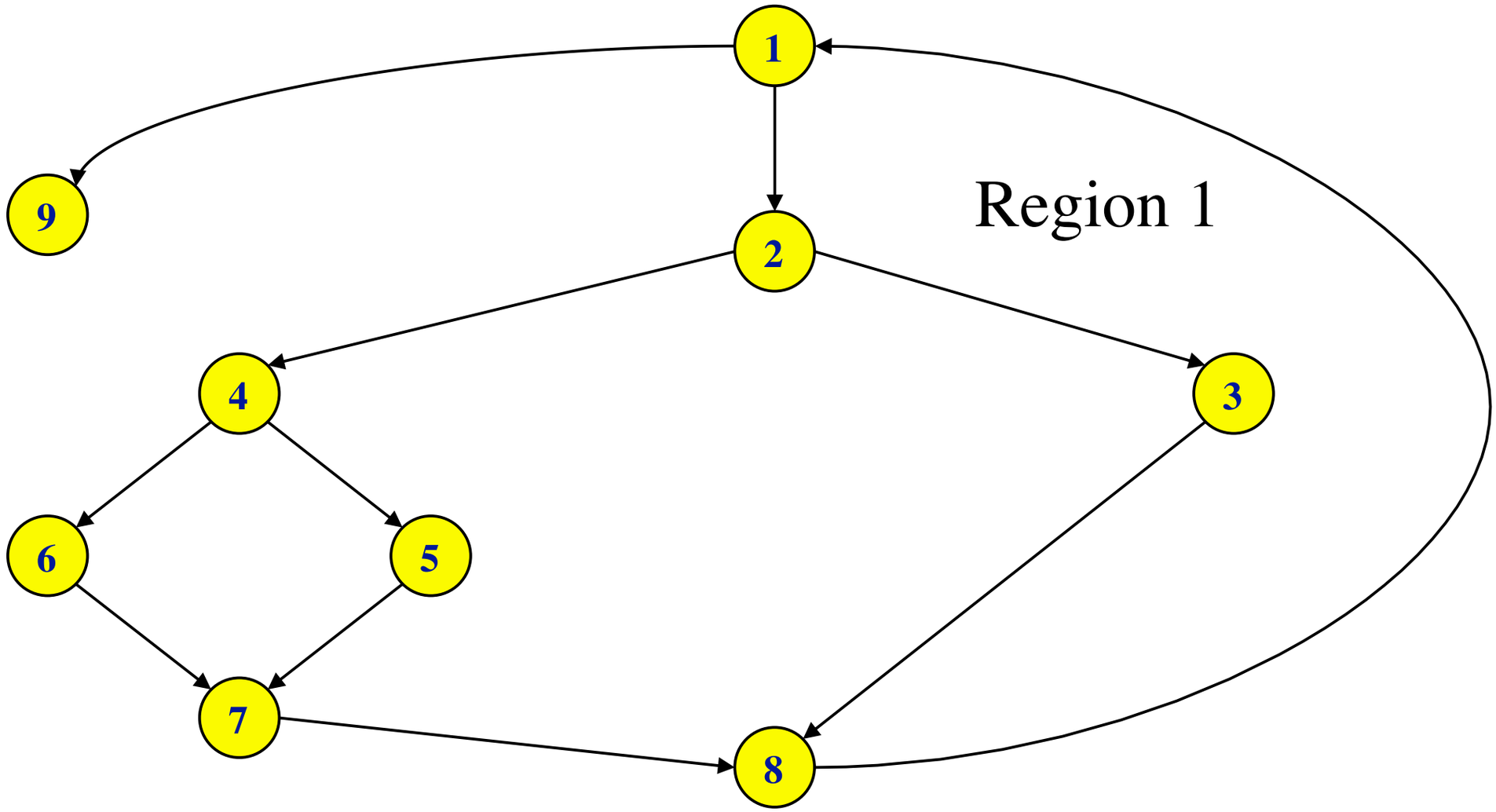


EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY



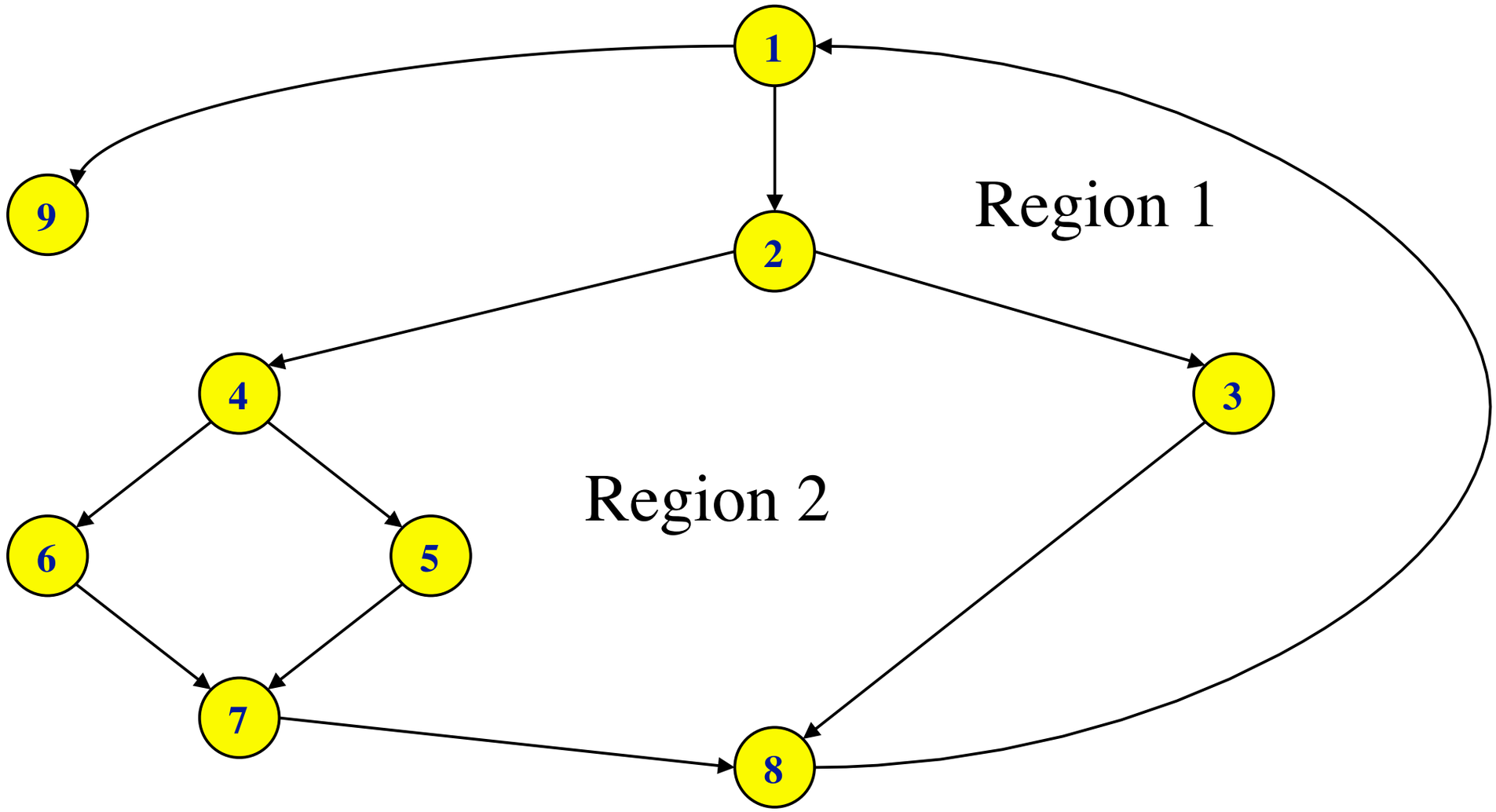
V(G)=4

EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY



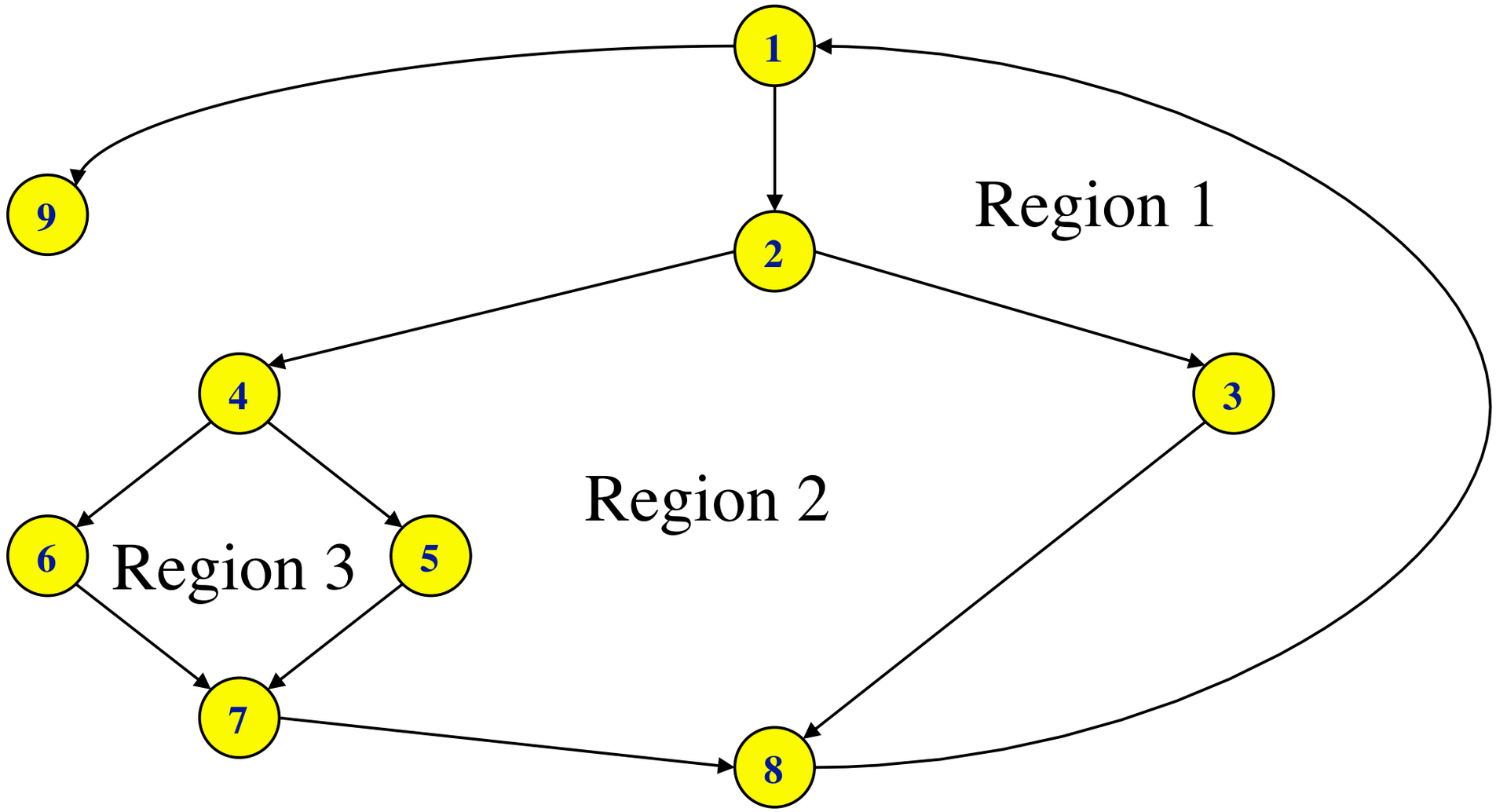
V(G)=4

EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY



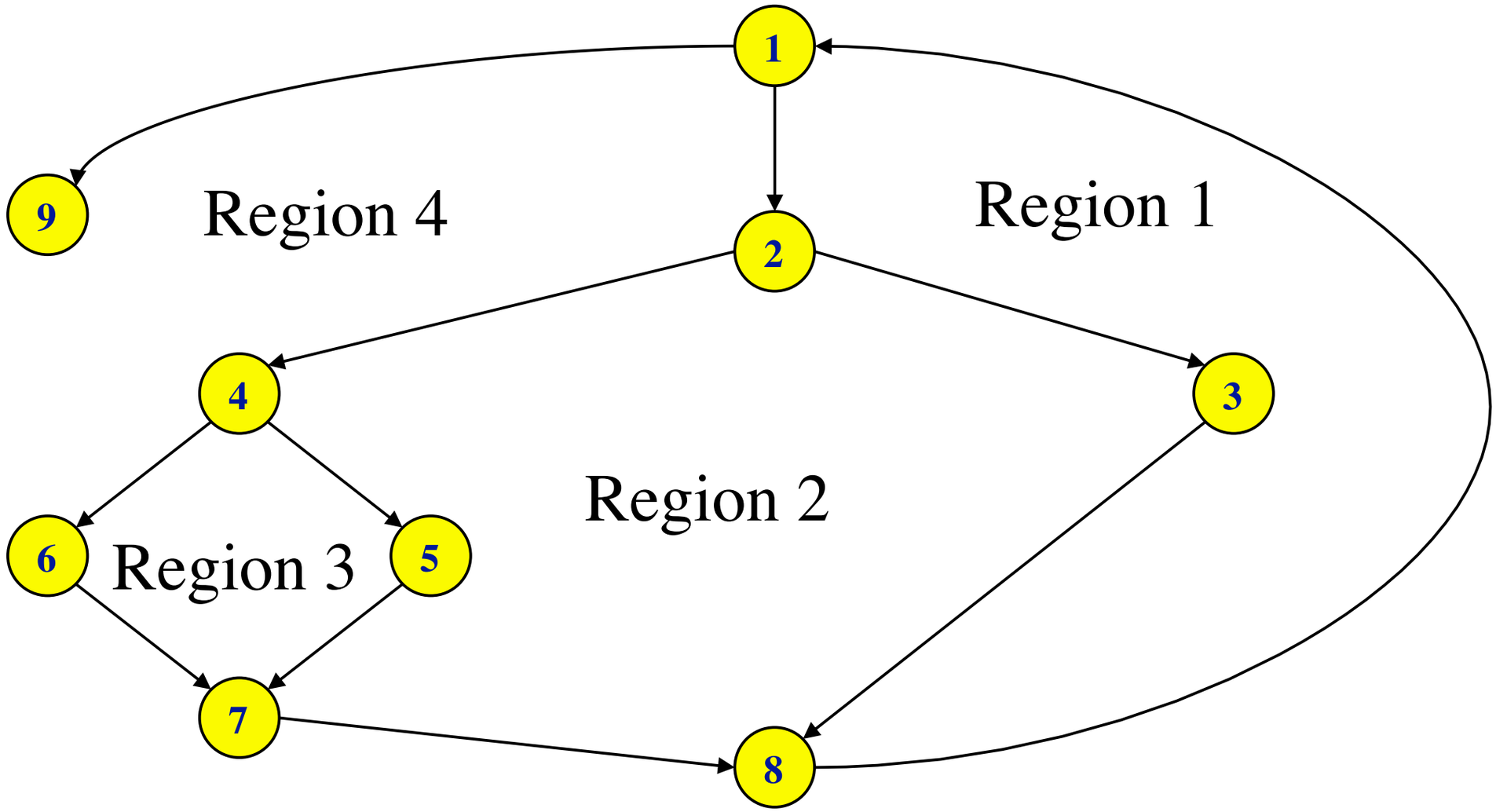
V(G)=4

EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY



V(G)=4

EXAMPLE BASIS PATH TESTING:
CYCLOMATIC COMPLEXITY



V(G)=4

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

3. Determine a basis set of linearly independent paths based on the cyclomatic complexity.

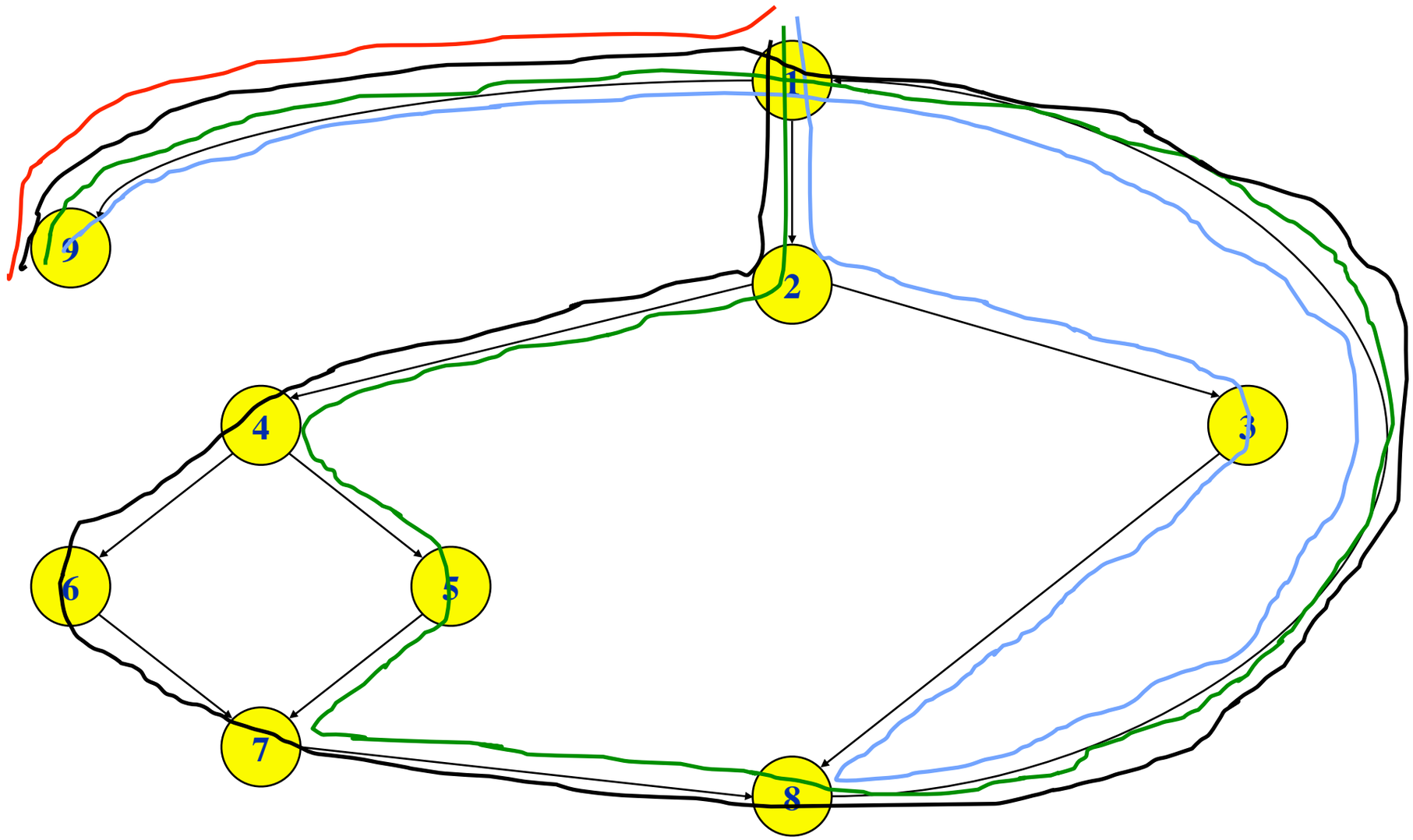
An **independent path** is any path that introduces at least one new set of processing statements or a new condition.

A **basis set** is the set of independent paths through the code.

☞ The basis set is not unique.

☞ Test cases derived from a basis set are guaranteed to execute every statement at least one time during testing.

EXAMPLE BASIS PATH TESTING: INDEPENDENT PATHS



1-9

1-2-3-8-1-9

1-2-4-5-7-8-1-9

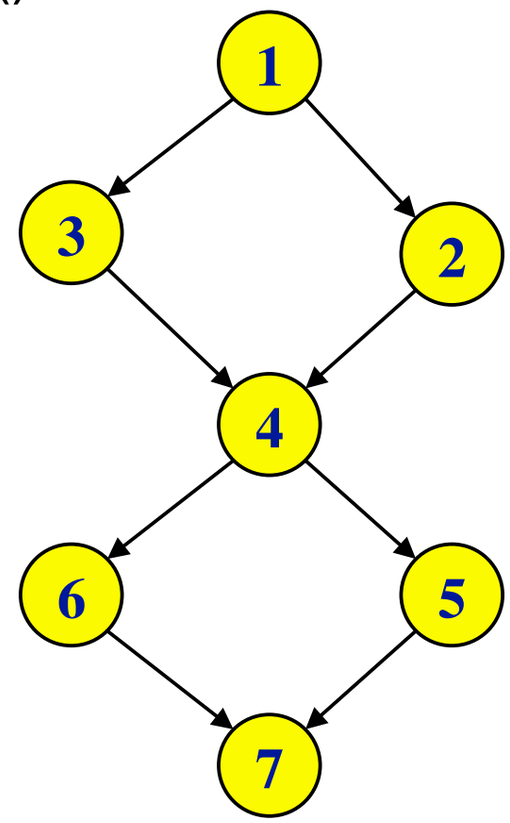
1-2-4-6-7-8-1-9

BASIS PATH TESTING: INDEPENDENT PATHS

```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```

BASIS PATH TESTING: INDEPENDENT PATHS

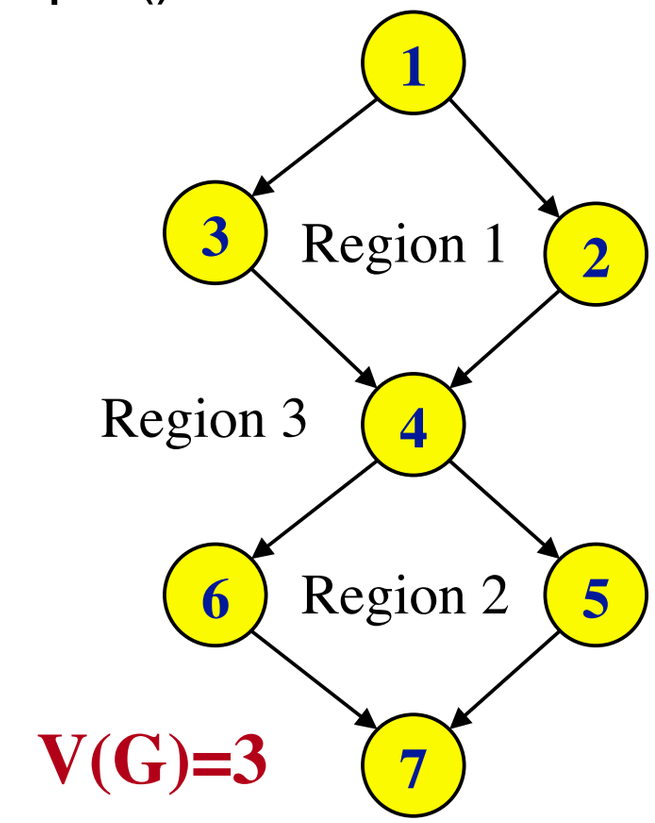
```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```



How many independent paths are there in the basis set?

BASIS PATH TESTING: INDEPENDENT PATHS

```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```



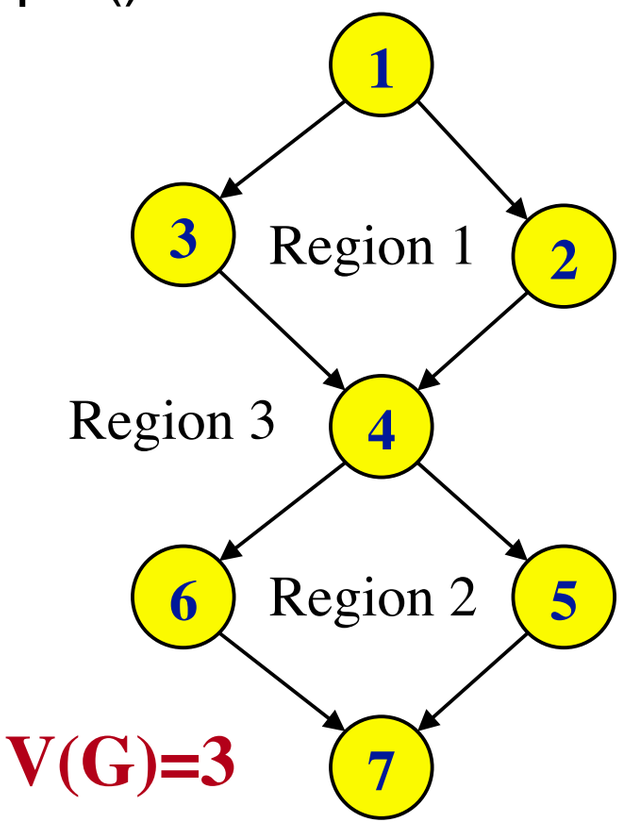
How many independent paths are there in the basis set?

Recall: An independent path is any path that introduces at least one new set of processing statements or a new condition.

☞ V(G) is just an **upper bound** on the number of independent paths.

BASIS PATH TESTING: INDEPENDENT PATHS

```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```



Basis set I:

[1, 2, 4, 5, 7]

[1, 3, 4, 6, 7]

Basis set II

[1,2,4,6,7]

[1,3,4,5,6]

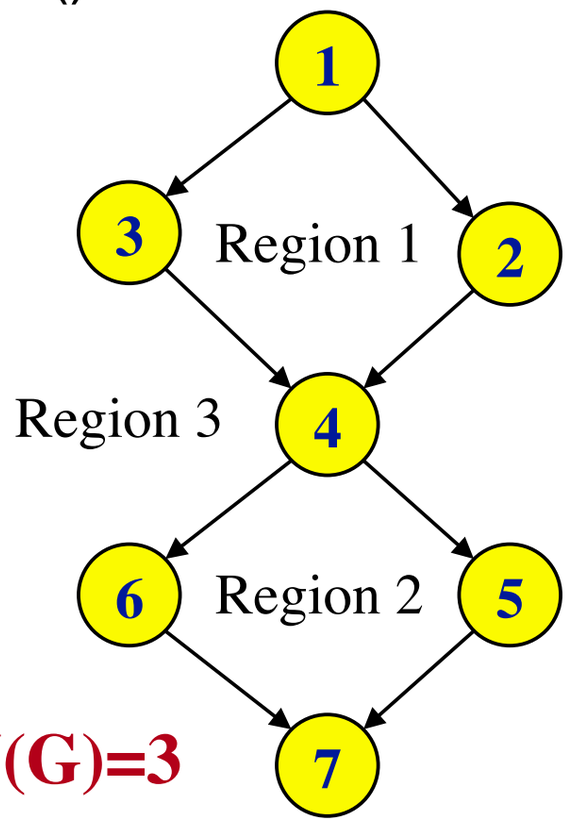
How many independent paths are there in the basis set?

Recall: An independent path is any path that introduces at least one new set of processing statements or a new condition.

☞ V(G) is just an **upper bound** on the number of independent paths.

BASIS PATH TESTING: INDEPENDENT PATHS

```
    Procedure: example()
1.  If c1
2.      f1()
3.  Else
4.      f2()
5.  Endif
6.  If c2
7.      f3()
8.  Else
9.      f4()
10. Endif
11. End
```



Basis set I:

[1, 2, 4, 5, 7]

[1, 3, 4, 6, 7]

Basis set II

[1,2,4,6,7]

[1,3,4,5,6]

How about?

[1,2,4,6,7]

[1,3,4,6,7]

How many independent paths are there in the basis set?

Recall: An independent path is any path that introduces at least one new set of processing statements or a new condition.

☞ V(G) is just an **upper bound** on the number of independent paths.

WHITE BOX TESTING: BASIS PATH TESTING (cont'd)

4. **Prepare test cases that force the execution of each path in the basis set.**

Basis path testing does not test all possible combinations of all paths through the code; *it just tests every path at least once*.

ASU Course Registration System

White Box Testing

Basis Path Testing

EXAMPLE ASU WHITE BOX TESTING:
BASIS PATH TESTING

checkRegistration Procedure

A student can register for a course if he has taken the prerequisites. However, he may not register for a course if he has already taken a course that is an exclusion. A student who does not have the prerequisites can register for a course if he is currently registered in the prerequisite course and has the permission of the instructor. Notwithstanding the preceding, the instructor may waive the prerequisite for a student.

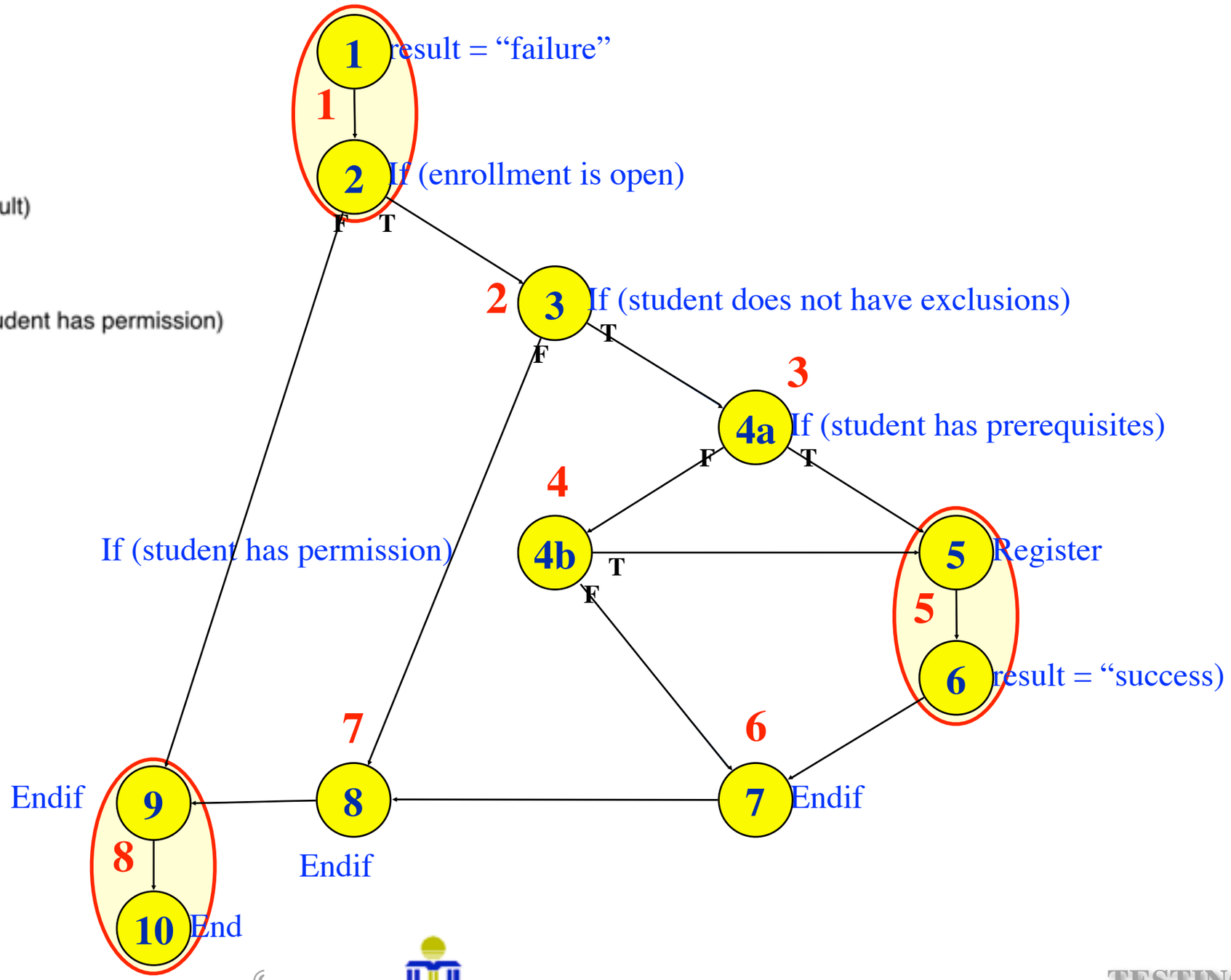
EXAMPLE ASU WHITE BOX TESTING: BASIS PATH TESTING

```
    Procedure: checkRegistration return(result)
1.  result = “failure”
2.  If (enrollment is open)
3.      If (student does not have exclusions)
4.          If (student has prerequisites) OR (student has permission)
5.              Register
6.              result = “success”
7.          Endif
8.      Endif
9.  Endif
10. End
```

EXAMPLE ASU BASIS PATH TESTING: FLOW GRAPH

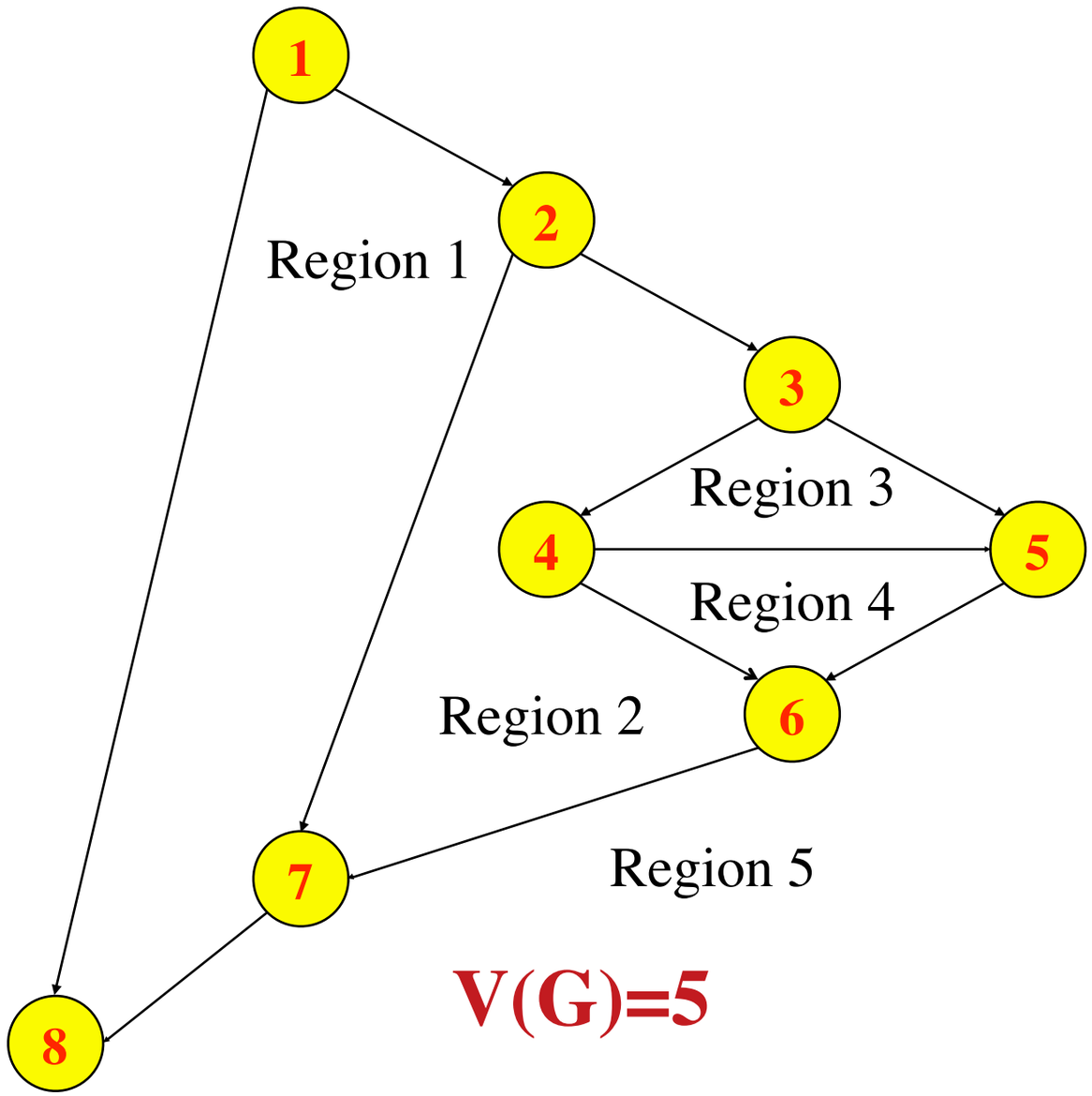
Procedure: checkRegistration

```
    Procedure: checkRegistration return(result)
1.  result = "failure"
2.  If (enrollment is open)
3.     If (student does not have exclusions)
4.        If (student has prerequisites) OR (student has permission)
5.           Register
6.           result = "success"
7.        Endif
8.     Endif
9.  Endif
10. End
```



EXAMPLE ASU BASIS PATH TESTING: FLOW GRAPH

Procedure: checkRegistration



Flow graph node to program statement mapping:

Node	Statement
1.	1, 2
2.	3
3.	4a
4.	4b
5.	5, 6
6.	7
7.	8
8.	9, 10

Basis set (statement numbers):

1. 1 2 3 4a 5 6 7 8 9 10
2. 1 2 3 4a 4b 5 6 7 8 9 10
3. 1 2 3 4a 4b 7 8 9 10
4. 1 2 3 8 9 10
5. 1 2 9 10

EXAMPLE ASU BASIS PATH TESTING: TEST DATA

checkRegistration Procedure

Basis set path 1: enrollment is open
student does not have exclusions
student has prerequisites

Basis set path 2: enrollment is open
student does not have exclusions
student does not have prerequisites
student has permission

Basis set path 3: enrollment is open
student does not have exclusions
student does not have prerequisites
student does not have permission

Basis set path 4: enrollment is open
student has exclusions

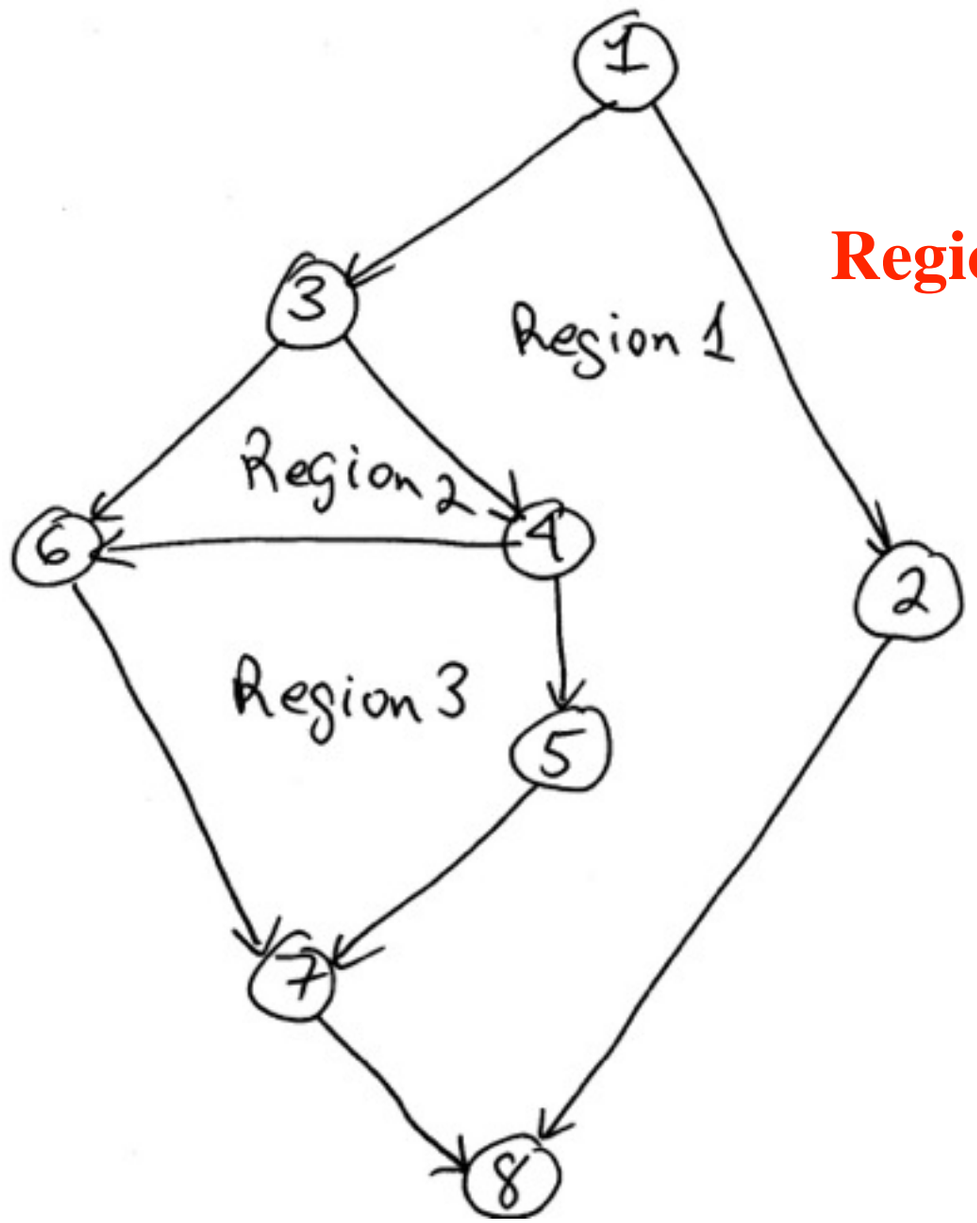
Basis set path 5: enrollment is closed

BASIS PATH TESTING EXAMPLE: COMMON ERRORS

- Missing the outside region
- Missing a node for the last statement(s)
- Having a condition with only one out-going branch
- Not understanding what is a flow graph

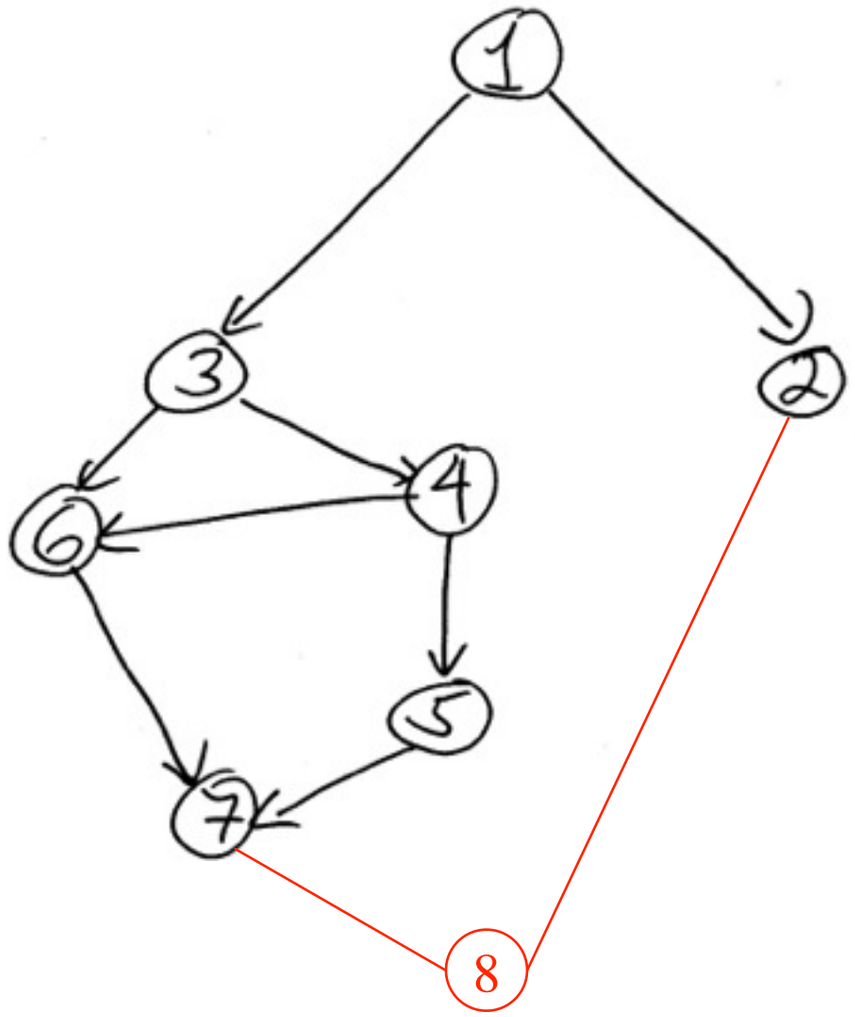
BASIS PATH TESTING EXAMPLE: COMMON ERRORS

- Missing the outside region



BASIS PATH TESTING EXAMPLE: COMMON ERRORS

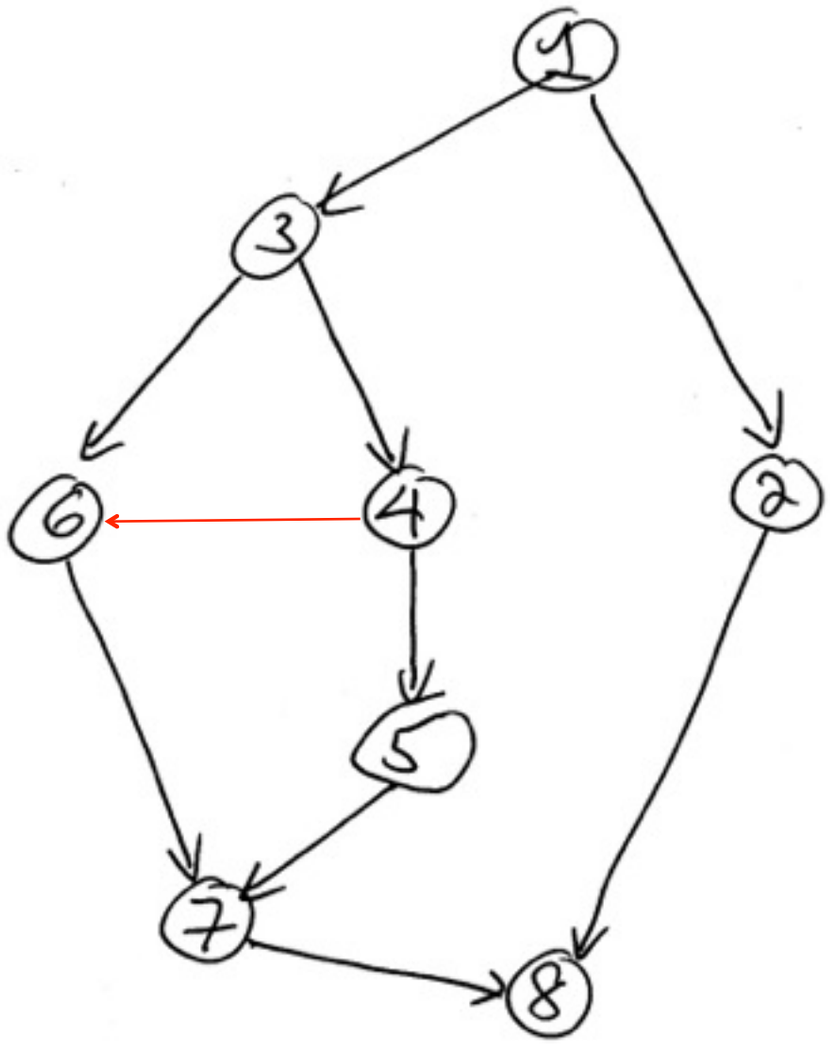
- Missing a node for the last statement(s)



Node	Statement
1	1
2	2, 3, 4
3	5, 6a
4	6b
5	7, 8, 9
6	10, 11
7	12
8	13, 14

BASIS PATH TESTING EXAMPLE: COMMON ERRORS

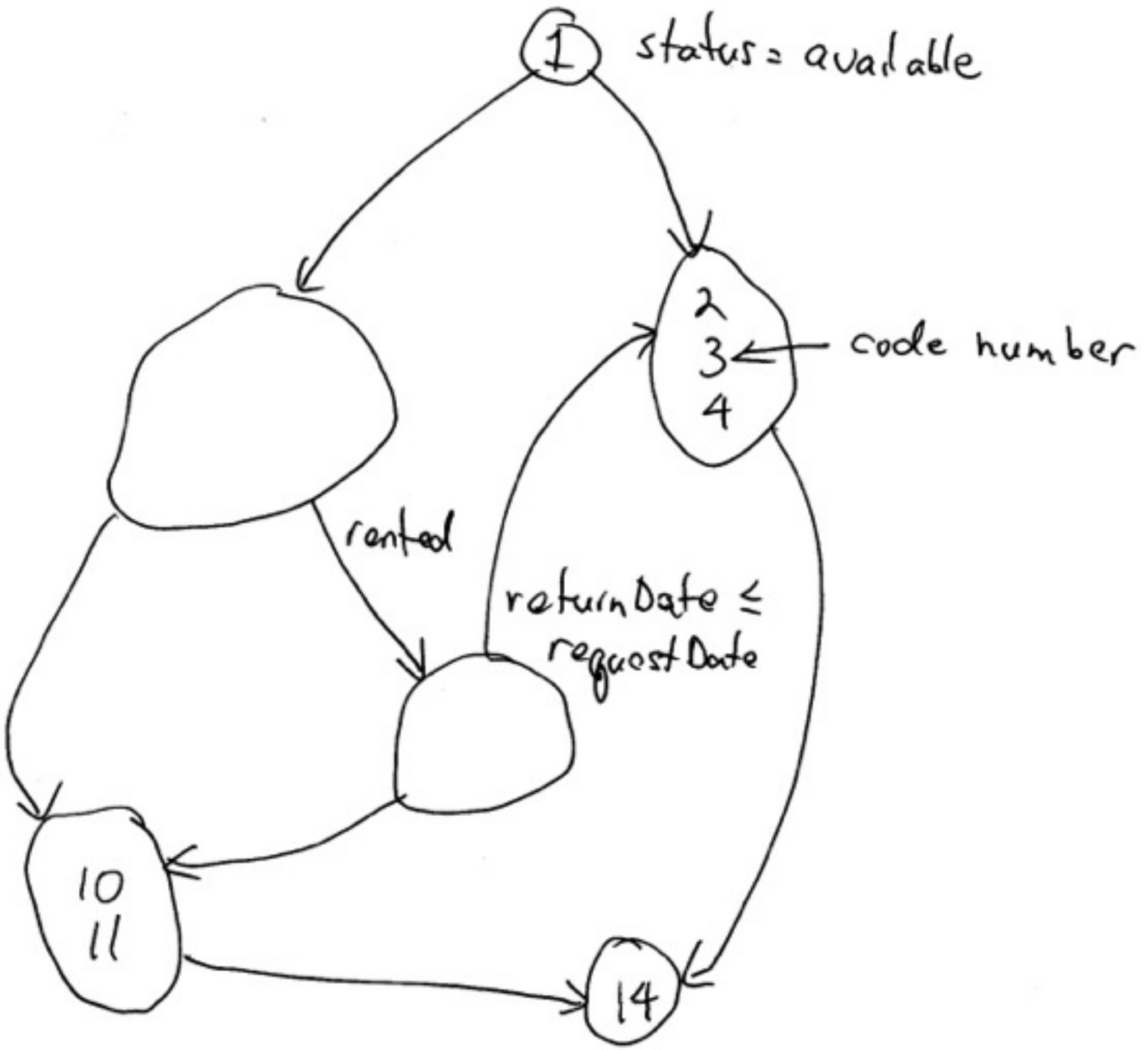
- Having a condition with only one out-going branch



Node	Statement
1	1
2	2, 3, 4
3	5, 6a
4	6b
5	7, 8, 9
6	10, 11
7	12
8	13, 14

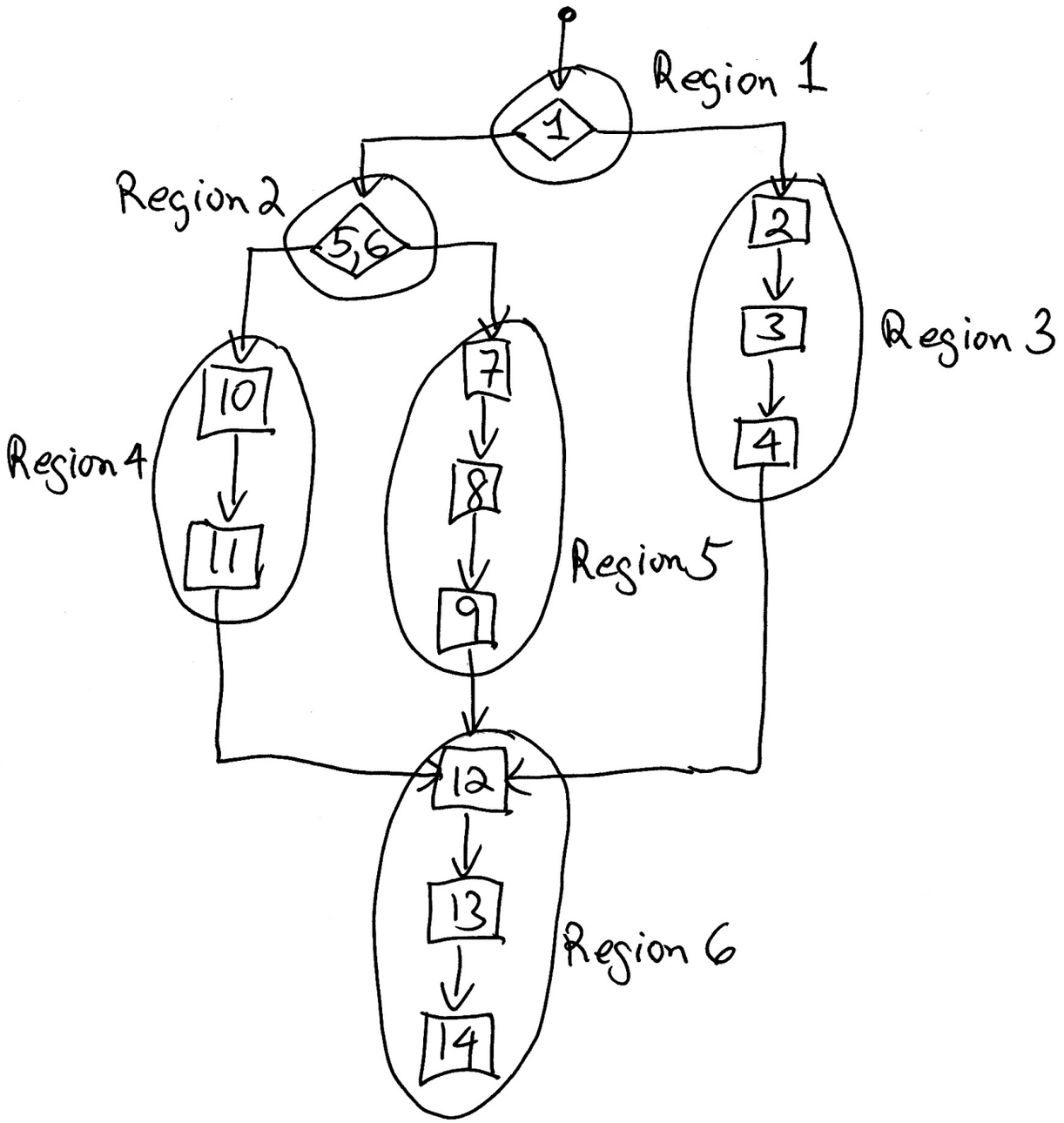
BASIS PATH TESTING EXAMPLE: COMMON ERRORS

- Not understanding what is a flow graph



BASIS PATH TESTING EXAMPLE: COMMON ERRORS

- Not understanding what is a flow graph



DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: To design *selective tests* to ensure that we have executed/ exercised all:

1. independent paths in the code at least once.

   ☞ **Basis Path Testing**

2. logical decisions on their true and false sides.

   ☞ **Condition Testing**

3. loops at their boundaries and within their bounds.

   ☞ **Loop Testing**

4. internal data structures to ensure their validity.

   ☞ **Data Flow Testing**

WHITE BOX TESTING: CONDITION TESTING

***Condition testing* further exercises the true and false value of each simple logical condition in a component.**

Errors in a condition result from errors in:

simple condition: (a rel-op b) where rel-op={<, ≤, =, ≠, ≥, >} may be negated with NOT, e.g., a≤b; NOT(a≤b)

compound condition: two or more simple conditions connected with AND, OR, e.g., (a>b) AND (c<d)

relational expression: (E_1 rel-op E_2) where E_1 and E_2 are arithmetic expressions, e.g., ((a*b+c)>(a+b+c))

Boolean expression: non relational expressions (e.g., NOT A)

Due to incorrect/missing/extra:

- Boolean operator
- Boolean variable
- parenthesis
- relational operator
- arithmetic expression

WHITE BOX TESTING: CONDITION TESTING (cont'd)

1. Branch testing

For a compound condition C, test true and false branches of C and every simple condition of C.

e.g., for C = (a>b) AND (c<d) we test for:

C	TRUE and FALSE
a>b	TRUE and FALSE
c<d	TRUE and FALSE

Basis path testing already covers these cases.

2. Domain testing

For an expression E_1 rel-op E_2, we test using values that make: E_1 greater than E_2, E_1 equal to E_2, and E_1 less than E_2.

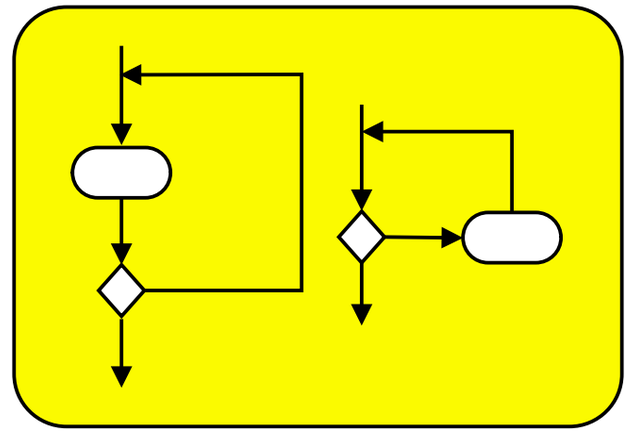
- This guarantees detection of rel-op error if E_1 and E_2 are correct.
- To detect errors in E_1 and E_2, the difference between E_1 and E_2 for the tests that make E_1 greater than E_2 and E_1 less than E_2 should be as small as possible.

WHITE BOX TESTING: LOOP TESTING

***Loop testing* executes loops at and within their bounds.**

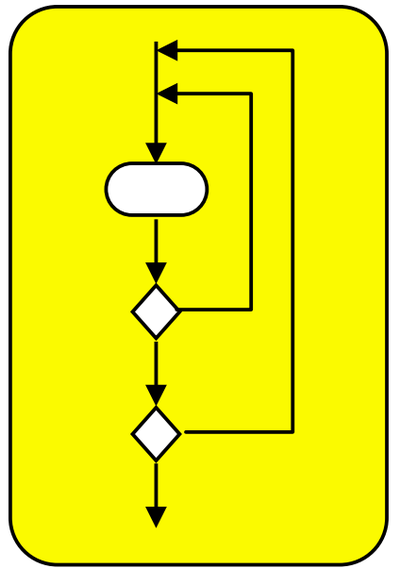
1. Simple Loops (n iterations)

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n-1, n, n+1 passes through the loop.



2. Nested Loops

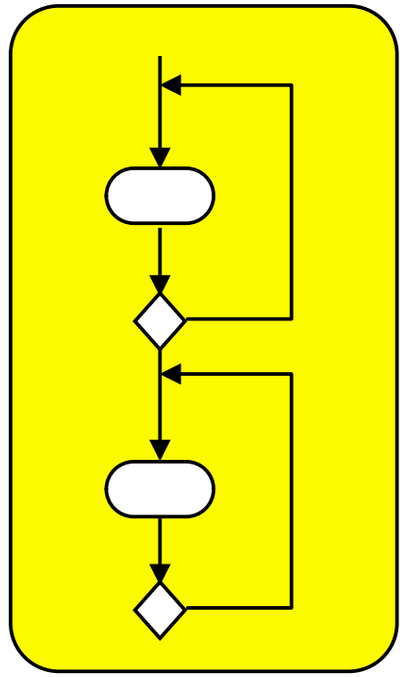
1. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration values.
2. Work outward, conducting simple loop tests for the next innermost loop.
3. Continue until all the loops have been tested.



Tests grow geometrically as the level of nesting increases!

WHITE BOX TESTING: LOOP TESTING (cont'd)

3. Concatenated Loops



☞ If loops are independent
→ use simple loop testing

☞ If loops are dependent (i.e., one loop depends on a variable set in the other loop)
→ use nested loop testing

4. Unstructured Loops

☞ **Redesign the code!!!**

These loop test are designed to have the highest likelihood of uncovering errors with the minimum amount of effort and test overlap.

WHITE BOX TESTING: DATA FLOW TESTING

***Data flow testing* ensures that the value of a variable is correct at certain points of execution in the code.**

Select test paths according to the locations of definitions (S) and uses (S') of a variable (X).

DEF(S) = {X I S contains a definition of X} → locations of *definition* of X

USE(S´) = {X I S´ contains a use of X} → locations of *use* of X

Definition Use (DU) Chain (X) is the set of [X, S, S´] where X is "live" (i.e., not redefined between S and S´)

S, S' are statement numbers

A testing strategy: Every DU chain must be covered once.

For every variable, do a test along the path from where the variable is defined to the statement(s) where the variable is used.
These tests can be combined with basis path testing.

DESIGN TESTS: WHITE/GLASS BOX TESTING

Goal: To design *selective tests* to ensure that we have executed/ exercised all:

1. independent paths in the code at least once.

   ☞ **Basis Path Testing**

2. logical decisions on their true and false sides.

   ☞ **Condition Testing**

3. loops at their boundaries and within their bounds.

   ☞ **Loop Testing**

4. internal data structures to ensure their validity.

   ☞ **Data Flow Testing**