TDDD04: White box testing

Lena Buffoni lena.buffoni@liu.se

LINKÖPING UNIVERSITY

White box testing - outline

- Control flow coverage
  - Statement, decision and condition coverage
  - Condition/decision coverage
  - Multiple condition coverage
  - Modified condition/decision coverage
  - Loop testing
  - Basis path testing (a.k.a. structured testing)
- Program complexity
- Mutation testing

White box testing

- Analyze SUT
- Identify paths to execute
- Choose inputs to trigger those paths and determine expected outputs
- Run tests
- Compare actual outputs to expected

- Can be applied at all levels: unit, integration and system

Limitations

- Testing all paths is complicated/impossible
- Not data sensitive
  - eg: p=q/r;
- Non-existent paths cannot be discovered
- The tester must have programming skills

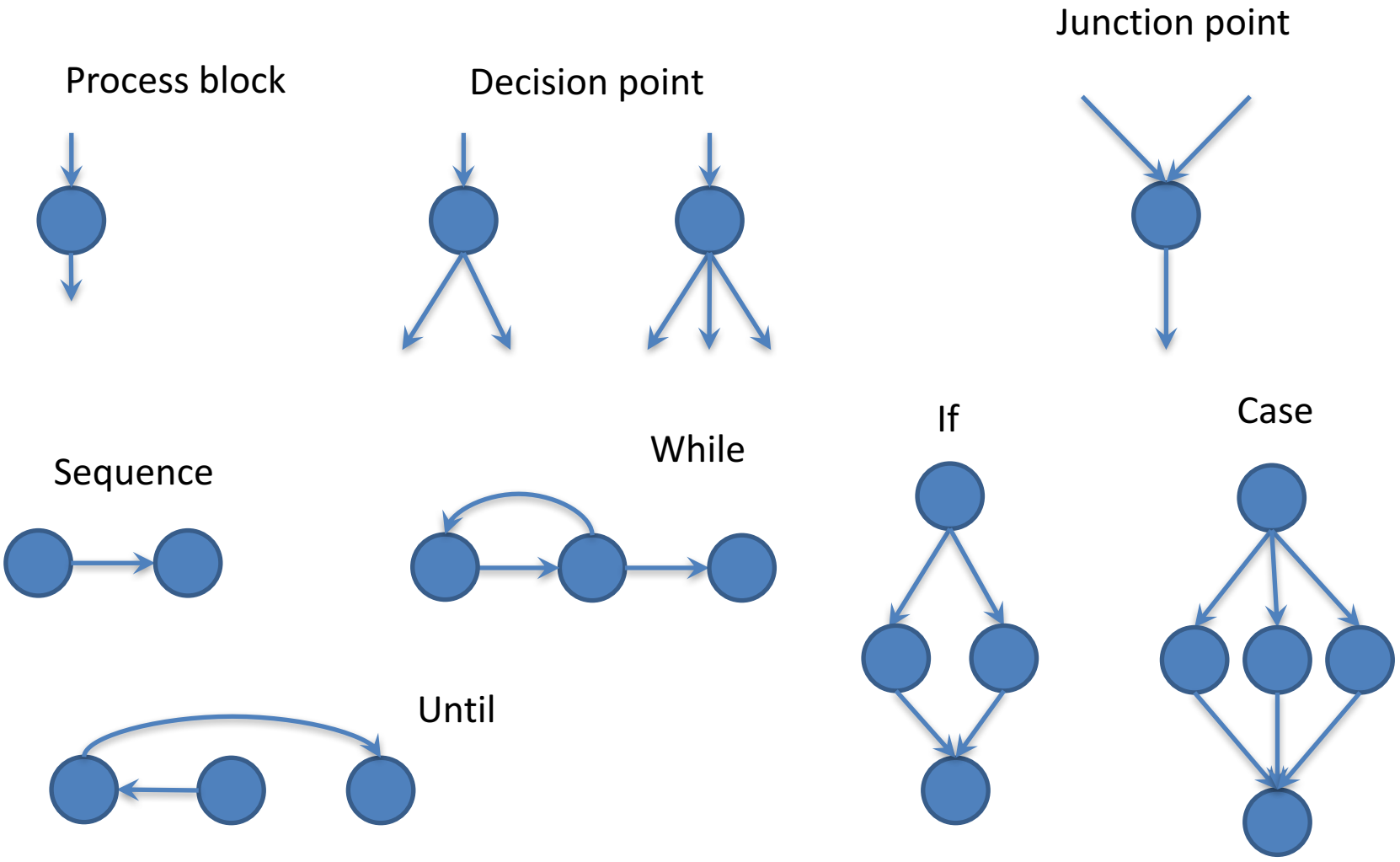
Control flow testing

- Based on the flow of control in the program
- Logical decisions
- Loops
- Execution paths
- Coverage metrics
- Measure of how complete the test cases are
- *Not* the same as how good they are!

Control flow graphs (CFGs)

- **Definition**: a control flow graph is a graph representation of a program in which the *vertices* (nodes) represent *basic blocks of the program*, and *edges* represent *transfer of execution* between basic blocks.
- A **basic block** is a region in the program with a single entry point and a single exit point. This means that all jump targets start new basic blocks, and all jumps terminate basic blocks.

CFG Notation

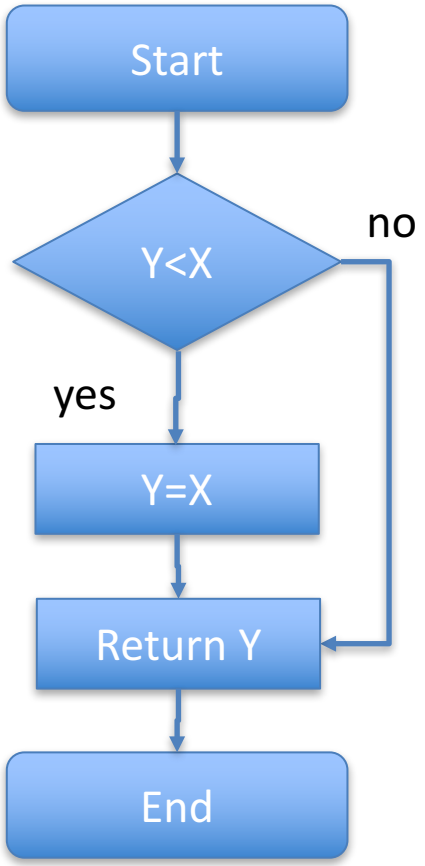


Levels of code coverage

- Statement/Line/Basic block/Segment Coverage
- Decision (Branch) Coverage
- Condition Coverage
- Multiple Condition Coverage
- Decision/Condition Coverage
- Loop Coverage
- Path Coverage

Statement coverage

```
// Return smallest value int min(int y, int x) {
if (y < x)
        y = x;
return y;
}
```

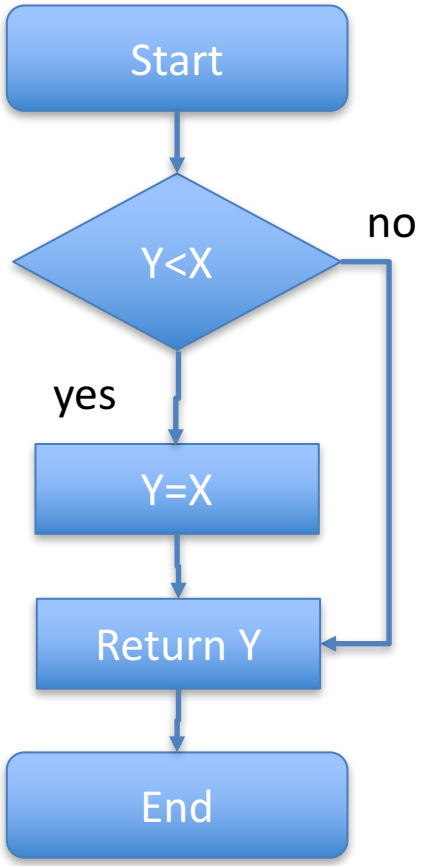


Write test cases to execute each statement at least once

Test case	x	y	expected	actual

Statement coverage

```
// Return smallest value int min(int y, int x) {
if (y < x)
        y = x;
return y;
}
```



Test case	x	y	expected	actual
1	1	0	0	1

100% statement coverage

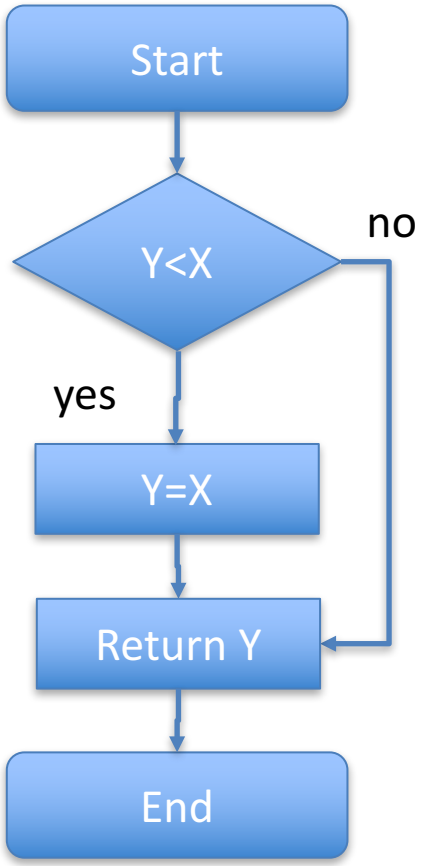
What is wrong with line (statement) coverage?

Steve Cornett (Bullseye testing technology)

- Software developers and testers commonly use line coverage because of its simplicity and availability in object code instrumentation technology.
- Of all the structural coverage criteria, line coverage is the weakest, indicating the fewest number of test cases.
- Bugs can easily occur in the cases that line coverage cannot see.
- The most significant shortcoming of line coverage is that it fails to measure whether you test simple if statements with a **false decision outcome**. Experts generally recommend to only use line coverage if nothing else is available. **Any other measure is better.**

Decision coverage

```
// Return smallest value int min(int y, int x) {
if (y < x)
        y = x;
return y;
}
```

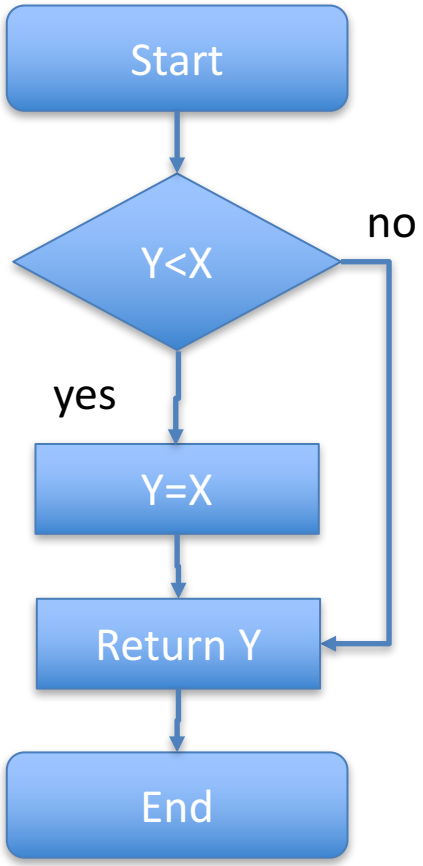


Write test cases to execute each edge in the CFG at least once

Test case	x	y	expected	actual

Decision coverage

```
// Return smallest value int min(int y, int x) {
if (y < x)
        y = x;
return y;
}
```



Write test cases to execute each edge in the CFG at least once

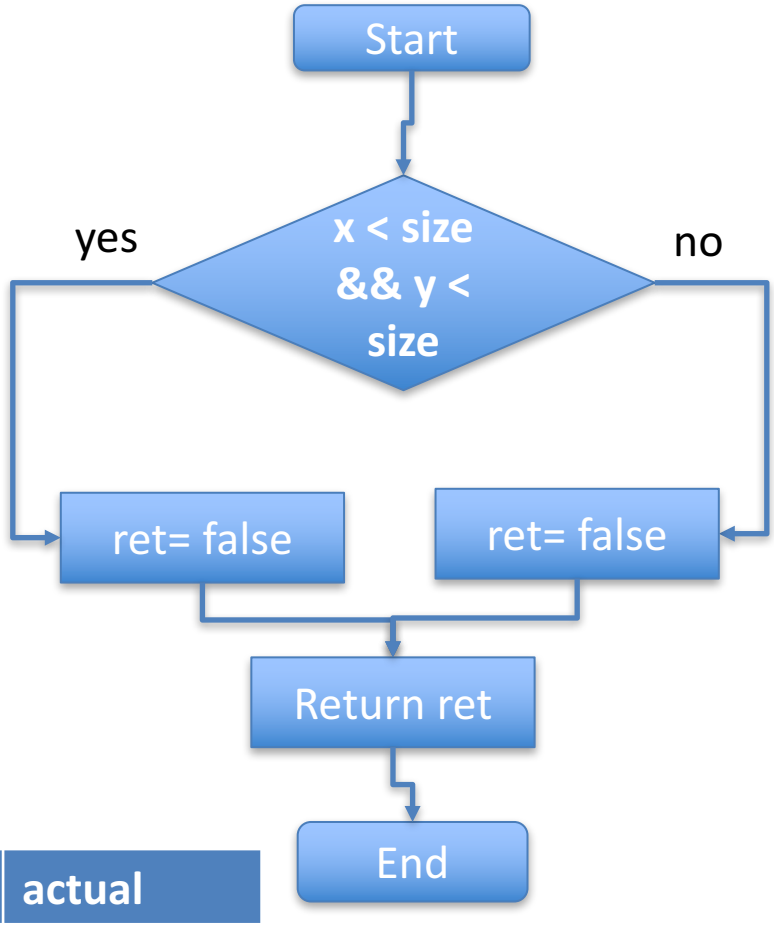
Test case	x	y	expected	actual
1 (T)	1	1	1	1
2 (F)	0	1	0	1

100% branch coverage

Condition coverage

```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

Write test cases so that each condition in each decision is both true and false

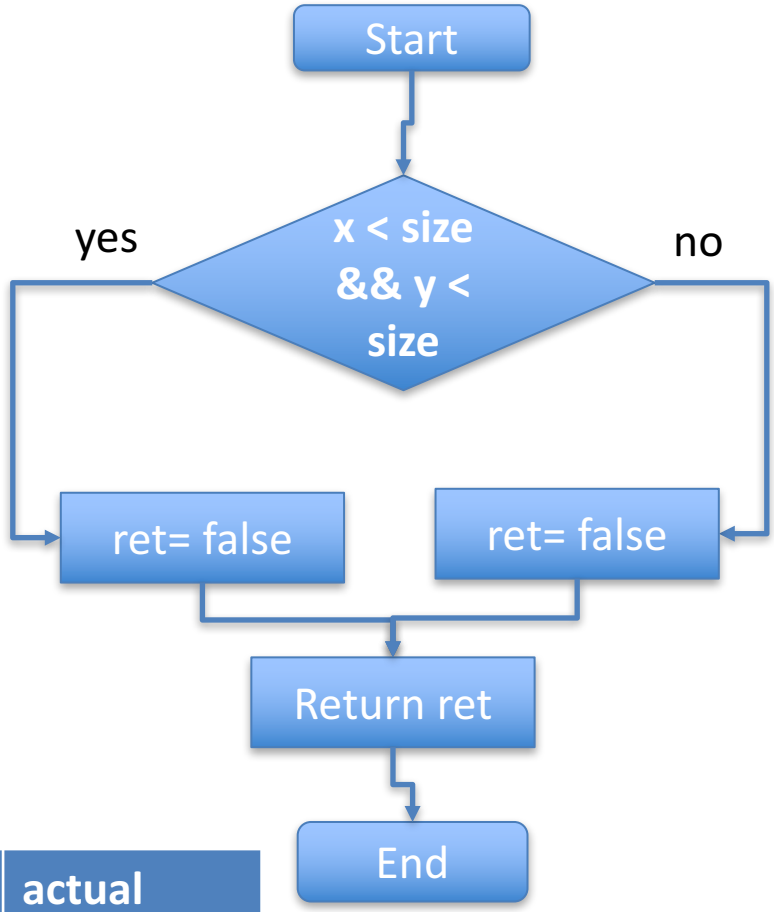


Test case	x	y	size	expected	actual

Condition coverage

```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

Write test cases so that each condition in each decision is both true and false

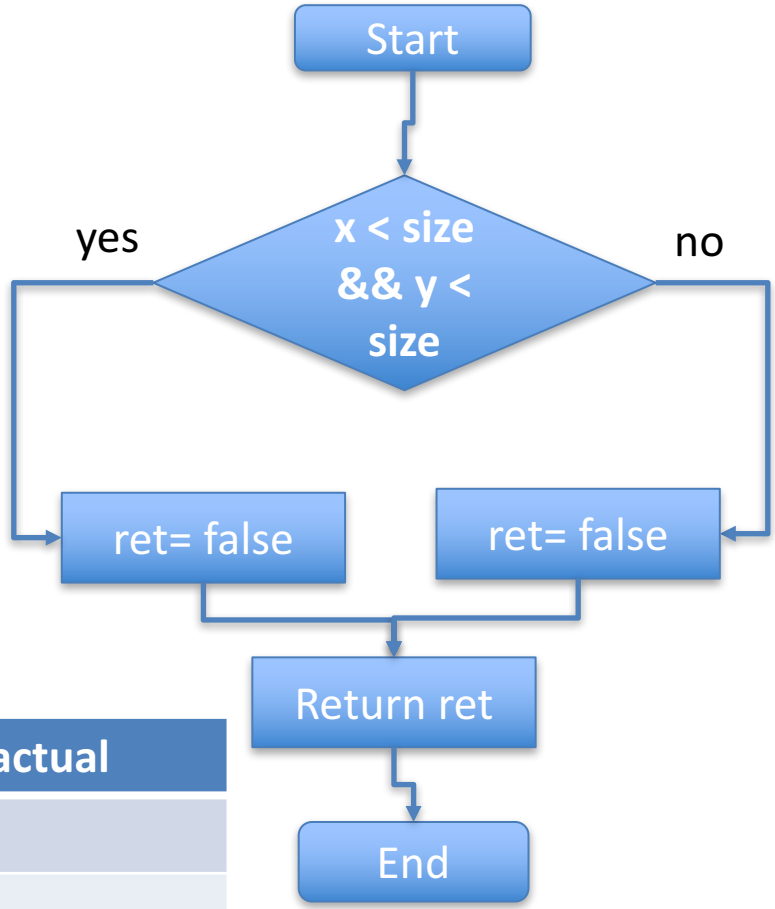


Test case	x	y	size	expected	actual
1(TF)	3	7	5	false	false
2(FT)	7	3	5	false	false

No 100% branch coverage

Multiple condition coverage (MCC)


```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```



Write test cases so that all combinations of conditions are executed in each decision

Test case	x	y	size	expected	actual
1(TF)					
2(FT)					
3(TT)					
4(FF)					

Problems with MCC

Consider the following simplified rule for insurance coverage

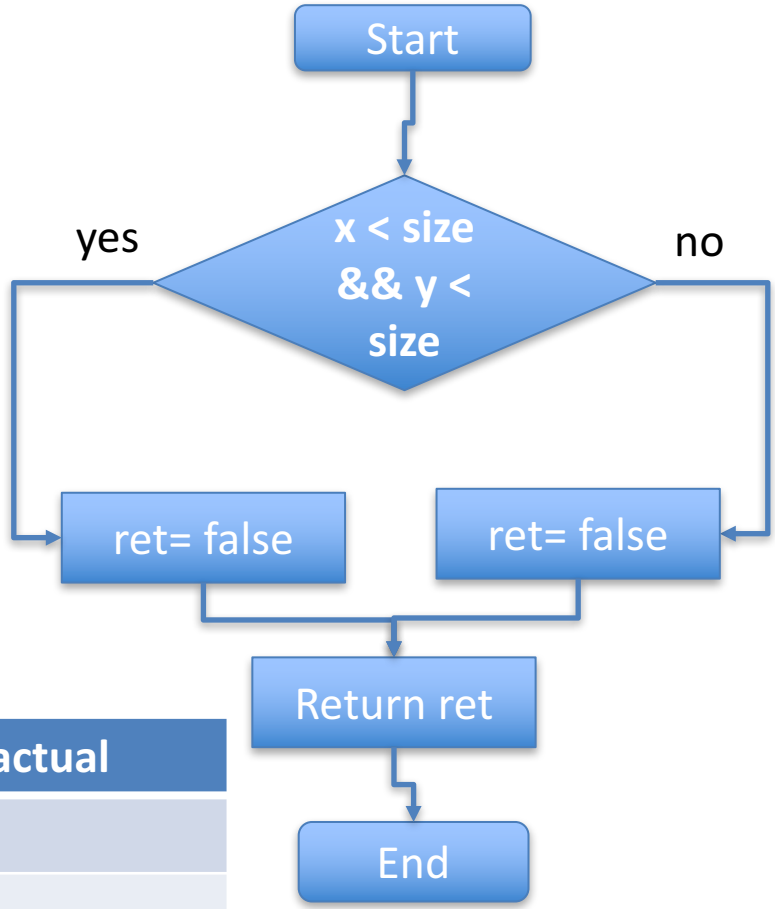
```
if ((age >= 17 and age + duration <= 80
     and issued <= today – timedelta(year=1)
     and country == ‘UK’ and accidents <= 5
     and convictions <= 3 and
     and (owner == applicant or owner == spouse or owner == partner)
     and not modified
     and (driver == applicant or driver == spouse or driver == partner)

     and milage <= 50000 and value <= 50000
     and not (job == entertainment or job == fashion
              or job == sports or job == fs or job == diplomatic
              or job == scrap or job == general_dealer or job == minicab)
     and ((use == social or use == pleasure or use == commuting)
     and (job == motor_trade or job == retired or job == house
              or job == driver or job == none or job == student
              or job == unemployed)):
```

- **How many test cases? Are all test cases even possible?**

Condition/decision coverage

```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

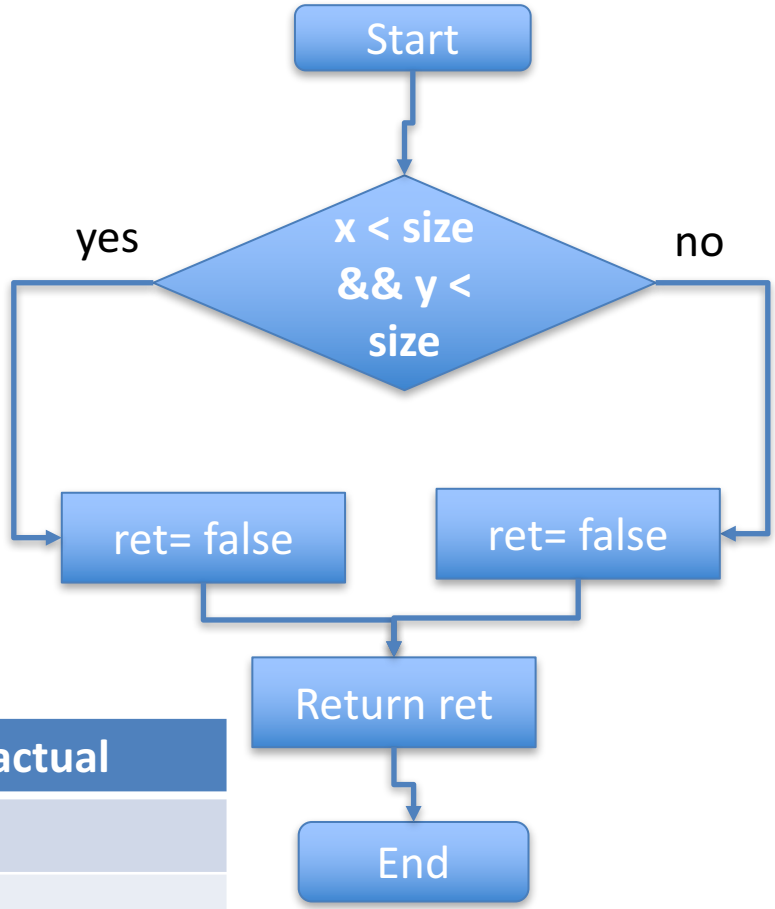


Write test cases so that achieve both condition and decision coverage

Test case	x	y	size	expected	actual

Condition/decision coverage


```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

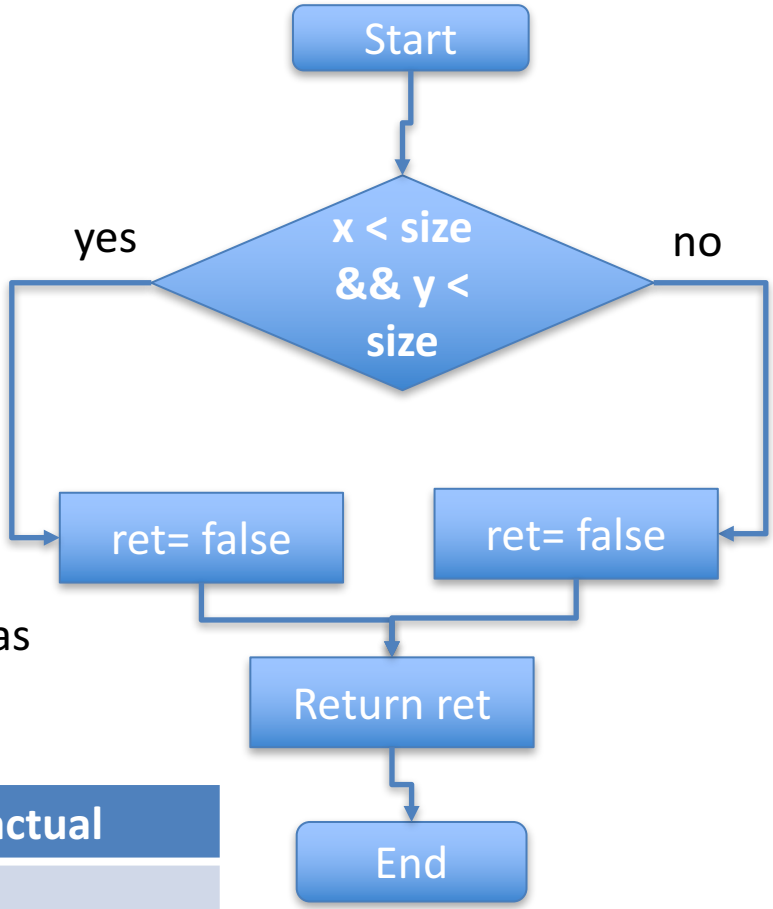


Write test cases so that achieve both condition and decision coverage

Test case	x	y	size	expected	actual
1(TT)					
2(FF)					

Modified condition/decision coverage (MCDC)


```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

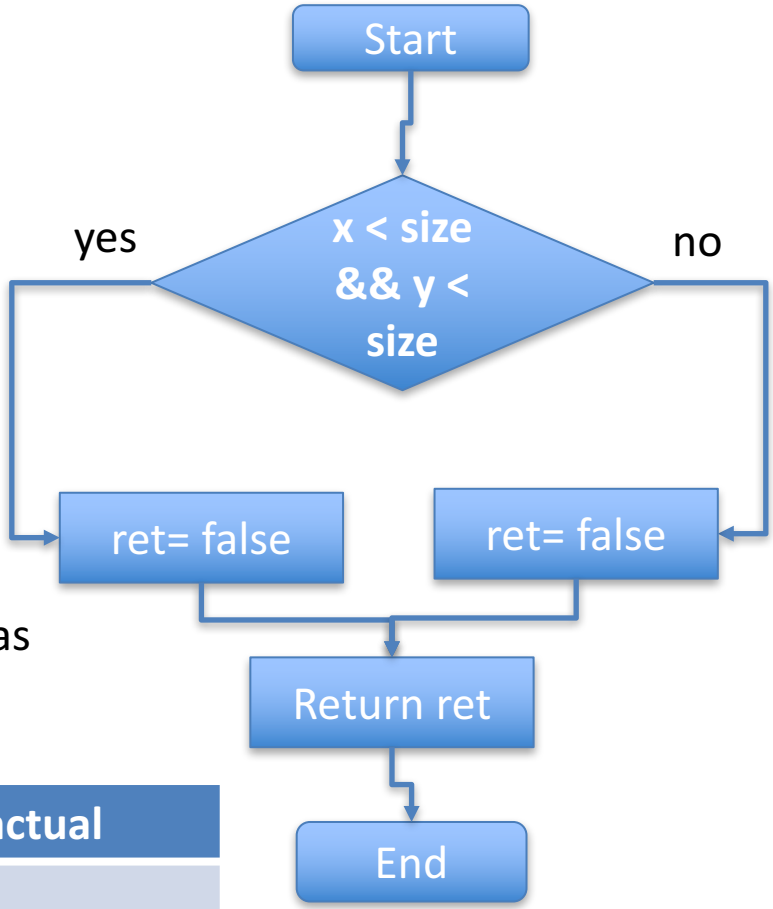


Write test cases so that achieve both condition and decision coverage AND each condition in a decision has been shown to independently affect that decision's outcome

Test case	x	y	size	expected	actual

Modified condition/decision coverage (MCDC)


```
// Return true if (x,y) is in the
// lower left size x size grid sector.
boolean is_ll(int x, int y, int size) {
boolean ret if (x < size && y < size)
        ret = false;
        else ret = false;
return ret;
}
```

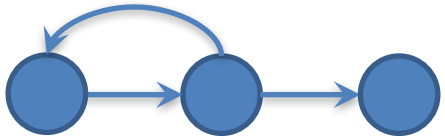


Write test cases so that achieve both condition and decision coverage AND each condition in a decision has been shown to independently affect that decision's outcome

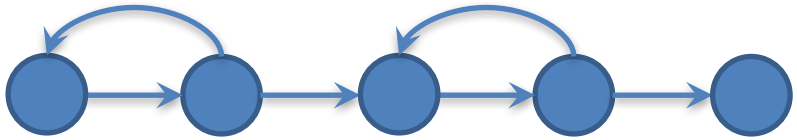
Test case	x	y	size	expected	actual
1(TF)					
2(FT)					
3(TT)					

Loop coverage

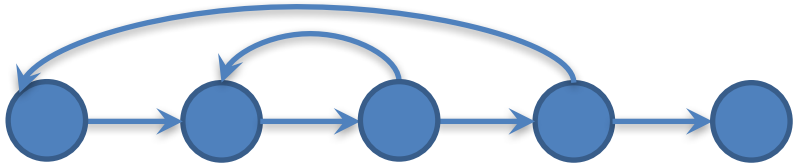
Simple loop



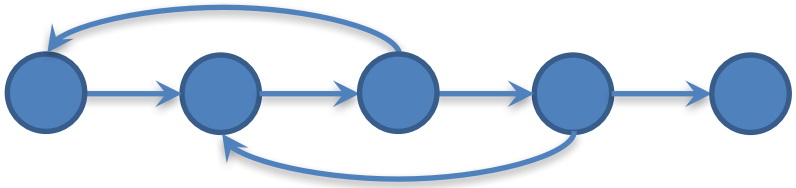
Concatenated loops



Nested loops



Knotted loops



Loop coverage

- Minimum number
- Minimum number + 1
- Skip the loop entirely (unless covered above)
- One pass through the loop (unless covered above)
- Two passes through the loop (unless covered above)
- Maximum expected – 1
- Maximum expected
- One less than minimum (if possible)
- One more than maximum (if possible)

Path testing

- A path is a sequence of branches, or conditions
- A path corresponds to a test case, or a set of inputs
- Bugs are often sensitive to branches and conditions
- **All-paths coverage:** cover all possible paths through a program
  - Not possible in the general case (e.g. loops)
  - Approximations must be used: statement, branch, MCC, MCDC, loop…
- **Basis path coverage:** cover all *independent paths*
  - Idea behind the structured testing method
  - Independent paths are limited in number

Independent paths

Finding independent paths

- An **independent path** is any path through the program that introduces at least **one new set of processing statements or a new condition.** When stated in terms of a flow graph, an independent path must move along **at least one edge** that has not been traversed before the path is defined.

Basis path set

- A set of linearly independent paths through the program
- Any path through the program can be formed as a linear combination of elements in the basis set
- Size equals the **cyclomatic complexity** of the control flow graph

Cyclomatic complexity

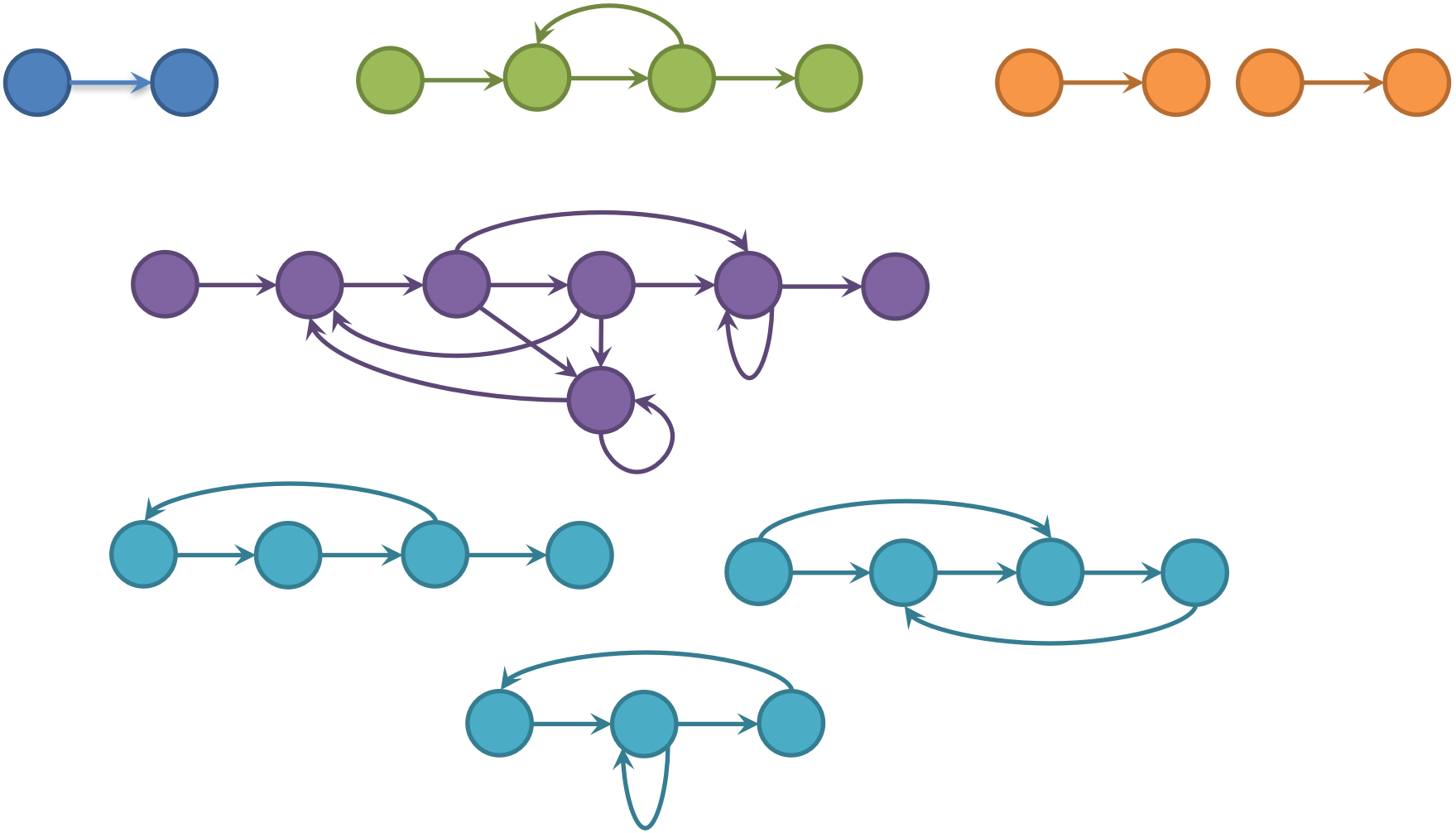
Cyclomatic Complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an **upper bound for the number of tests** that must be conducted to ensure that all statements have been executed at least once.

Developed by John McCabe in 1976
as a software complexity metric

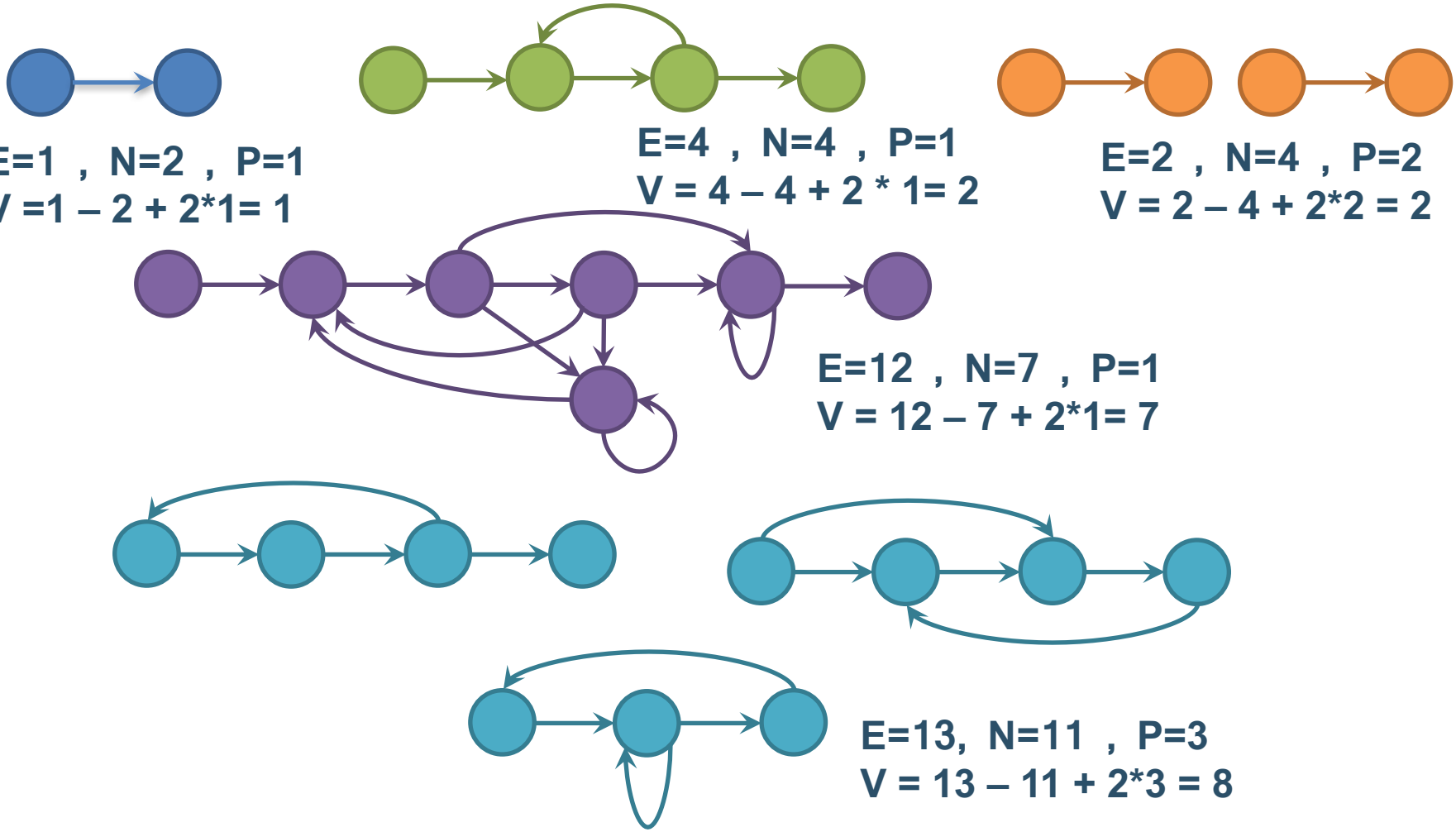
Computation of cyclomatic complexity

- Cyclomatic complexity V(*G*) for a control flow graph *G*, is defined as:
- $\mathbf{V(G) = E - N + 2P}$
  E: number of edges
  N: number of nodes
  P: number of disconnected partitions of the graph
- Simplifications
- No decisions: V(G) = 1
- Only *b* binary decisions: V(G) = *b* + 1

Calculate cyclomatic complexity



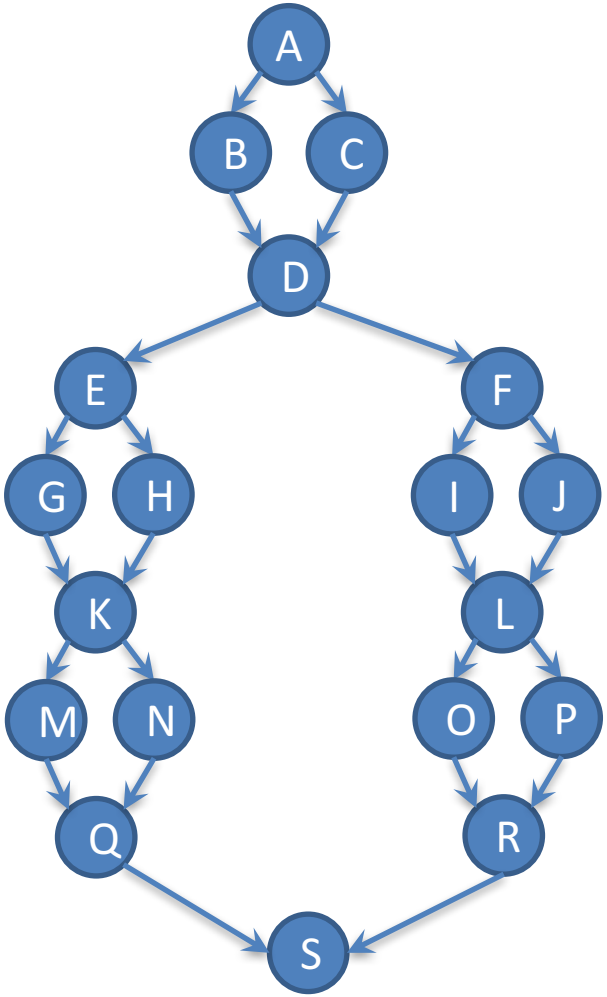
Calculate cyclomatic complexity



Basis Path Testing

- Derive the control flow graph from the software module
- Compute the cyclomatic complexity of the resultant flow graph
- **Determine a basis set of linearly independent paths**
- Create a test case for each basis path
- Execute these tests

Basis path testing: cyclomatic complexity



V(G) = E – N + 2P

E=

N=

P=

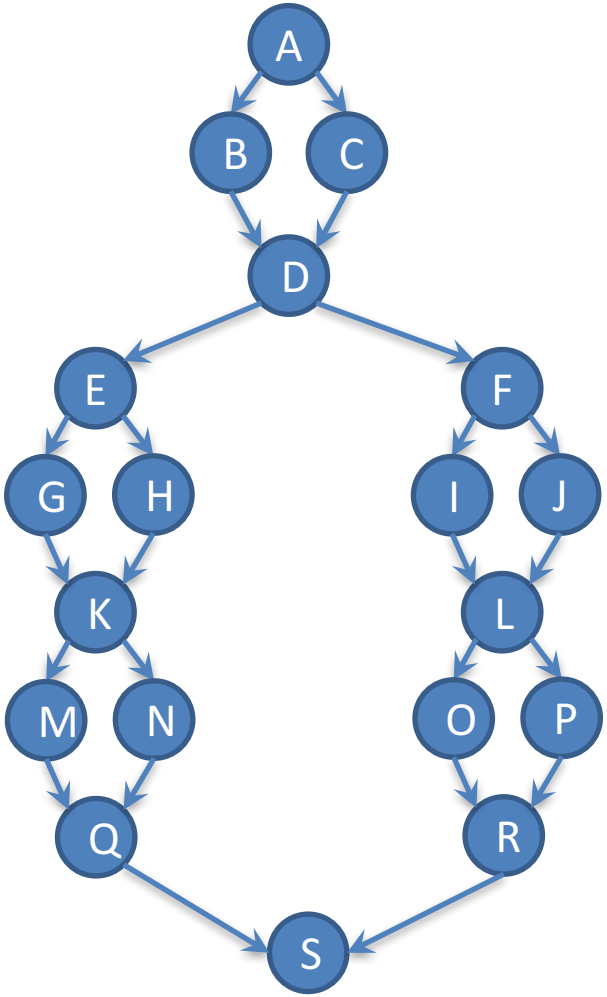
V(G) =

V(G) = b + 1

b=

V(G) =

Basis path testing: cyclomatic complexity



V(G) = E – N + 2P

E = 24

N = 19

P=1

V(G) = 24 – 19 + 2*1 = 7

V(G) = b + 1

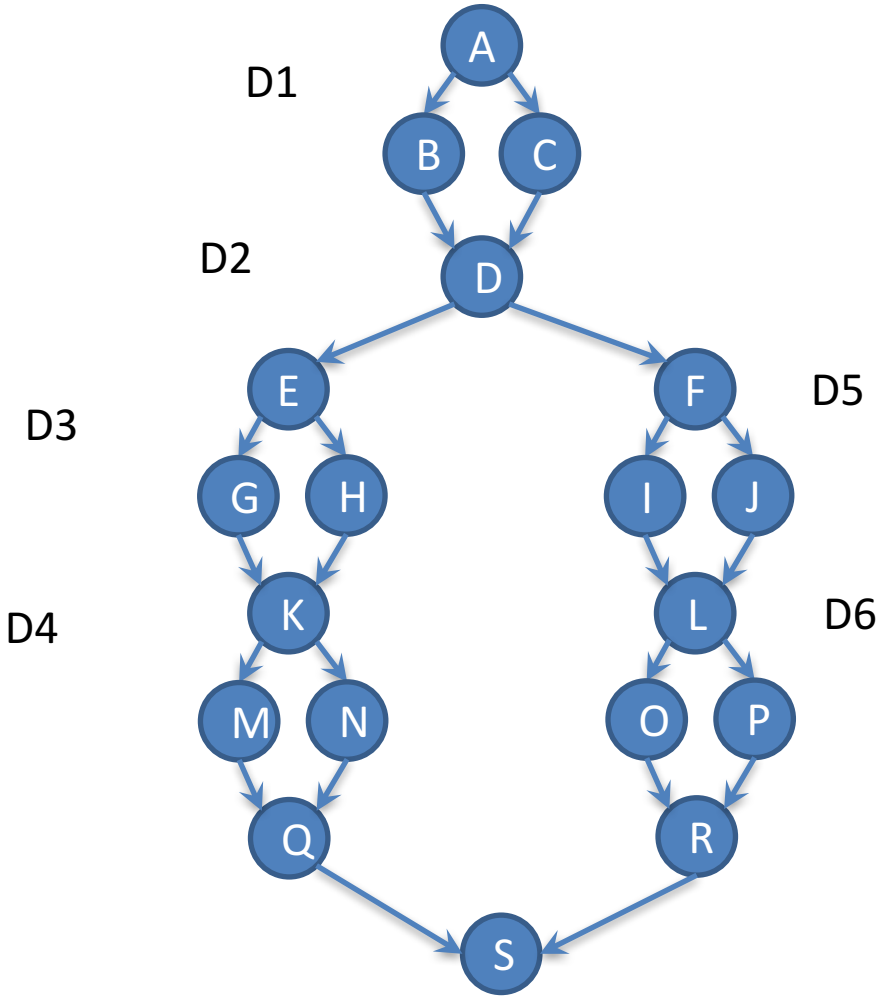
b= 6

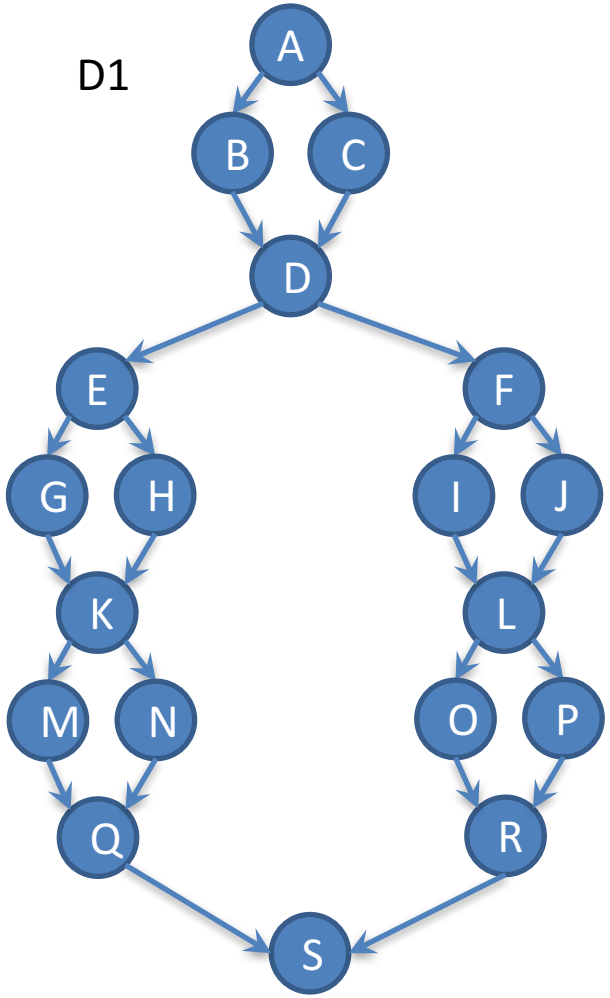
V(G) = 6 + 1 = 7

Determine a basis set of linearly independent paths McCabe's baseline method

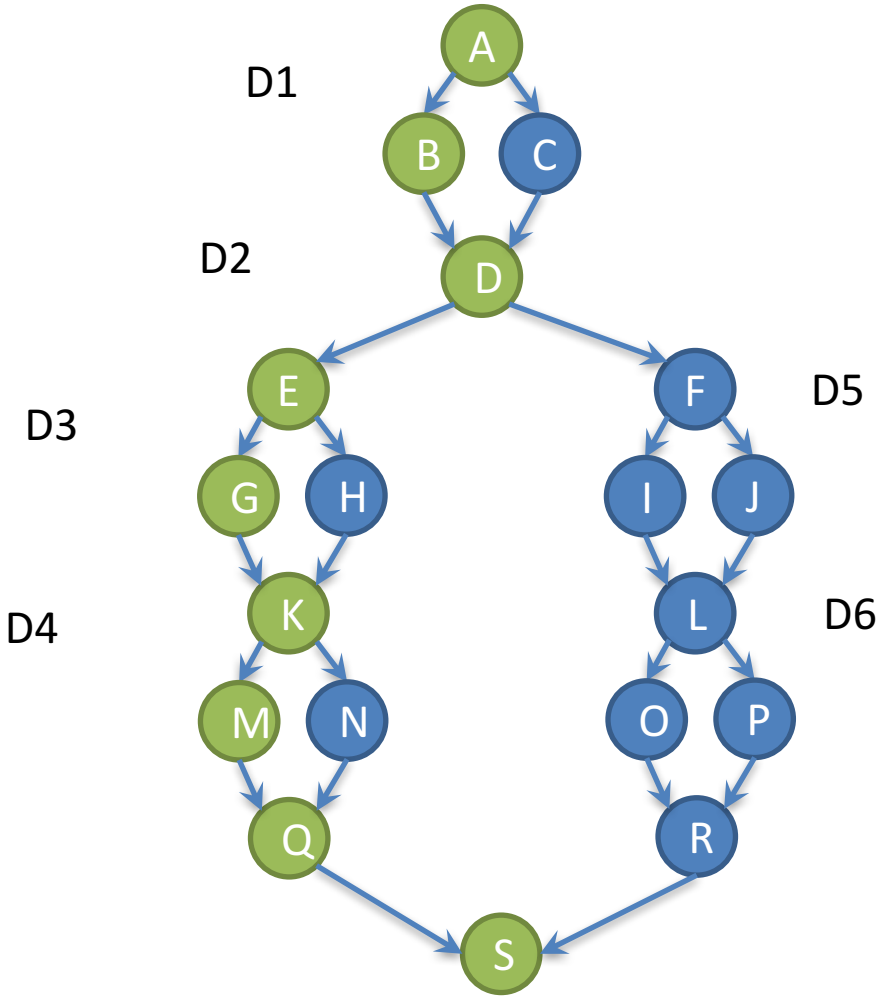
1. Pick a **"baseline"** path. This path should be a **"normalcase"** program execution. McCabe advises: **choose a path with as many decisions as possible**.
2. To choose the **next path**, change the outcome of the first decision along the baseline path while keeping the maximum number of other decisions the same as the baseline path.
3. To generate the **third path**, begin again with the base line but vary the second decision rather than the first.
4. Repeat step 3 for other paths until all decisions along baseline path have been flipped.
5. Now proceed to the **second path**(from step 2), flipping its decisions, one by one until the basis path set is completed.

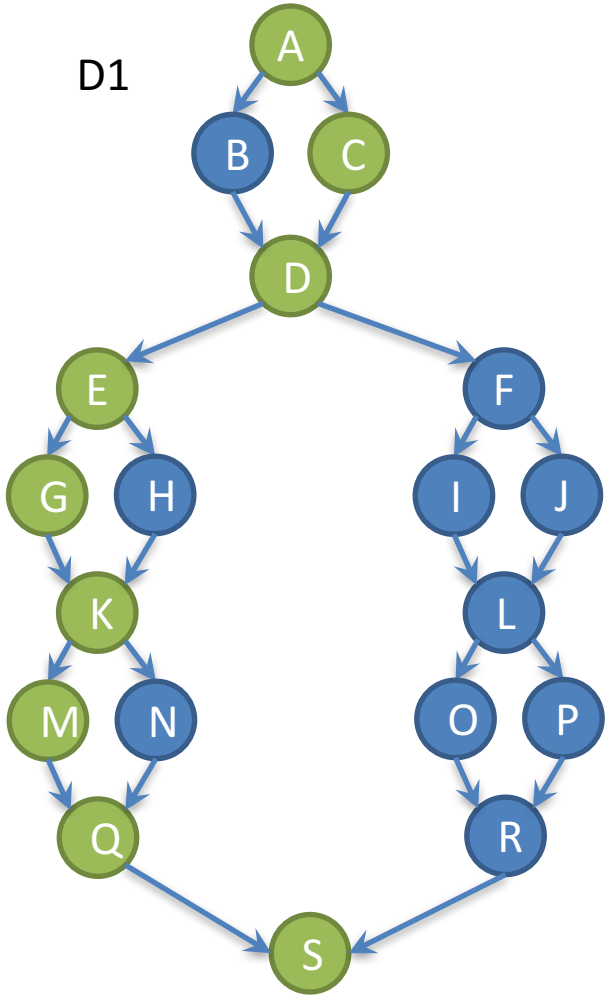
Create the basis set: paths 1-2



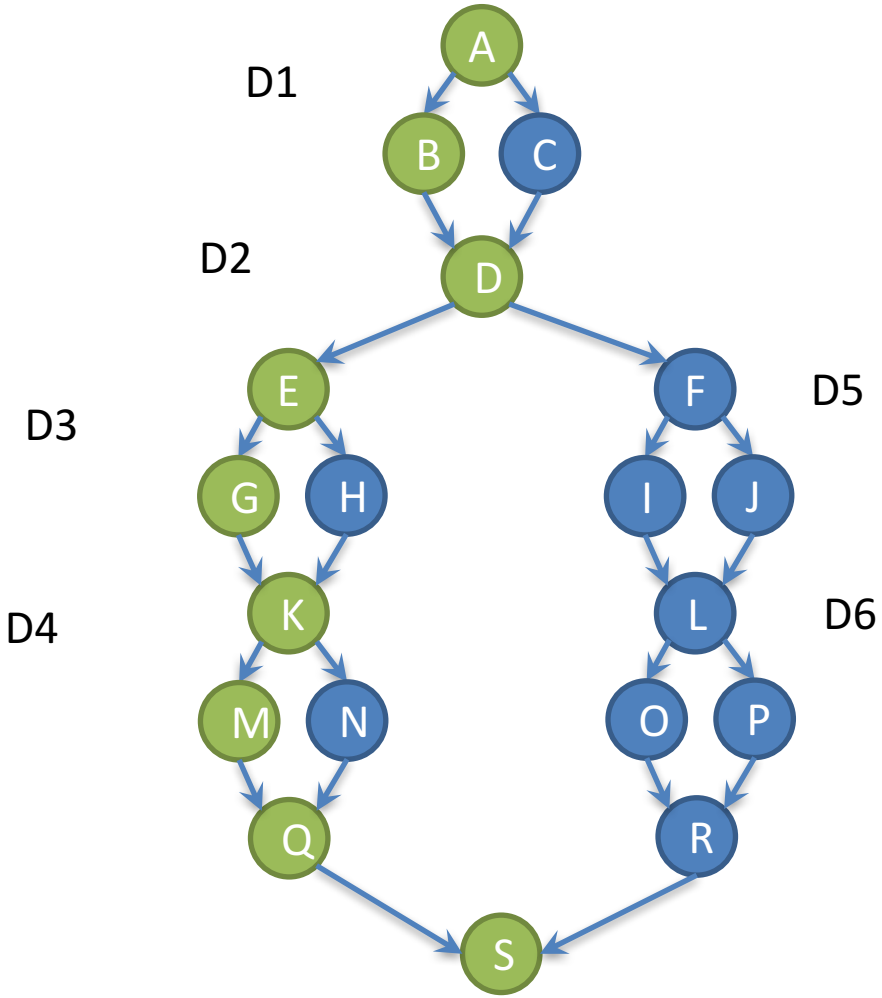


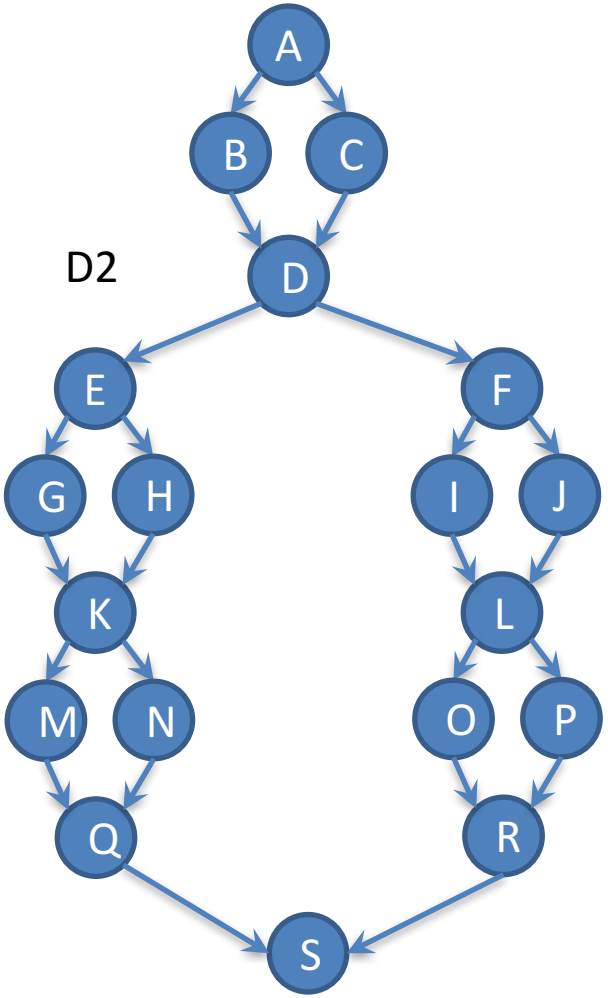
Create the basis set: paths 1-2



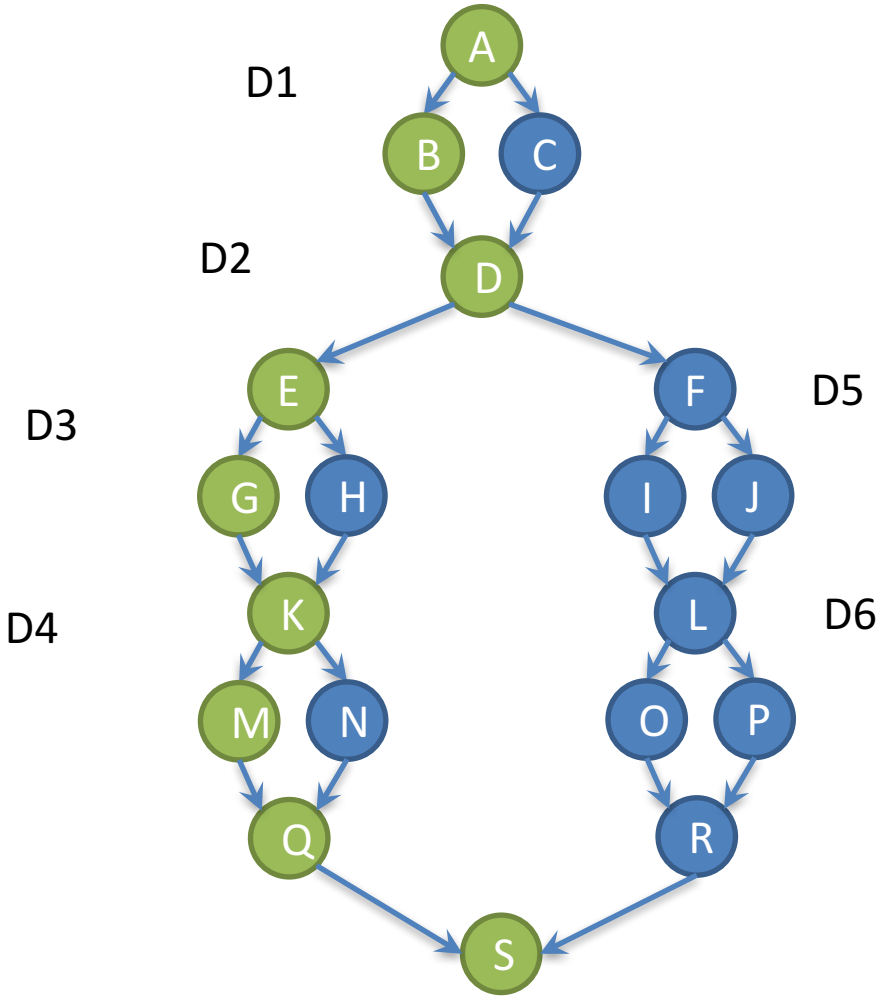


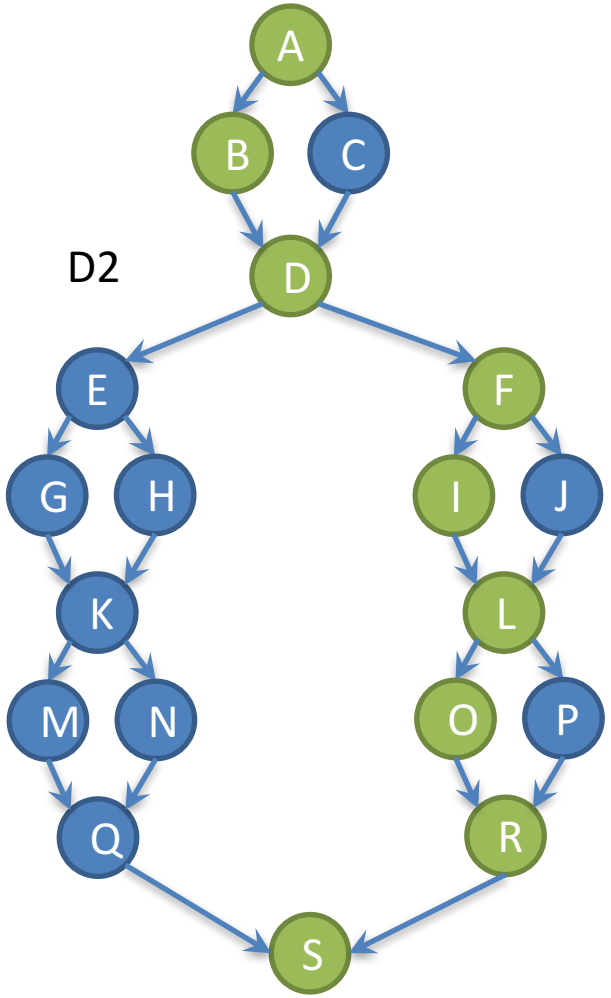
Baseline + second decision (path 3)



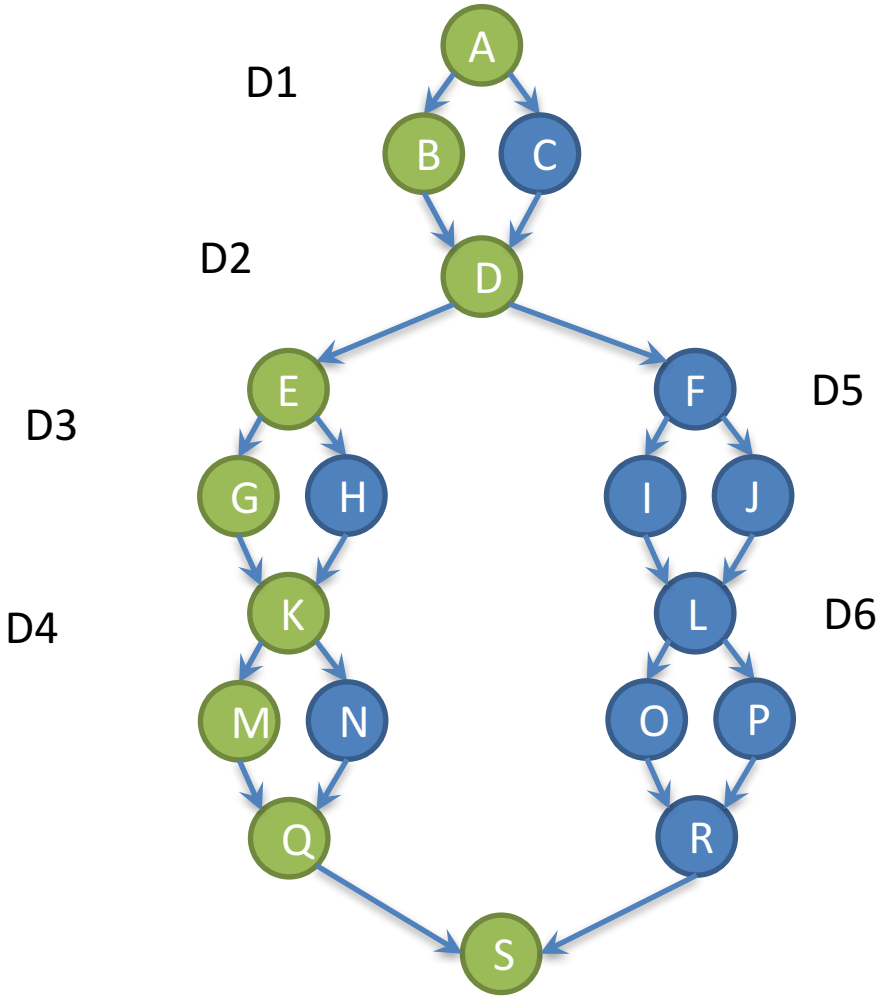


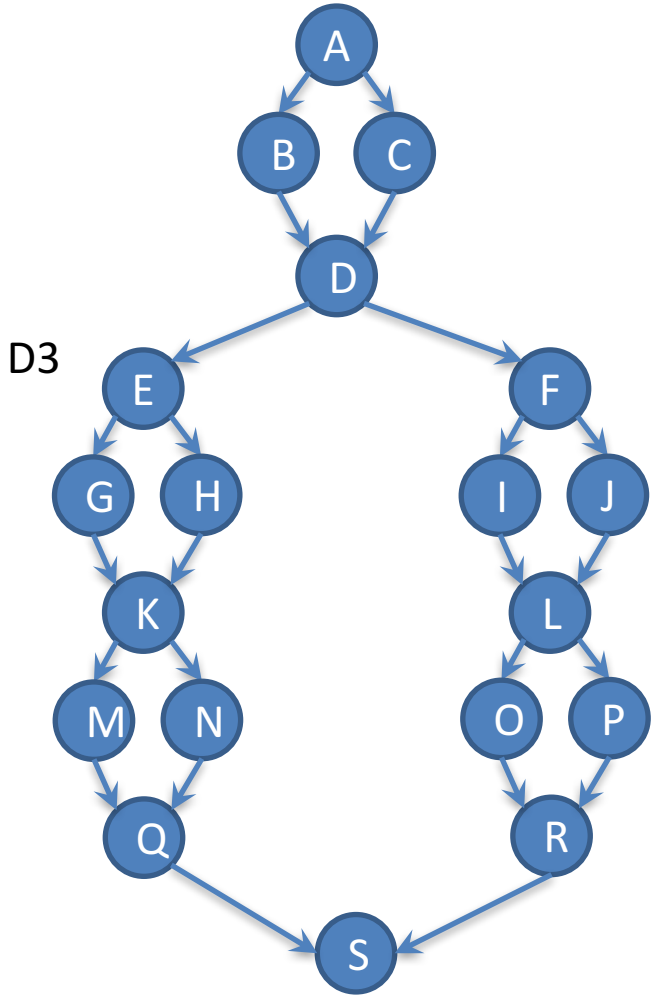
Baseline + second decision (path 3)



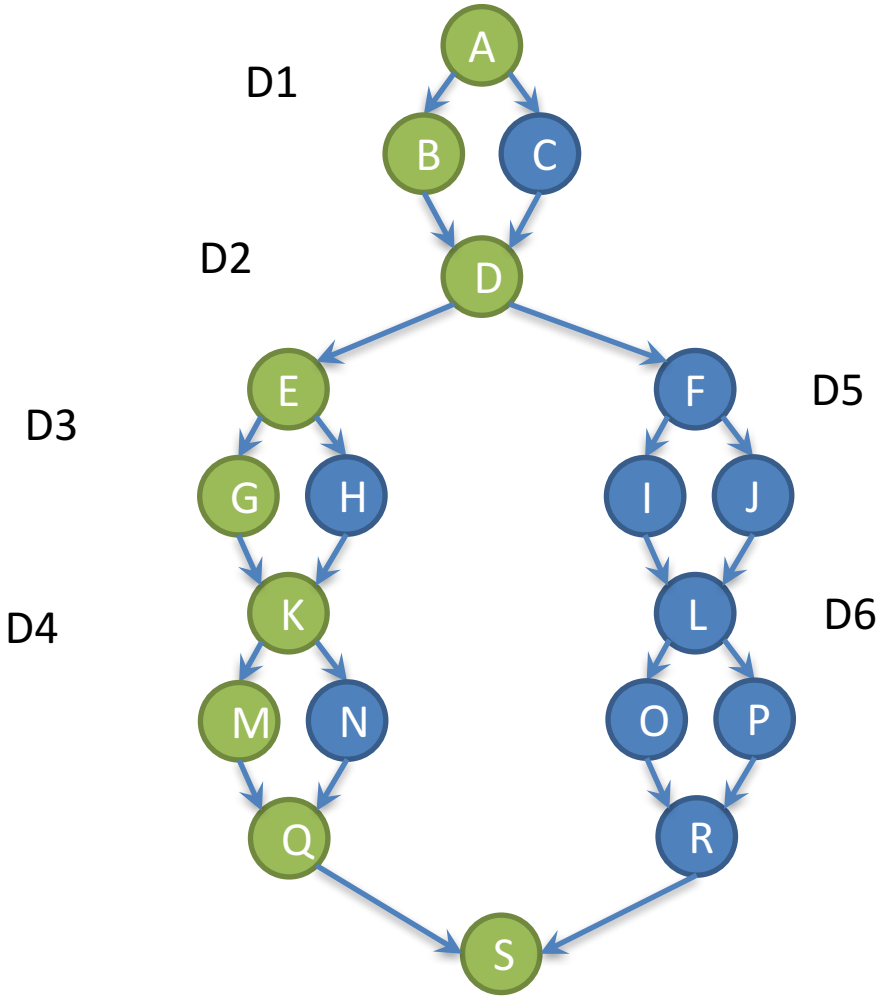


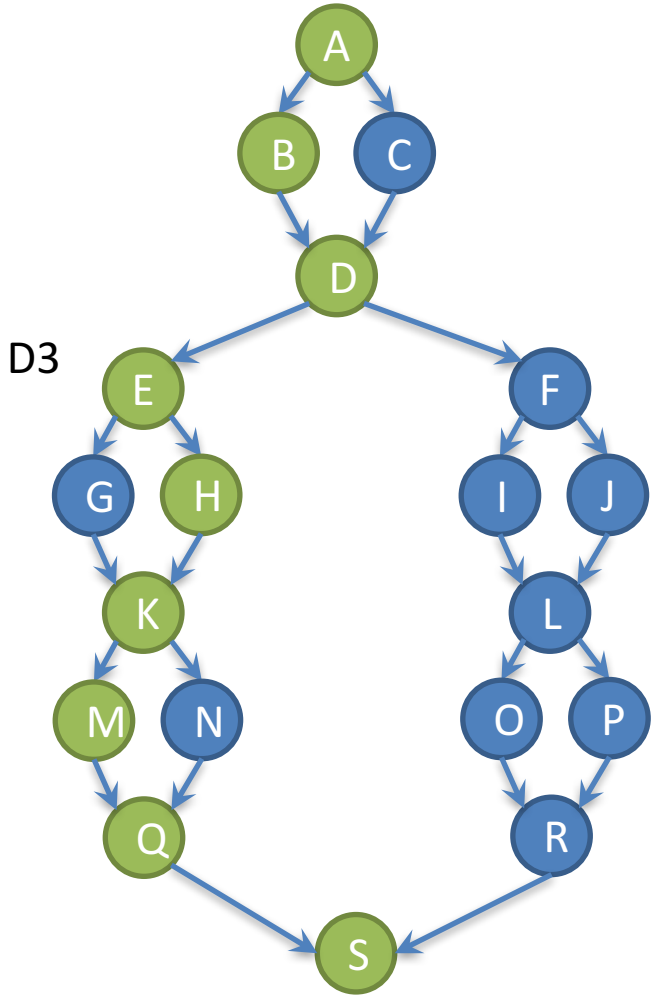
Baseline + third decision (path 4)



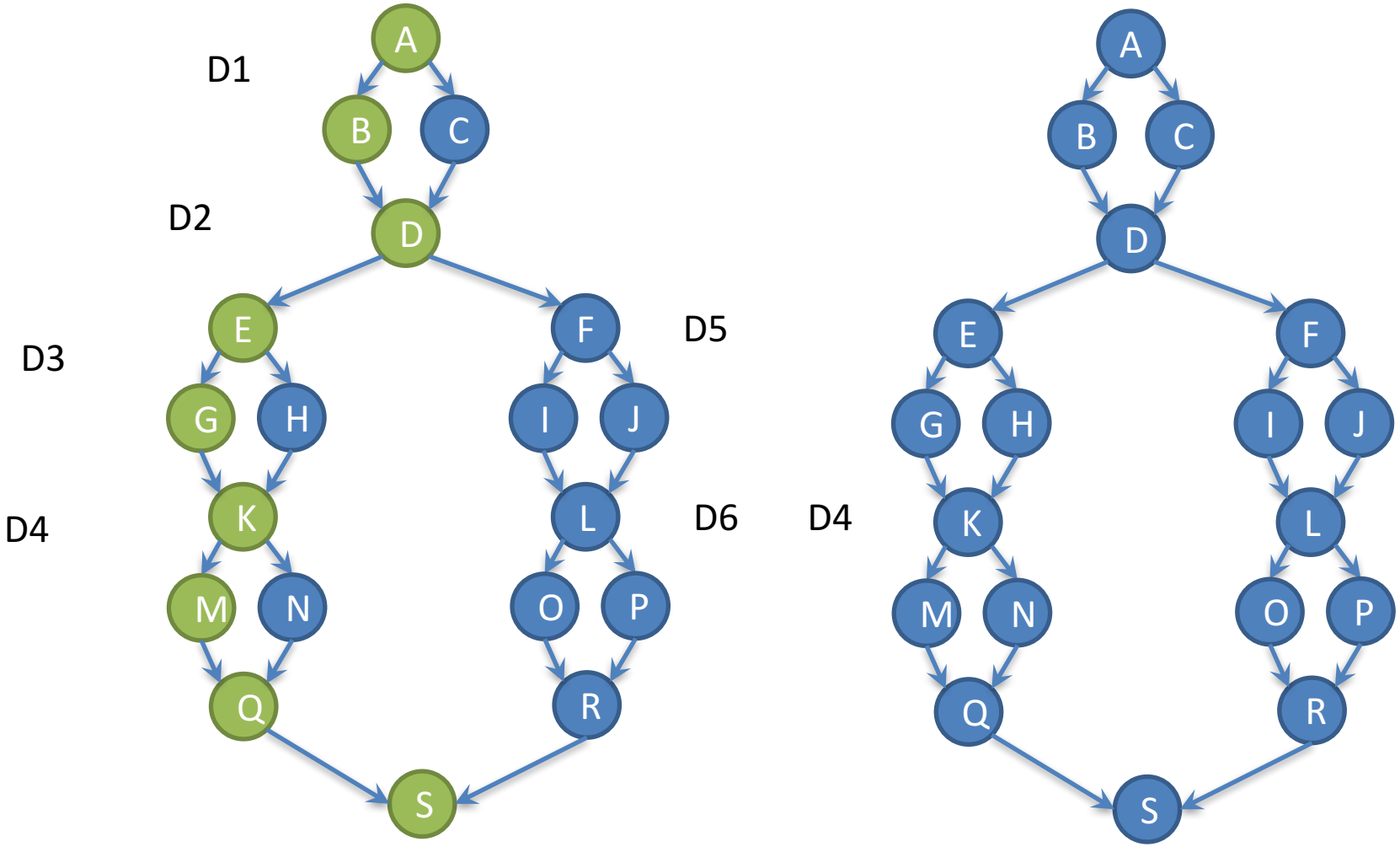


Baseline + third decision (path 4)

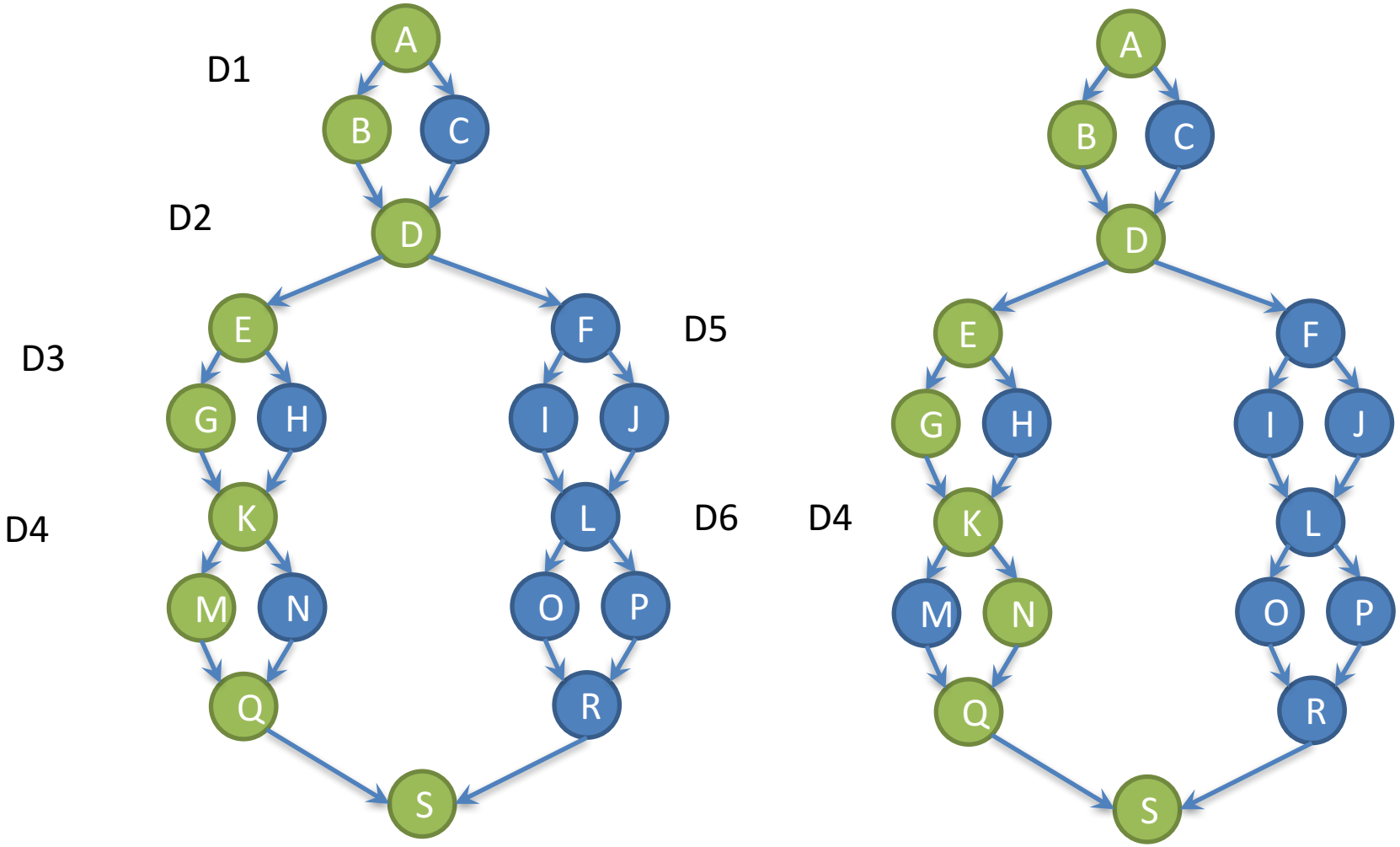




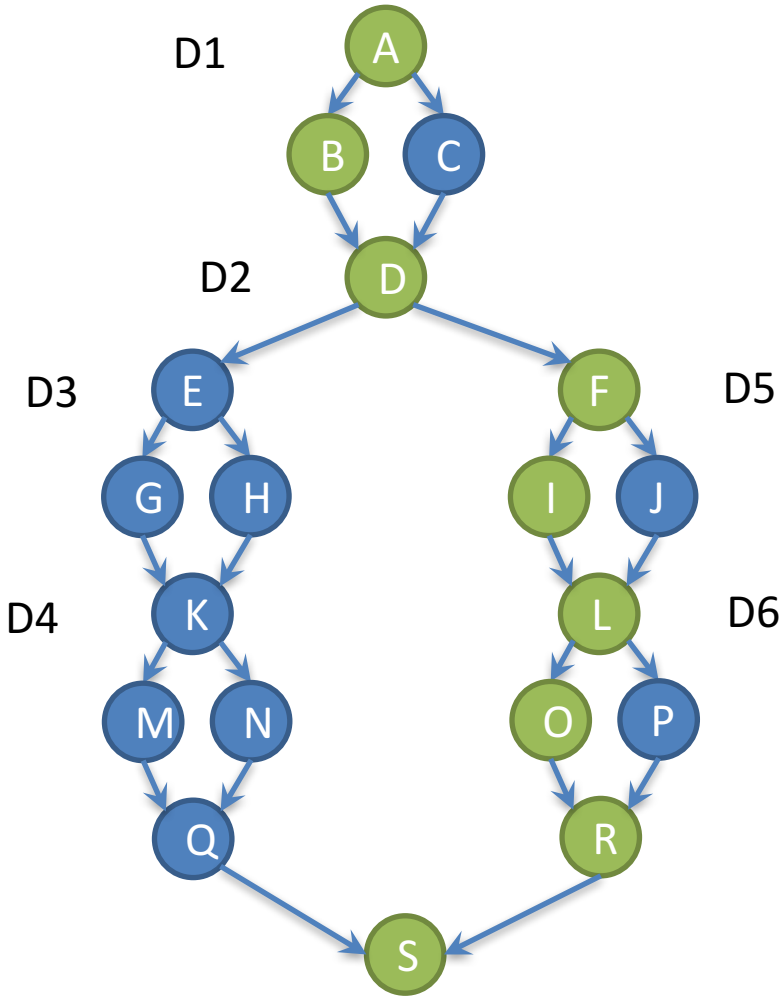
Baseline + fourth decision (path 5)

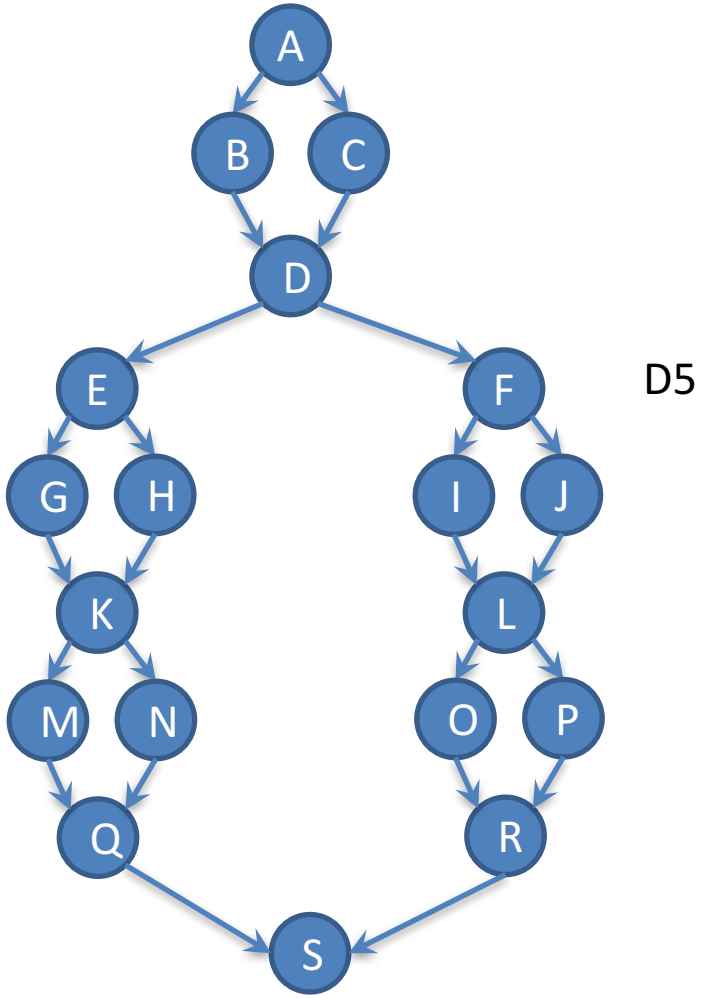


Baseline + fourth decision (path 5)

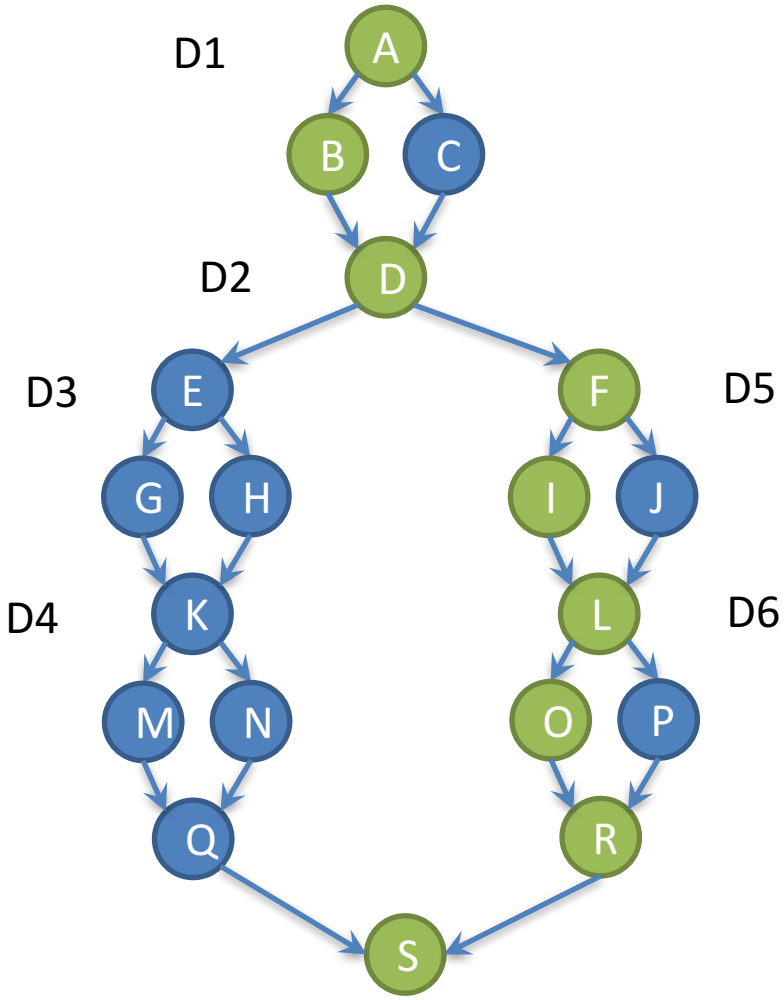


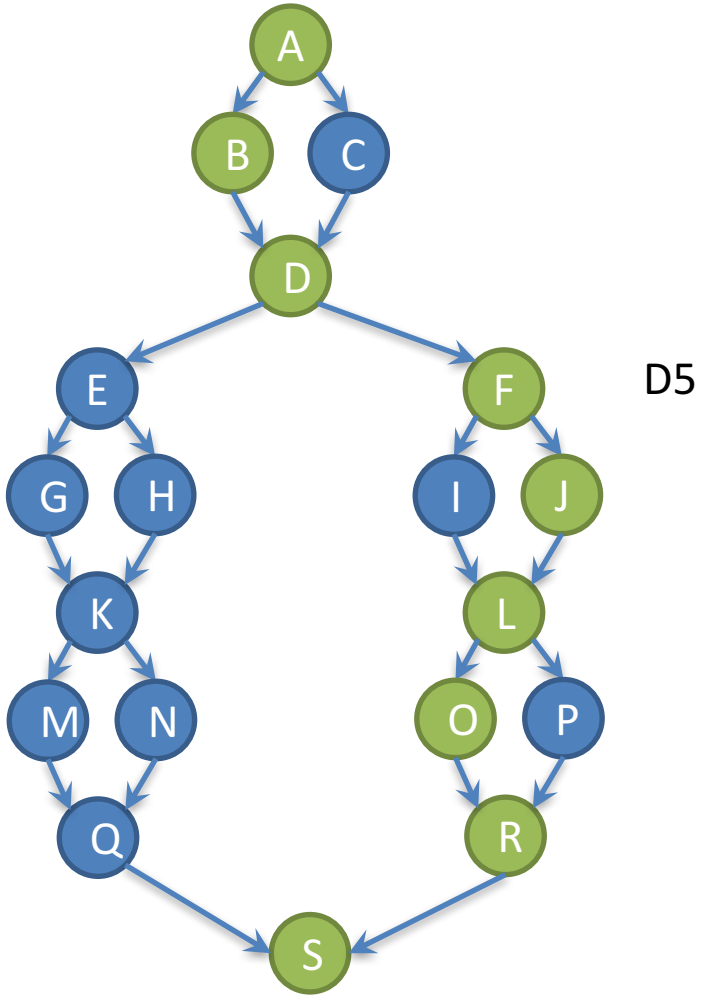
Third path + fifth decision (path 6)



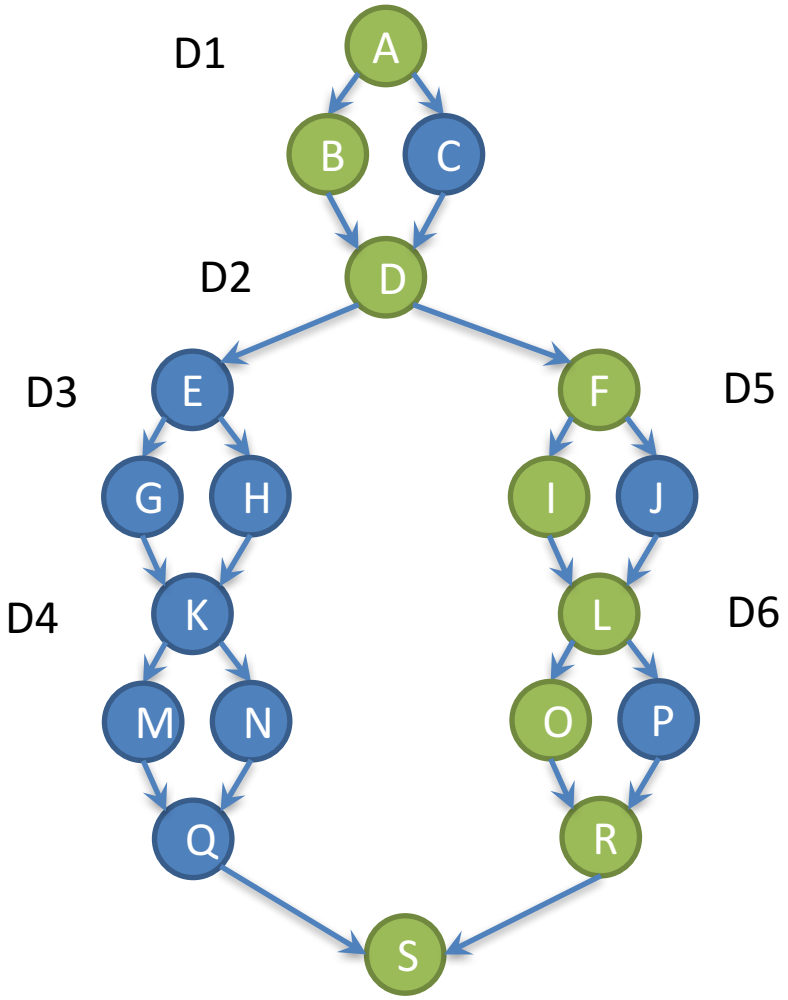


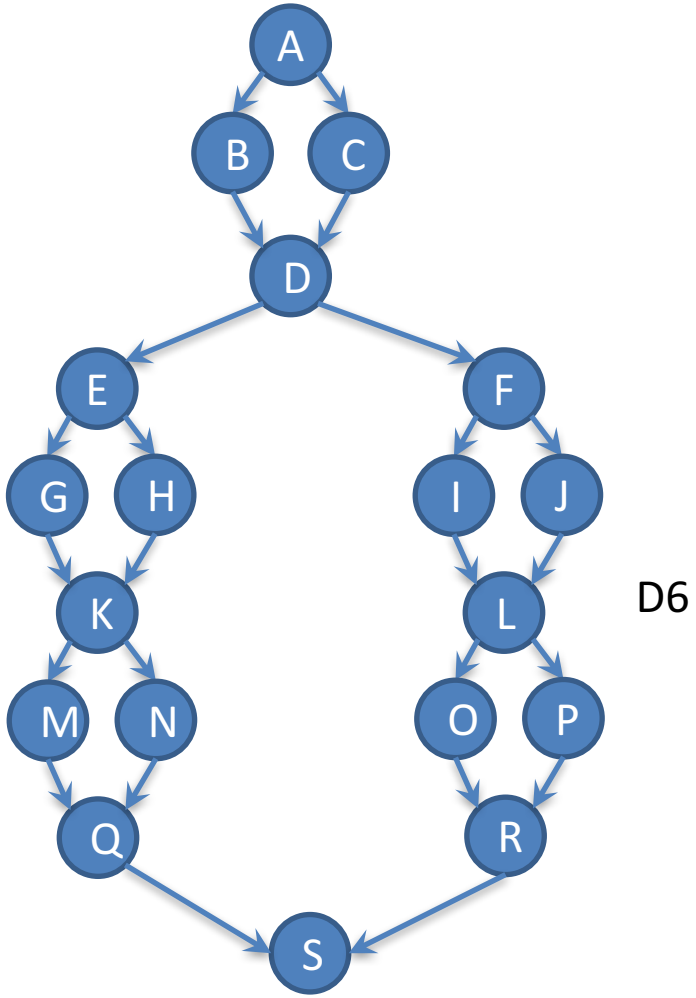
Third path + fifth decision (path 6)



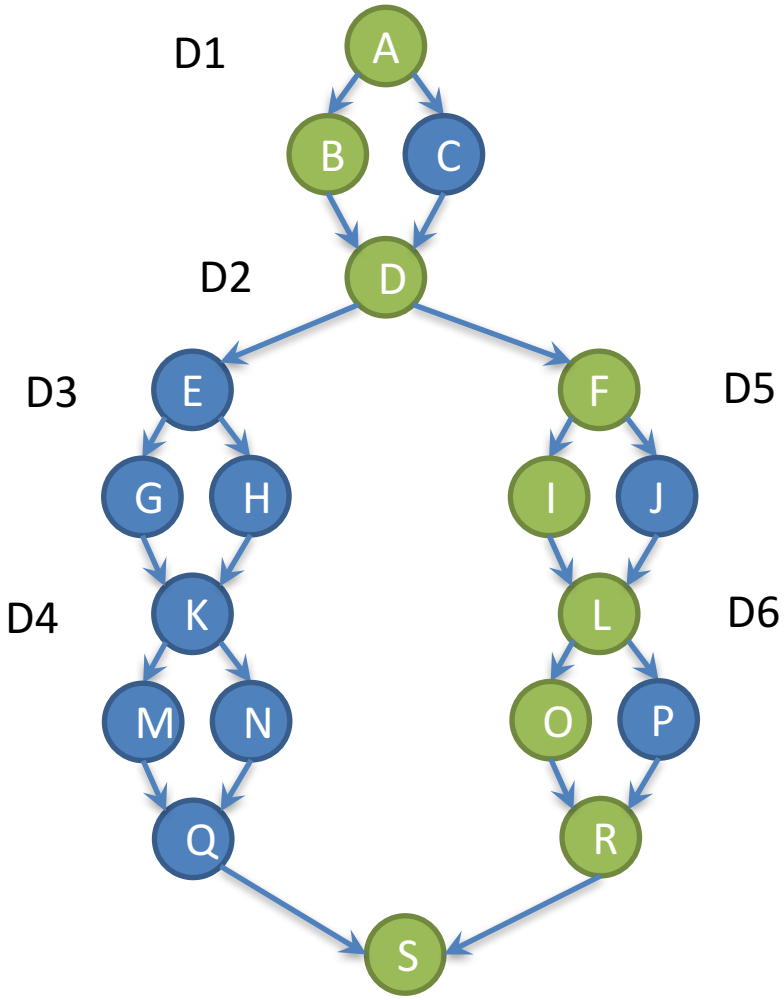


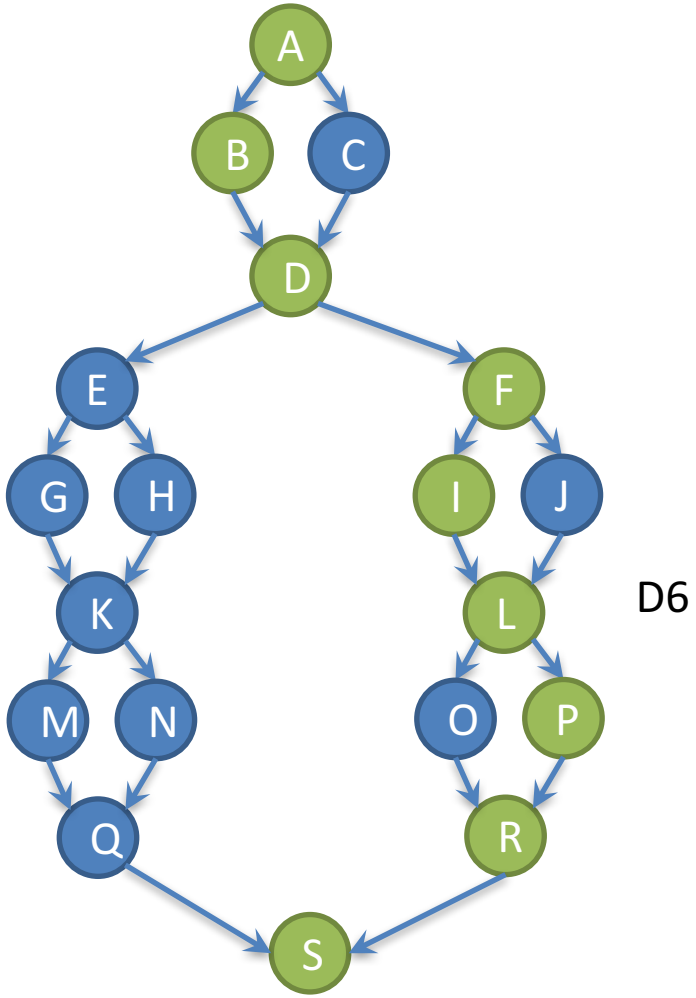
Third path + sixth decision (path 7)



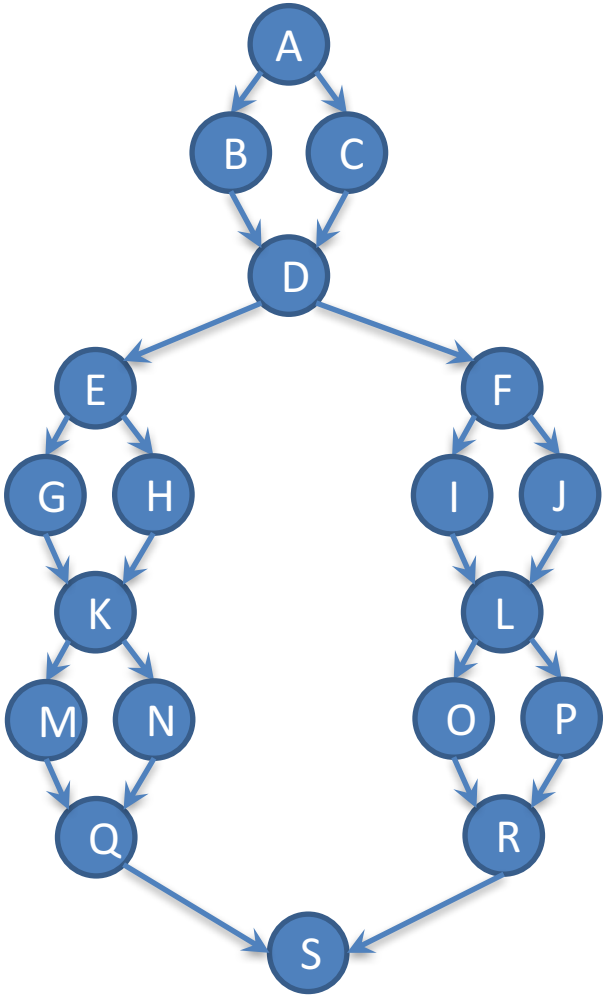


Third path + sixth decision (path 7)





Set of basis paths



1. ABDEGKMQS
2. ACDEGKMQS
3. ABDFILORS
4. ABDEHKMQS
5. ABDEGKNQS
6. ACDFJLORS
7. ACDFILPRS

Observation

- Basis path testing calls for the **creation of a test case** for each of these
- paths.
- This set of test cases will **guarantee** both **statement and branch**
- **coverage**.
- Using a path/edge indicence matrix, independent paths becomes linearly independent vectors, spanning the vector space of possible paths

Why coverage?

"We found that **there is a low to moderate correlation between coverage and effectiveness** when the number of test cases in the suite is controlled for. In addition, we found that **stronger forms of coverage do not provide greater** insight into the **effectiveness** of the suite. Our results suggest that **coverage**, while useful for identifying under-tested parts of a program, should not be used as a quality target because it **is not a good indicator of test suite effectiveness**." [1]

[1] L. Inozemtseva and R. Holmes. *Coverage is not strongly correlated with test suite effectiveness*. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

Why is Coverage insufficient?

Remember this example?

```
int scale(int j) {
    j = j - 1; // should be j = j + 1
    j = j / 30000;
    return j;}
```

Would coverage testing find this bug?

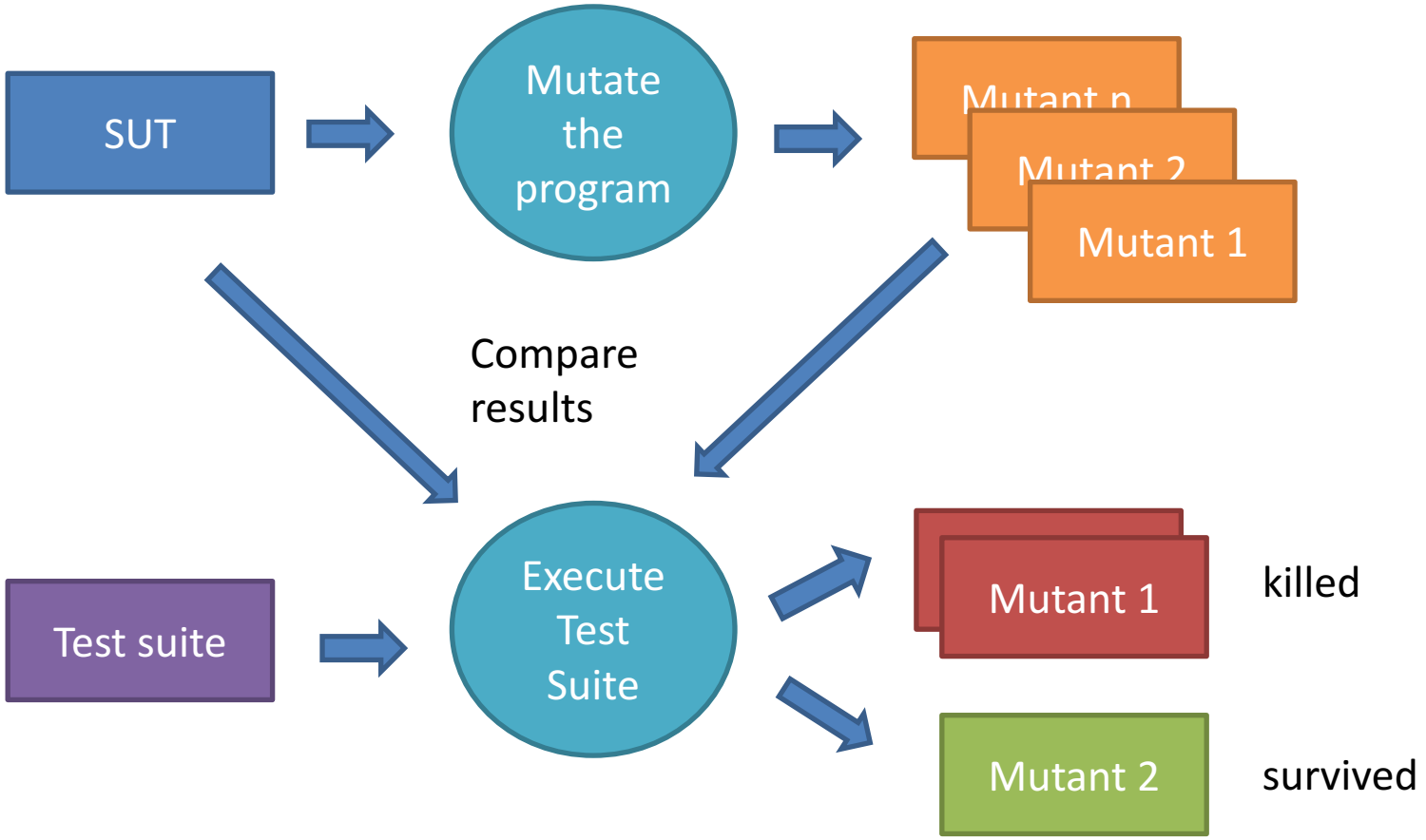
Example with branch coverage:

```
public static String foo(boolean b) {
        if ( b ) {
                performVitallyImportantBusinessFunction();
                return "OK"; }
        return "FAIL"; }

@Test public void shouldFailWhenGivenFalse() {
        assertEquals("FAIL", foo(false)); }

@Test public void shouldBeOkWhenGivenTrue() {
        assertEquals("OK", foo(true)); }
```

Mutation testing



Mutation testing

- Mutation operators
- Conditionals Boundary Mutator
- Negate Conditionals Mutator
- Remove Conditionals Mutator
- Math Mutator
- Increments Mutator
- Invert Negatives Mutator
- Inline Constant Mutator
- Return Values Mutator
- Void Method Calls Mutator
- Non Void Method Calls Mutator

Conditionals Boundary Mutator

Original conditional	Mutated conditional
<	<=
<=	<
>	>=
>=	>

For example

```
if (a < b) { // do something } ‘
```

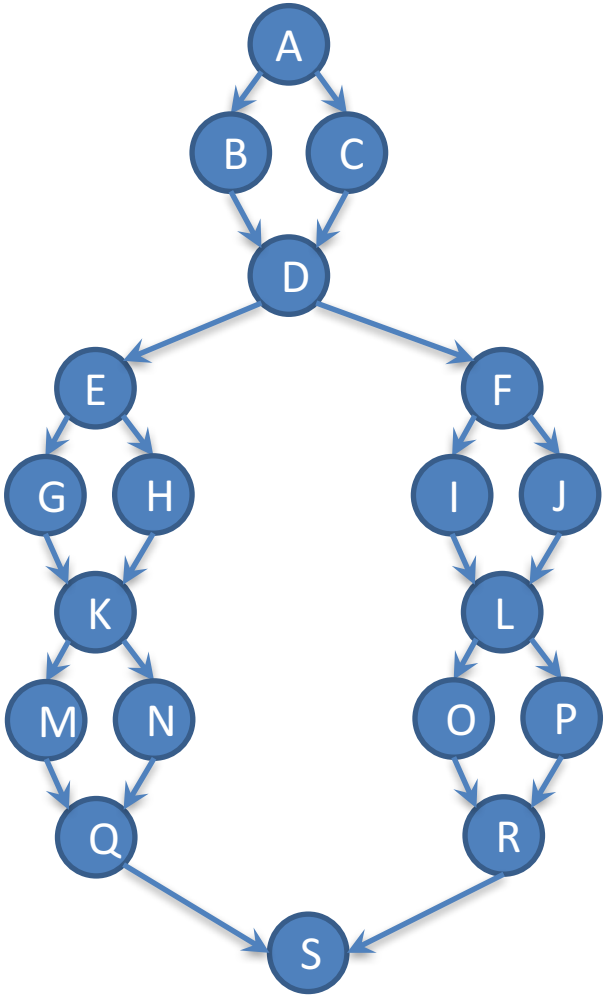
will be mutated to

```
if (a <= b) { // do something }
```

Equivalent mutations

- The resulting mutant behaves the same as the original
- The difference in behavior is outside the scope of testing

Automating white-box testing



- Creating a representation – abstract syntax tree (AST)
- Generating a symbolic execution model
- Putting the right constraints on the inputs

Automation of white-box testing: Java Pathfinder

Lab on symbolic execution

Next lecture:

- Guest lecture from Ahmed Rezne on model checking

Questions?

Thank you for your attention

LINKÖPING UNIVERSITY