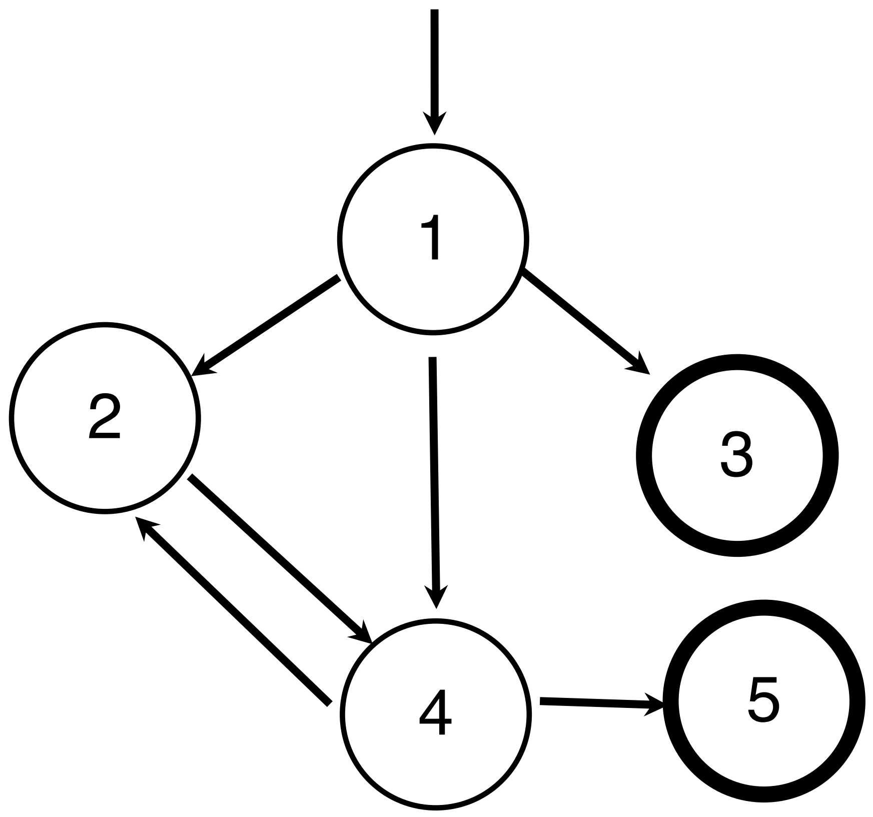
Graph-based test coverage (part 1)

- Graph definitions
- Structural graph coverage criteria
- Coverage of control flow graphs for source code
- Coverage of call graphs
- FSM abstractions for software / FSM test coverage

Ciências ULisboa

slides: Eduardo Marques, Vasco Vasconcelos, Francisco Martins, João Neto

Graph coverage



- Approach
  - A graph representation models the software under test (SUT).
  - The execution of a test case corresponds to a path in the graph.
  - Coverage criteria specify requirements as sets of paths that must be covered by test paths.

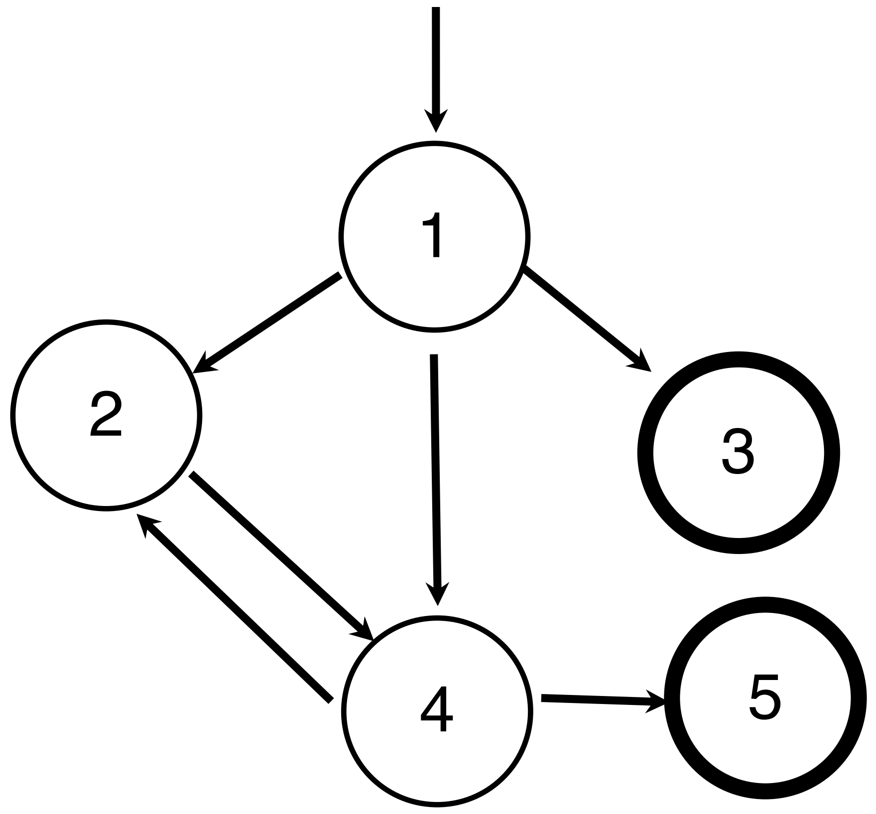
Test graph

$N = \{1, 2, 3, 4, 5\}$

$N_0 = \{1\}$

$N_f = \{3, 5\}$

$E = \{(1,2), (1,3), (1,4), (2,4), (4,2), (4,5)\}$



- A test graph is a tuple $G = (N, N_0, N_f, E)$ where:
  - N is a non-empty set of **nodes**
  - $N_0 \subseteq N$ is a non-empty set of **initial** (also called entry) nodes
  - $N_f \subseteq N$ is a non-empty set of **final** (exit) nodes
  - $E \subseteq N \times N$ is a set of **edges**
  - For $(a,b) \in E$ we say a is a **predecessor** of b and b is a **successor** of a.

Definitions

- A **path** is a sequence $p = [n_0, n_1, \ldots, n_k]$ such that n_{i+1} is a successor of n_i for $i=0, \ldots, k-1$.
  - The **length of the path** is k (paths with just one node have length 0).
  - A subsequence of p is called a **subpath** of p.
  - A path is called a **cycle** if $n_0 = n_k$.
  - A path **contains a cycle** if one of its subpaths is a cycle.
- A **test path** is a path $p = [n_0, n_1, \ldots, n_k]$ such that $n_0 \in N_0$ and $n_k \in N_F$, i.e., a test path starts with an entry node and ends with an exit node.

[2] is a path with length 0.

[2,4] is path with length 1.

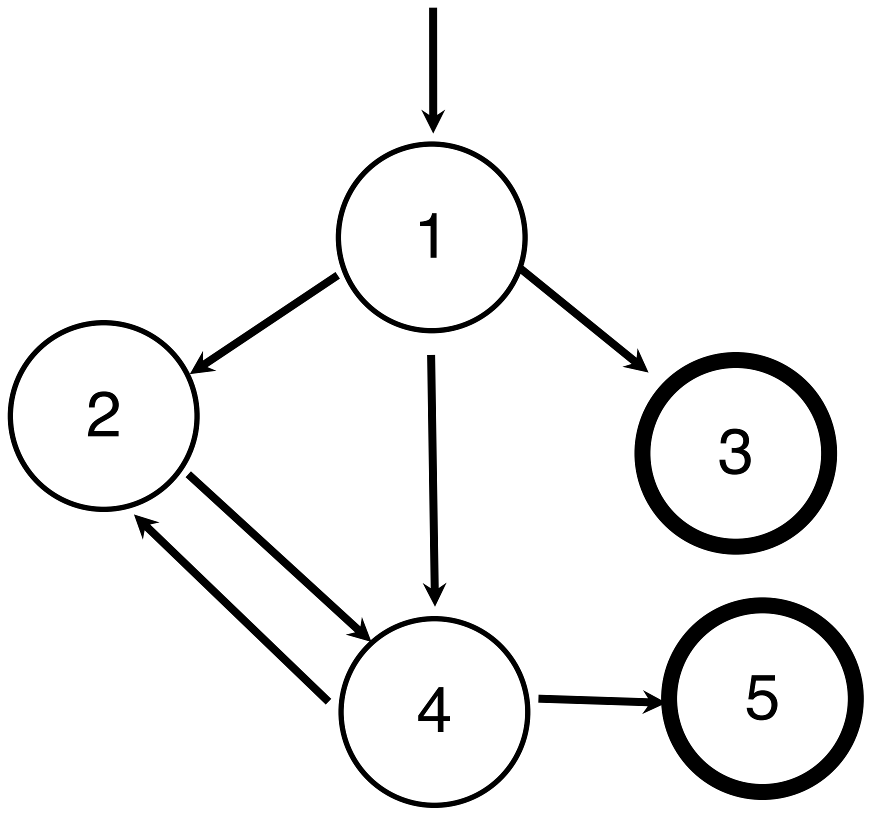
[2,4] is a subpath of [1,2,4].

[2,4,2] is a cycle.

[2,4,2,4,5] contains a cycle.

[1,2,4,5] is a test path.

[1,3] and [1,2,4,2,4,5] are also test paths.

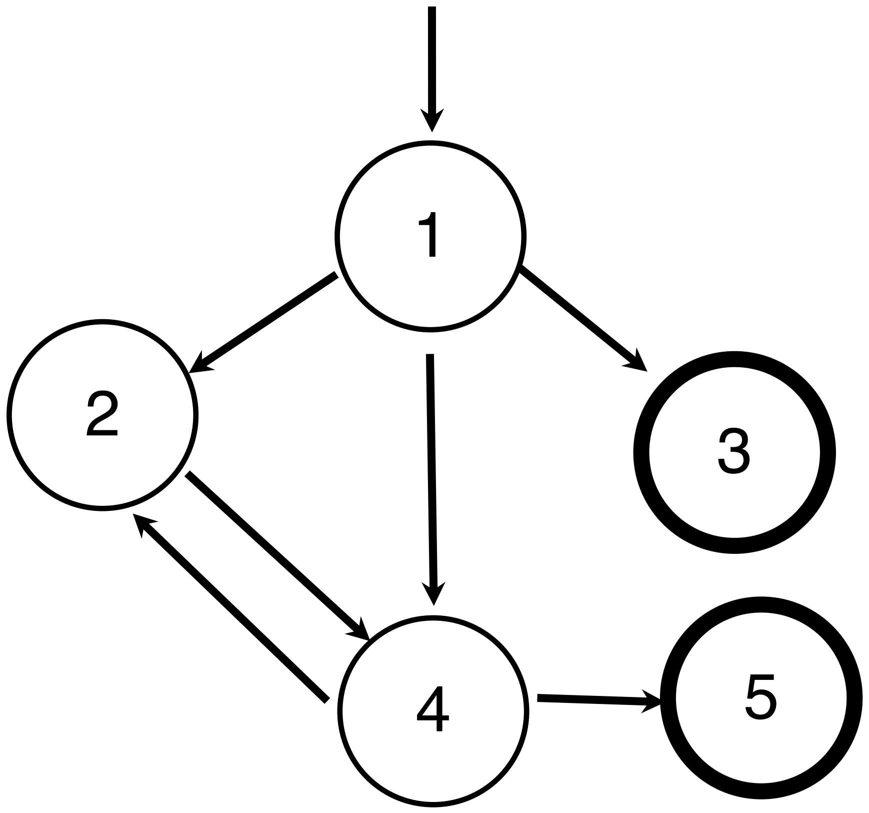


Coverage criteria for graphs

- A **test criterion** C is a set of rules that impose requirements for test paths.
- **Test requirements** TR(C) are expressed as paths or subpaths of the graph at stake.
- The execution of a test case t results in a test path path(t).
- A set of test cases T **satisfies** C if and only if for every requirement $r \in TR(C)$ there is a test $t \in T$ that tours (or covers) r, i.e., r is a subpath of path(t).

Node and edge coverage (NC, EC)

- **Node coverage (NC)**
  - Test requirements: cover every node (all graph paths up of length 0)
  - **TR(NC)** = set of nodes in the graph
- **Edge coverage (EC):** cover every edge.
  - Test requirements: cover every edge (all paths up to length 1)
  - **TR(EC)** = set of edges in the graph
- **EC subsumes NC.** Why?



TR(NC) = {[1], [2], [3], [4], [5]}

TR(EC) = TR(NC) U {[1,2], [1,3], [1,4], [2,4], [4,2], [4,5]}

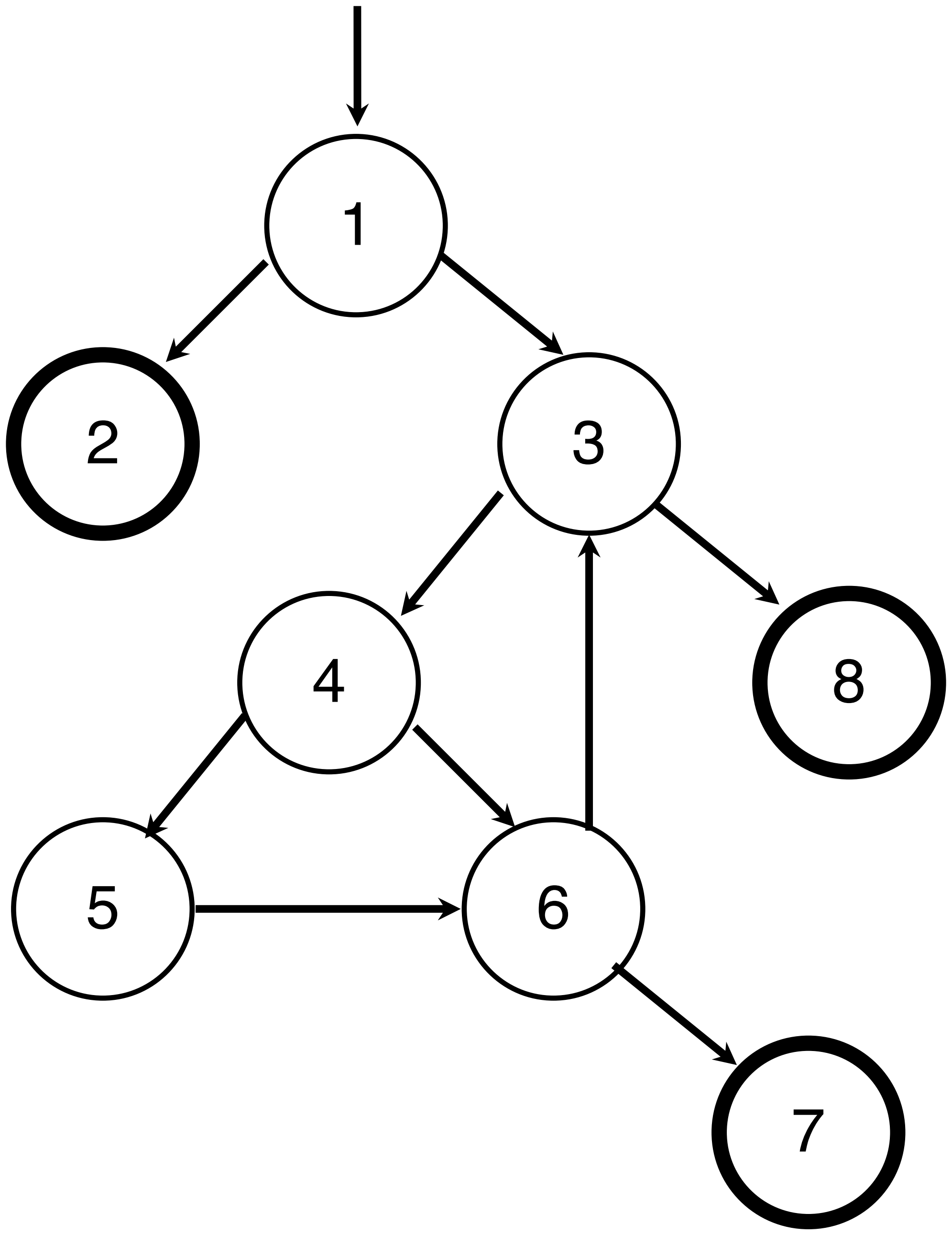
T1 = {[1,3], [1,2,4,5]} satisfies NC, but not EC

T2 = {[1,3], [1,2,4,5], [1,4,2,4,5]}
satisfies both NC and EC

Exercise 1

For the graph shown right:

1. Identify TR(NC), TR(EC)
2. Identify test paths that satisfy NC but not EC.
3. Identify test paths that satisfy EC.
4. Is it is possible to satisfy EC using only test paths without cycles?



Control flow graph (CFG)

- A **control flow graph** (**CFG**) can be used to represent the control flow of a piece of (imperative) source code.
  - **Nodes** represent **basic blocks** - sequences of instructions that always execute together in sequence.
  - **Edges** represent **control flow** between basic blocks.
  - The **entry node** corresponds to a method's entry point.
  - **Final nodes** correspond to exit points, e.g. in Java: `return` or `throw` instructions.
  - **Decision nodes** represent choices in control flow - e.g. in Java: `if`, `switch-case` blocks, condition tests for loops.

Example

```
public static int occurrences(char[] v, char c) {
  if (v == null)
      throw new IllegalArgumentException();
  int n = 0;
  for (int i = 0; i < v.length; i++)
      if (v[i] == c)
        n++;
  return n;
}
```

Basic blocks (nodes)

```
1: if (v == null)
2: throw ...;
3: n = 0; i = 0;
4: i < v.length;
5: if (v[i] == c)
6: n++;
7: i++;
8: return n;
```

Entry node

1

Decision nodes

1, 4, 5

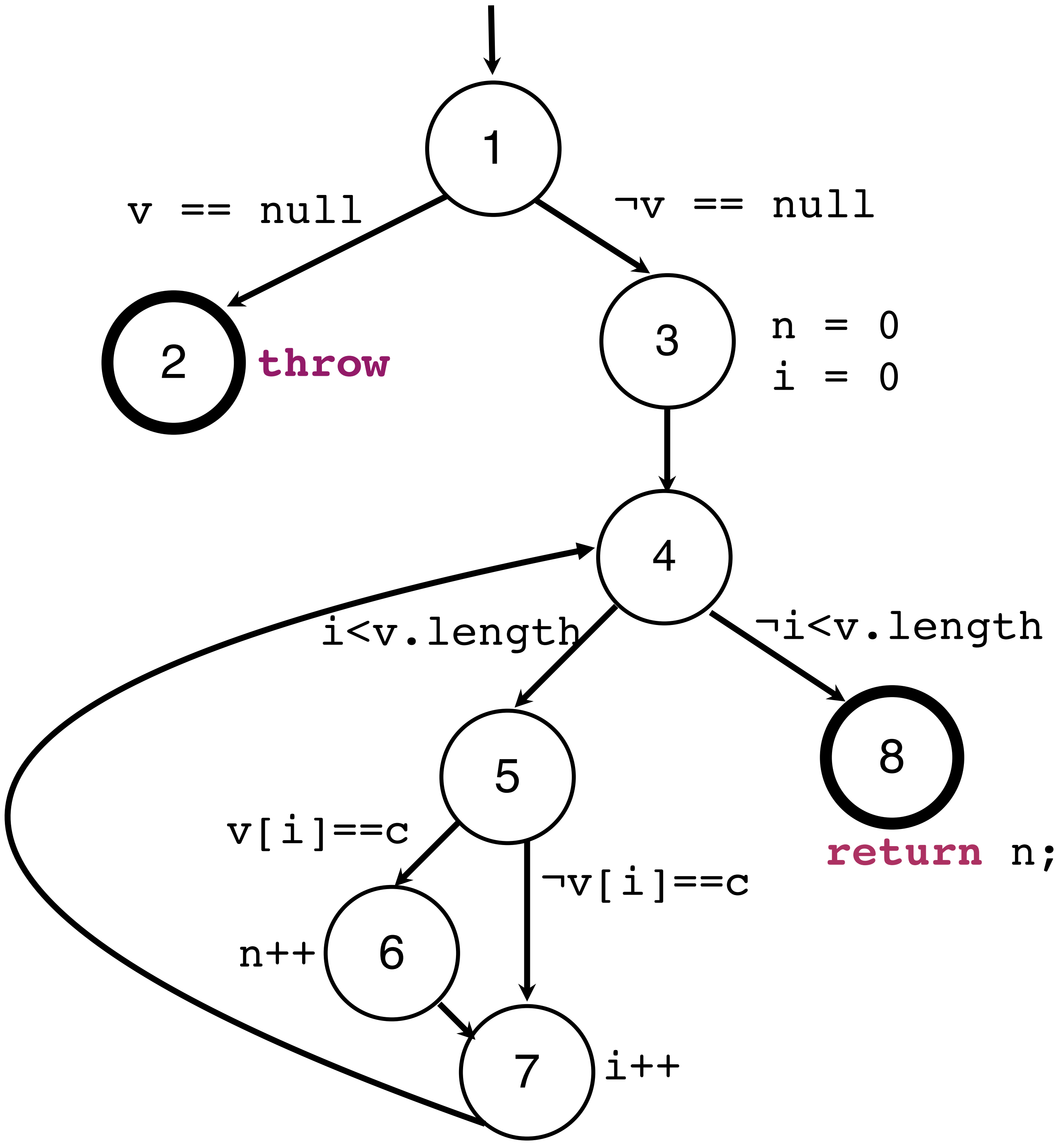
Exit nodes

2, 8

Control flow (edges)

$1 \rightarrow 2$,	$1 \rightarrow 3$
$3 \rightarrow 4$	
$4 \rightarrow 5$,	$4 \rightarrow 8$
$5 \rightarrow 6$,	$5 \rightarrow 7$
$6 \rightarrow 7$	
$7 \rightarrow 4$	

CFG for `occurrences()`

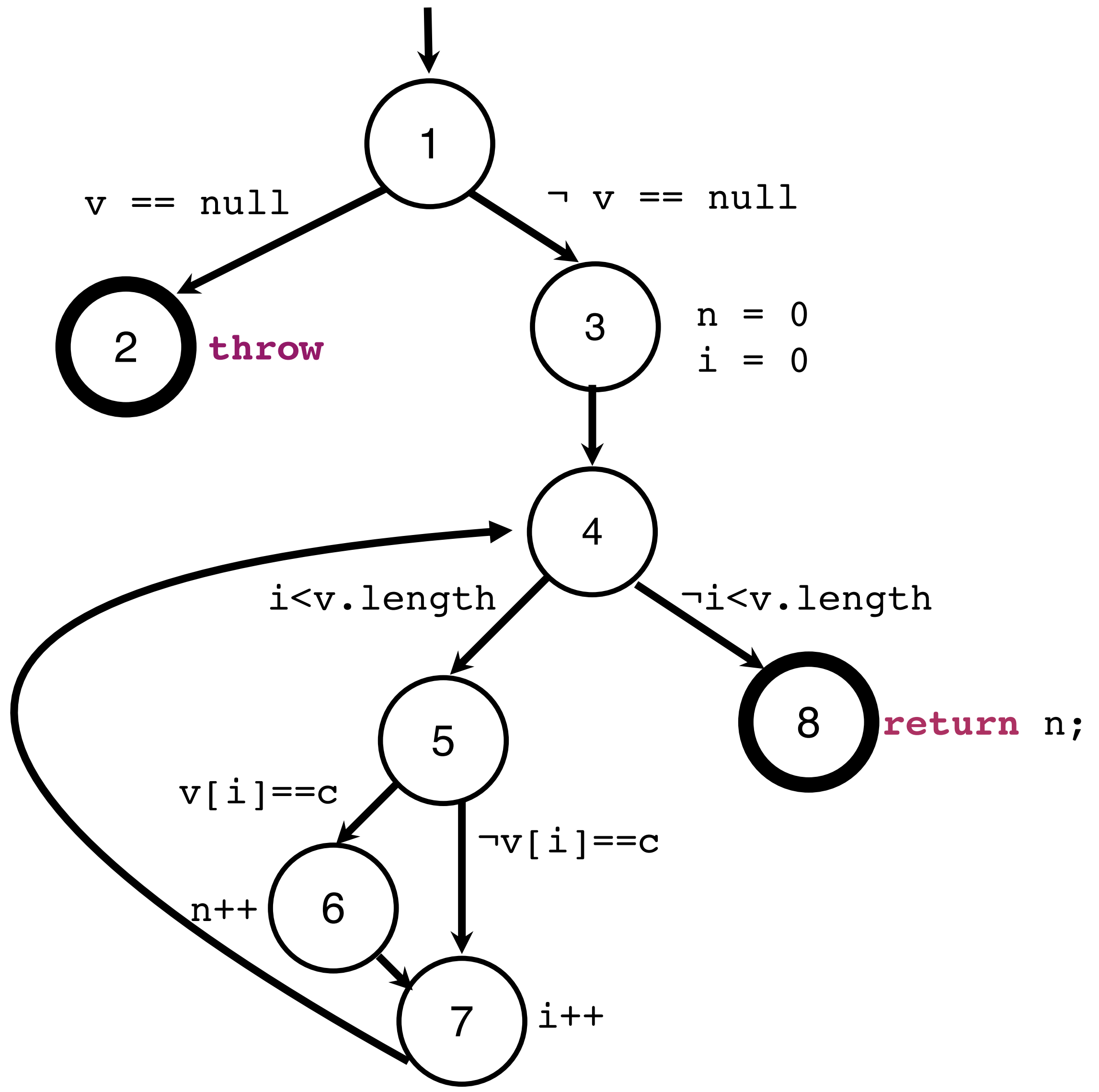


Basic blocks (nodes)

```
1: if (v == null)
2: throw ...;
3: n=0; i=0;
4: i < v.length;
5: v[i] == c;
6: n++;
7: i++;
8: return n;
```

Control flow (edges)

```
1 → 2,    1 → 3
3 → 4
4 → 5,    4 → 8
5 → 6,    5 → 7
6 → 7
7 → 4
```



Node coverage

TR(NC) = {[1], [2], [3], [4], [5], [6], [7], [8]}

NC satisfied by {t1, t2} or {t1, t3}

Edge coverage

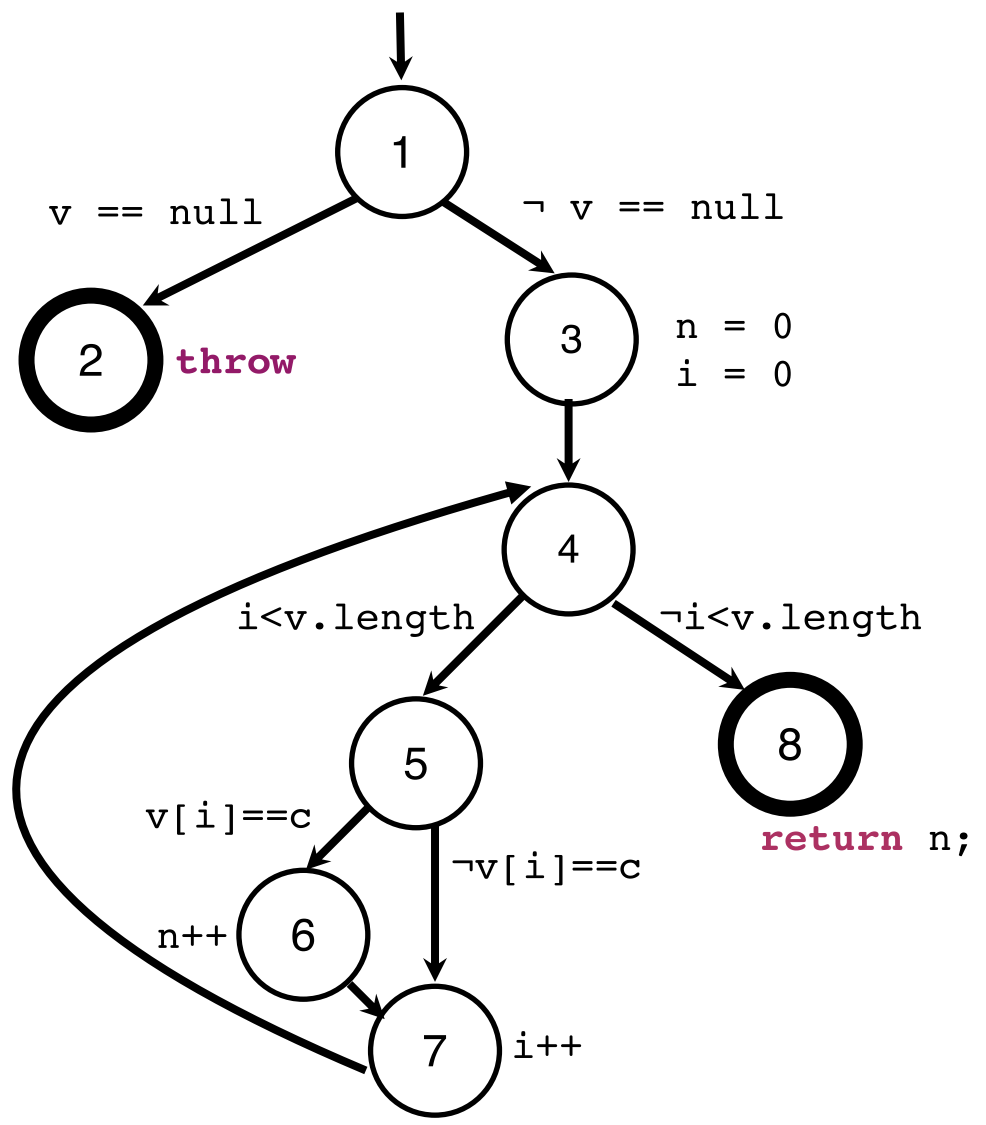
TR(EC) = TR(NC) U {[1,2], [1,3], [3,4], [4,5], [4,8], [5,6], [5,7], [6,7], [7,4]}

EC satisfied by {t1, t3} but not by {t1,t2}.

t	test case values (v,c)	exp. Value	test path	nodes covered	edges covered
t1	`(null, 'a')`	`IArgExc`	[1,2]	1, 2	[1,2]
t2	`({'a'}, 'a')`	1	[1,3,4,5,6,7,4,8]	1, 3, 4, 5, 6, 7, 8	[1,3], [3,4], [4,5], [5,6], [6,7], [7,4], [4,8]
t3	`({'x', 'a'}, 'a')`	1	[1,3,4,5,7,4,5,6,7,4,8]	1, 3, 4, 5, 6, 7, 8	[1,3], [3,4], [4,5], [5,6], [6,7], [7,4], [5,7], [4,8]

Edge-pair coverage

- **Edge-pair coverage** (**EPC**) - cover all paths up to length 2
  - **EPC** subsumes **NC** and **EC**



Edge-pair coverage

TR(EPC) = TR(EC) U{[1,3,4], [3,4,5], [3,4,8], [4,5,6], [4,5,7], [5,6,7], [5,7,4], [6,7,4], [7,4,5], [7,4,8]}

EC satisfied by {t1, t3}.

EPC satisfied by {t1, t2, t3} but not by {t1,t3}.
Observe that t3 does not cover path [3,4,8].

t	test case values (v, c)	exp. Value	test path	requirements covered
t1	`(null, 'a')`	`IArgExc`	`[1,2]`	`[1,2]`
t2	`({}, 'a')`	`0`	`[1,3,4,8]`	`[1,3,4][3,4,8]`
t3	`({'x', 'a'}, 'a')`	`1`	`[1,3,4,5,7,4,5,6,7,4,8]`	`[1,3,4][3,4,5][4,5,7]` `[5,7,4][7,4,5][4,5,6]` `[5,6,7][7,4,8]`

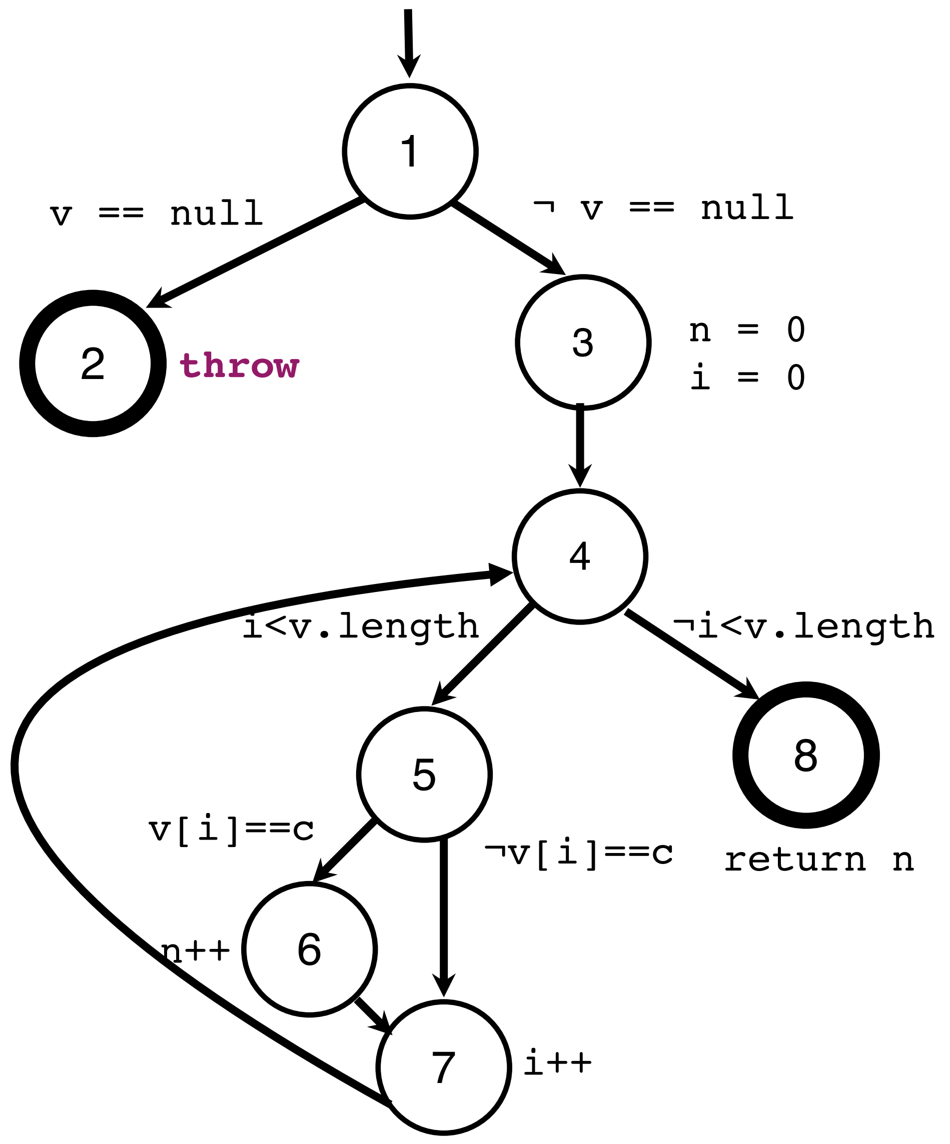
Beyond node/edge coverage

- NC, EC, EPC are instances of the general criterion: cover all paths up to length k
    - NC for k=0; EC for k=1; EPC for k=2;
- As k increases we approximate **Complete-Path-Coverage** (**CPC**)
    - **CPC:** Cover all possible paths.
    - The number of paths may be infinite (e.g., CFGs with loops), i.e., CPC is not applicable in most cases.
    - In practice, instead of “increasing k”, we should try to pick a subset of “relevant” paths in the graph. How?

Prime Path Coverage (PPC)

- A path $p = [n_1, n_2, ..., n_M]$ is a **simple path** if no node appears more than once, other than possibly the first and last ones.
  - A simple path has no internal loops, but may represent a loop if the first and last nodes are equal.
- A **prime path** is a maximal length simple path, i.e., a simple path that is not a proper subpath of any other simple path.
- **PPC requires every prime path to be covered by the test set.**

Finding prime paths in a graph



[1]
—
[2]!
—
[3]
—
[4]
—
[5]
—
[6]
—
[7]
—
[8]!

[1,2]!
[1,3]
—
[3,4]
—
[4,5]
[4,8]!
—
[5,6]
[5,7]
—
[6,7]
—
[7,4]

[1,3,4]
—
[3,4,5]
[3,4,8]!
—
[4,5,6]
[4,5,7]
—
[5,6,7]
[5,7,4]
—
[6,7,4]
—
[7,4,5]
[7,4,8]!

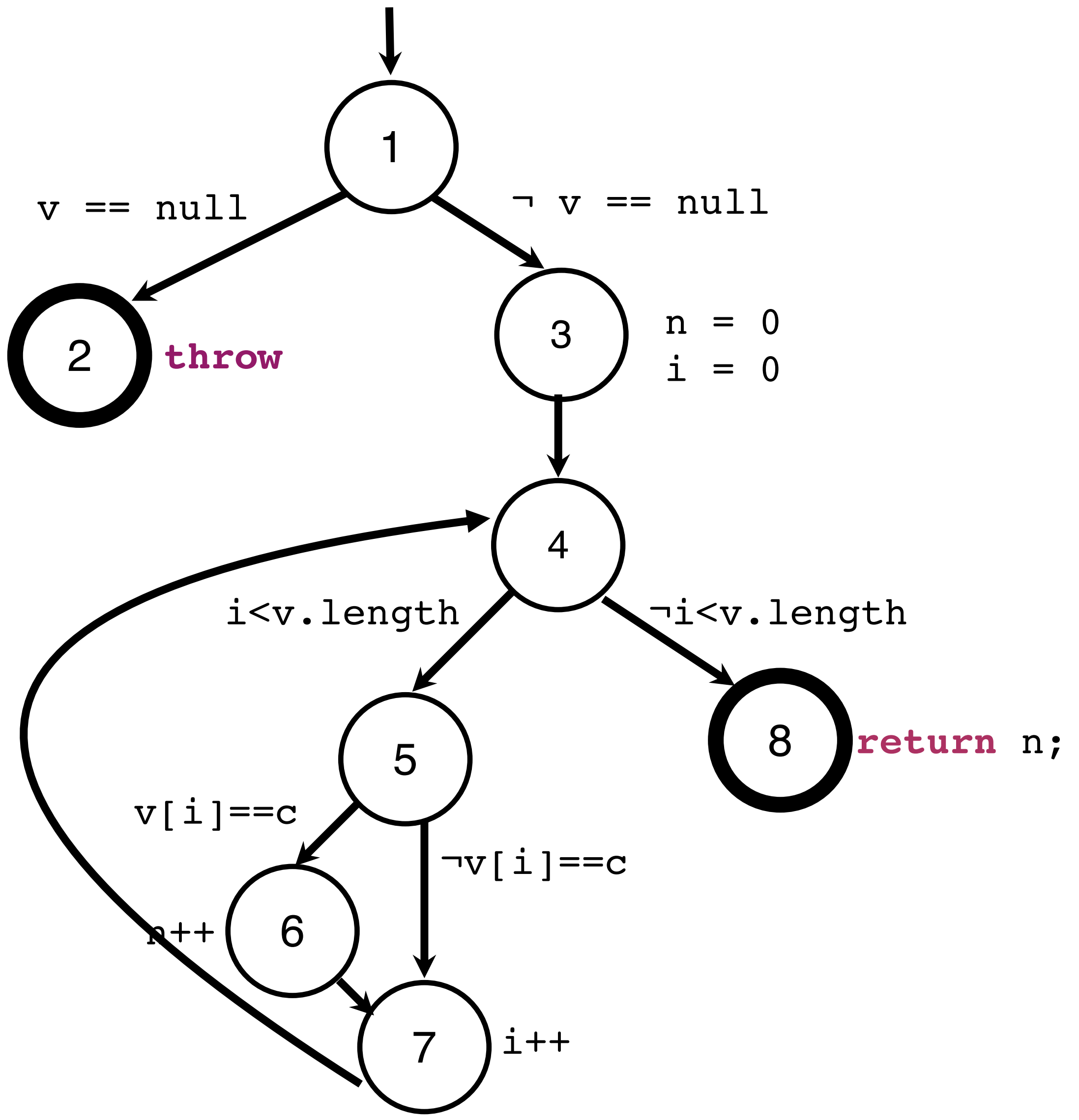
[1,3,4,5]
[1,3,4,8]!
—
[3,4,5,6]
[3,4,5,7]!
—
[4,5,6,7]
[4,5,7,4]*
—
[5,6,7,4]
[5,7,4,5]*
[5,7,4,8]!
—
[6,7,4,5]
[6,7,4,8]!
—
[7,4,5,6]
[7,4,5,7]*

[1,3,4,5,6]
[1,3,4,5,7]!
—
[3,4,5,6,7]!
—
[4,5,6,7,4]*
—
[5,6,7,4,5]*
[5,6,7,4,8]!
—
[6,7,4,5,6]*
—
[7,4,5,6,7]*

[1,3,4,5,6,7]!

! - cannot be extended
* - is a cycle

- Deriving prime paths:
  - Enumerate all simple paths of length 0,1, 2, 3, ... until no more simple paths can be found.
  - Pick the prime paths among all simple paths.



PPC satisfied by {t1,t2,t3,t4,t5}

Prime path	Covered by
[1,2]	t1
[1,3,4,8]	t2
[1,3,4,5,6,7]	t4 t5
[1,3,4,5,7]	t3
[4,5,7,4]	t3 t4
[4,5,6,7,4]	t3 t4 t5
[5,7,4,5]	t3
[5,7,4,8]	t4
[5,6,7,4,5]	t4 t5
[5,6,7,4,8]	t3 t5
[6,7,4,5,6]	t5
[7,4,5,7]	t4
[7,4,5,6,7]	t3 t5

t	test case values (v,c)	exp. value	test path
t1	`(null, 'a')`	IAE	[1,2]
t2	`({}, 'a')`	0	[1,3,4,8]
t3	`({'x','a'}, 'a')`	1	[1,3,4,5,7,4,5,6,7,4,8]
t4	`({'a','x'}, 'a')`	1	[1,3,4,5,6,7,4,5,7,4,8]
t5	`({'a','a'}, 'a')`	2	[1,3,4,5,6,7,4,5,6,7,4,8]

Exercise 2

```
public static boolean isPalindrome(String s) {
  if (s == null)
    throw new IllegalArgumentException();
  int left = 0;
  int right = s.length() - 1;
  boolean result = true;
  while (left < right && result) {
    if (s.charAt(left) != s.charAt(right))
      result = false;
    left++;
    right--;
  }
  return result;
}
```

1. Draw the CFG of `isPalindrome`.

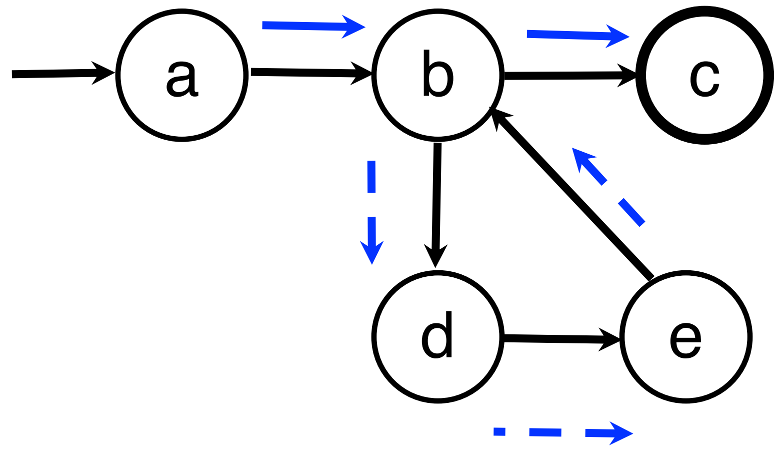
2. Identify TR(NC), TR(EC), TR(EPC).

3. If possible define test sets that satisfy: **a)** NC but not EC; **b)** EC but not EPC; **c)** EPC.

4. Identify TR(PPC). Are there infeasible requirements? Define test cases for all the feasible requirements.

5. Write a JUnit test class for the test cases in 4.

Best-effort touring using side-trips



[a,b,d,e,**b,c]**

- Assume that **p=[a,b,d,e,b,c]** is a feasible test path but **q=[a,b,c]** is not.
- We say ***p tours q with a side-trip*** since every edge in **p** also appears in **q** in the same order (the sidetrip is **[b,d,e,b]**).
- **Best-effort touring** - A test set T achieves best effort touring of TR if for every path in TR there is a test in T that tours the path either directly or using side-trips.
- Best-effort touring is a relevant way to deal with infeasible test requirements [though not helpful to deal with the `isPalindrome` example].

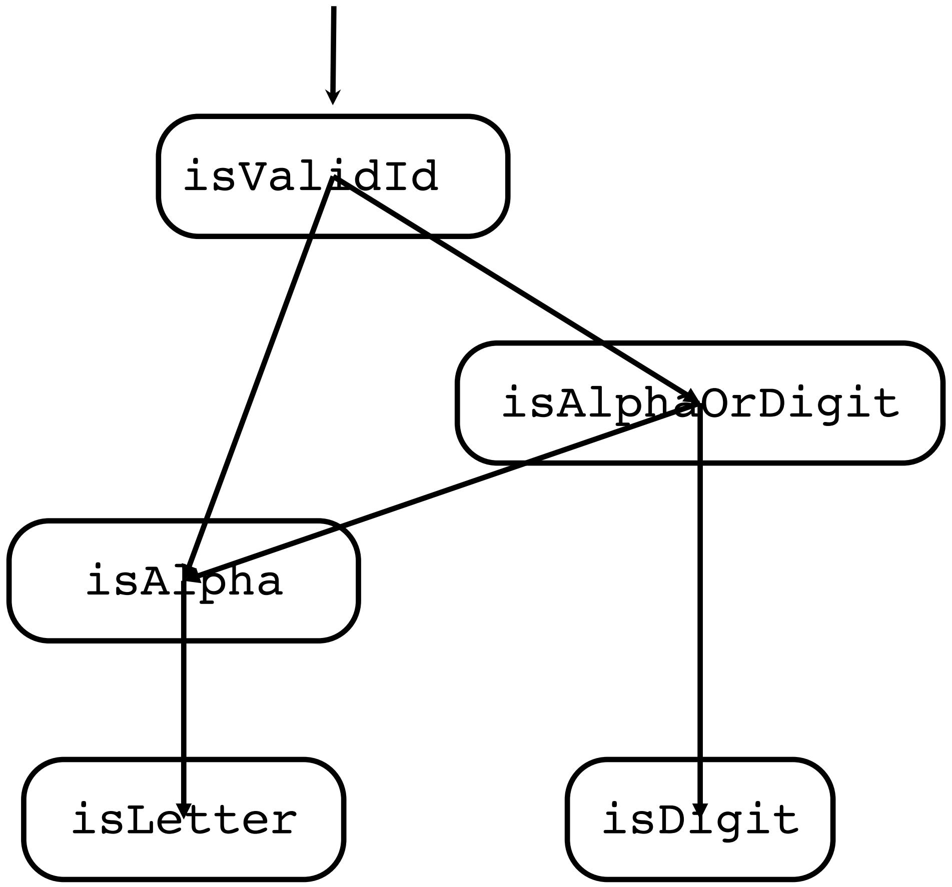
Exercise 3

```
public static int average(int[] v) {
  if (v == null || v.length == 0)
    throw new IllegalArgumentException();
  int r = 0;
  for (int i=0; i < v.length; i++)
    r = r + v[i];
  r = r / v.length;
  return r;
}
```

1. Draw the CFG of `average`.
2. Identify TR(PPC).
3. Are there infeasible requirements? Identify test cases that satisfy PPC using best-effort touring.
4. Write a JUnit test class for the test cases in 3.

Call graphs

```
public class CallGraphExample {
  public static boolean isValidId(String s) {
    if (s == null || s.length() == 0)
      return false;
    if (! isAlpha(s.charAt(0)))
      return false;
    for (int i=1; i < s.length(); i++) {
      if (! isAlphaOrDigit(s.charAt(i)))
        return false;
    }
    return true;
  }
  private static boolean isAlphaOrDigit(char c) {
    return Character.isDigit(c) || isAlpha(c);
  }
  private static boolean isAlpha(char c) {
    return c == '_'  || Character.isLetter(c);
  }
}
```



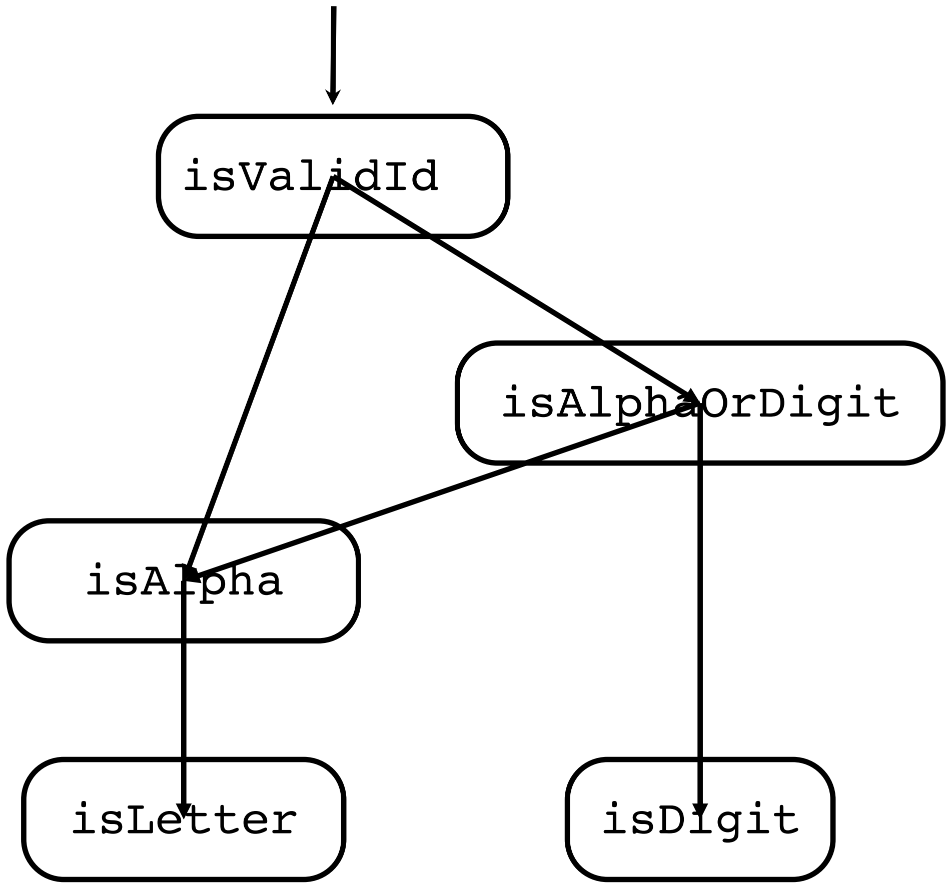
- A call graph expresses calls between units in the same module or different modules.
- The graph right expresses calls between units (Java methods) in module (class) `CallGraphExample` and also methods from class `Character`. We could have also included the call to methods in class `String (length(), charAt())`.

Graph coverage for design elements – Call graphs

- The call graph of a set of units (methods in Java) m_1, m_2, … is defined as follows:
  - A node is defined per each method m_i
  - An edge $m_I \rightarrow m_J$ is defined if there is a call from m_I from m_J
- The usual structural coverage criteria can be applied to call graphs. In particular:
  - Node coverage means that every method should be executed at least once, and is known as **method coverage** in the context of call graphs.
  - Edge coverage means that every call should be executed at least once, and is known as **call coverage** in the context of call graphs.

Exercise 4

```
public class CallGraphExample {
  public static boolean isValidId(String s) {
    if (s == null || s.length() == 0)
      return false;
    if (! isAlpha(s.charAt(0)))
      return false;
    for (int i=1; i < s.length(); i++)
      if (! isAlphaOrDigit(s.charAt(i)))
        return false;
    return true;
  }
  private static boolean isAlphaOrDigit(char c) {
    return Character.isDigit(c) || isAlpha(c);
  }
  private static boolean isAlpha(char c) {
    return c == '_'  || Character.isLetter(c);
  }
}
```



1. What calls are covered by considering test case `("x_1",true)` ?
2. Test case `("y1",true)` satisfies method coverage but not call coverage. Why?

Exercise 5

```
public static int f1(int n) {
  if (n % 4 == 0)
    return f2(n / 3, n);
  else
    return f3(n);
}
private static int f2(int a, int b) {
  if (a == 0)
    return f3(b);
  else
    return f4(a);
}
private static int f3(int n) {
  if (n == 0)
    return 1;
  else
    return n * f3(n - 1);
}
private static int f4(int n) {
  return n * n + 1;
}
```

1. Draw the call graph for the methods shown.
2. Identify a test set that satisfies method coverage but not call coverage.
3. Identify a test set that satisfies call coverage.
4. Write a JUnit test class for the test cases.

Some problems

1. If units or modules are interrelated through calls, how do we test them (units or modules) in isolation?
   - We'll cover some techniques to handle this requirements (e.g., *mock objects*).
2. On the other extreme, units in a module may not call each other. Call graphs and coupling data flows are not useful. We need to derive tests based on *call sequences*.
   - Some examples are discussed next week.
3. Polymorphism in OO programming
   - The target method of calls depends on the runtime type of objects.
   - ***OO all-object-call criterion***: cover each call site for each different possible type.

Testing object states

1. Besides methods, it is interesting to be able to test an object overall behavior
    - Behaviour is interactions (message passing) between objects
    - State machines can model such behaviour
    - If our test model is a state machine, we can generate test cases for the modeled behaviour
- A state machine is a mathematical model whose output is determined by both current and past input
    - Previous inputs are represented by a given state
    - Current input is an event which may produce a change of state (a transition)
    - State-based behaviour: same inputs are not always accepted. And when accepted, they may produce different outputs (an action)

State machine dynamics

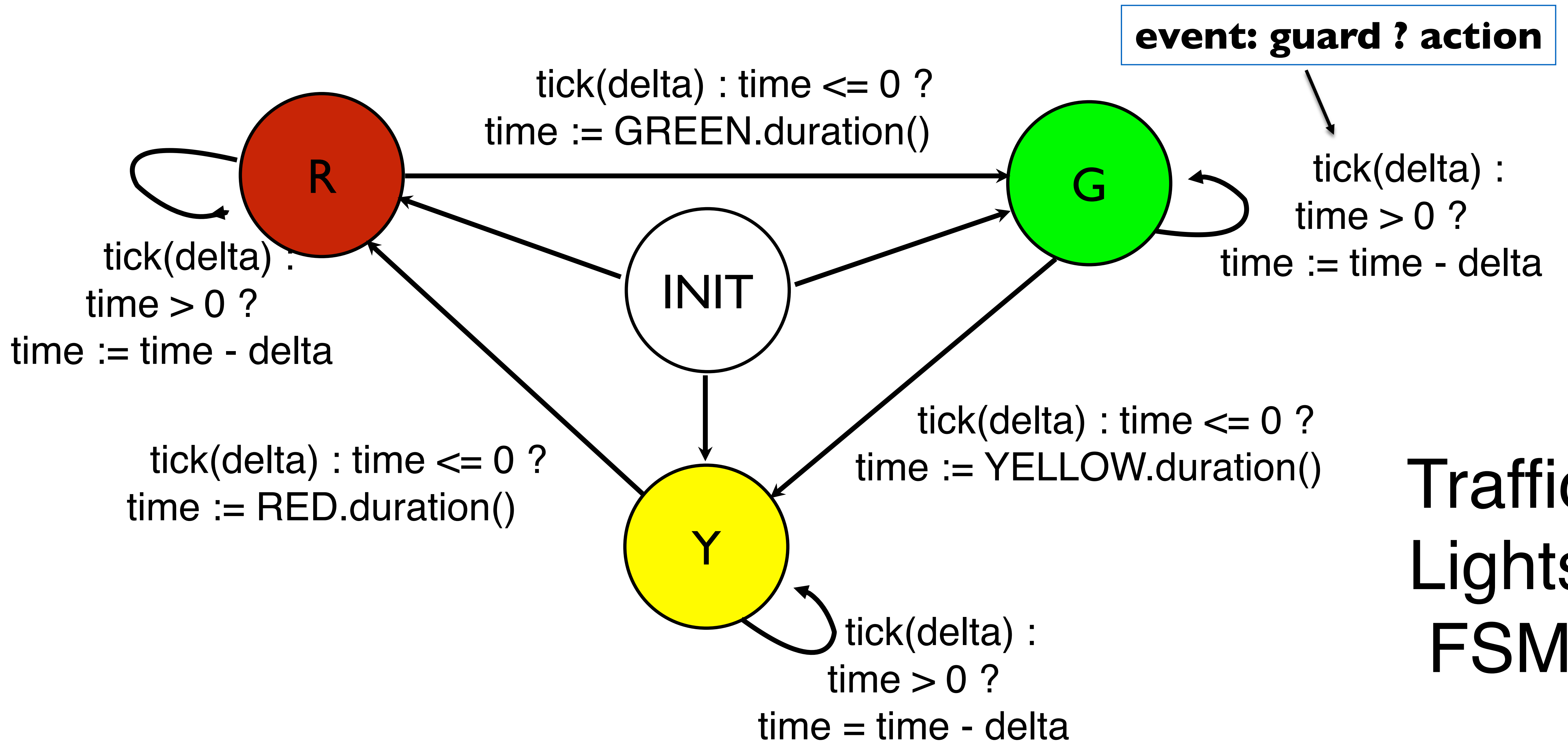
1. Begin in the **initial state**
2. Wait for an event
3. An event comes in
   - If not accepted by the current state, ignore
   - If accepted, check which transition is to occur (transitions might be associated with guards),
   - a transition occurs, output is produced (if any), the **resultant state** of the transition becomes the current state
4. Repeat from step 2 unless the current state is the **final state**

State machine properties

- How events are generated is not part of the model
- Transitions occur one at a time
- The machine can be in only one state at a time
- The current state cannot change except by a defined transition
- States, events, transitions, actions cannot be added during execution
- No concurrency

How to model the SUT's behaviour using a state machine?

Finite state machine (FSM) abstractions for SUT



Traffic Lights FSM

- **Nodes** represent **states of the SUT**
- **Edges** represent **transitions between states**, with associated pre-state (guards) and post-state (actions);

FSM - tabular representation

#	source state	transition event	target state
1	INIT	init(RED)	(RED, RED.duration())
2	INIT	init(YELLOW)	(GREEN, GREEN.duration())
3	INIT	init(GREEN)	(YELLOW, YELLOW.duration())
4	(RED, time)	tick(delta) : time > 0	(RED, time - delta)
5	(RED, time)	tick(delta) : time <= 0	(GREEN, GREEN.duration())
6	(GREEN, time)	tick(delta) : time > 0	(GREEN, time - delta)
7	(GREEN, time)	tick(delta) : time <= 0	(YELLOW, YELLOW.duration())
8	(YELLOW, time)	tick(delta) : time > 0	(YELLOW, time - delta)
9	(YELLOW, time)	tick(delta) : time <= 0	(RED, RED.duration())

- In the TrafficLights FSM, apart from INIT, we can represent states as tuples of the form (color, time).
- We can represent the FSM in tabular form as shown above.

State model validation

A state model must be complete, consistent, and correct before it is used to generate test cases

- There is an initial state with only outbound transitions
- There is a final state with only inbound transitions (if not, explicit reason is needed)
- No equivalent states
- Every state is reachable from the initial state
- The final state is reachable from all states
- Every defined event and every defined action appears in at least one transition

State model validation

A state model must be complete, consistent, and correct before it is used to generate test cases

- Except for the initial and final states, every state has at least one incoming and one outgoing transition
- The events accepted in a particular state are unique or differentiated by mutually exclusive guards
- The evaluation of a guard does not cause side effects

- Complete specification: For every state, every event is accepted or rejected (either explicitly or implicitly)

FSM coverage criteria

- FSM coverage criteria
  - **State Coverage** ~ Node Coverage
  - **Transition Coverage** ~ Edge Coverage
  - **Transition-Pair Coverage** ~ Edge-Pair coverage
  - …
- We can test TrafficLights using its FSM abstraction.
- A test path will correspond to a **sequence of calls** (constructor followed by tick(delta)):
  - TrafficLight(…); tick(delta)

TrafficLight class

```java
public class TrafficLight {
  private Light light;
  private int time;
  public TrafficLight(Light initialLight) {
    light = initialLight;
    time = initialLight.duration();
  }
  public Light getActiveLight() {
    return light;
  }
  public int getTimeLeft() {
    return time;
  }
  public void tick(int delta) {
    time = time - delta;
    if (time <= 0) {
      switch (light) {
        case RED:    light = GREEN; break;
        case GREEN:  light = YELLOW;
        case YELLOW: light = RED; break;
      }
      time = light.duration();
    }
  }
}
```

```java
public enum Light {
  RED(30),
  YELLOW(5),
  GREEN(30);

  public int duration() {
    return duration;
  }

  private int duration;

  private Light(int duration){
    this.duration = duration;
  }
}
```

Assume durations are positive

Exercise 6

1. Considering the TrafficLight FSM abstraction, identify a test set that satisfies state coverage. Represent each test case as a sequence of calls to the TrafficLight class.
2. Now identify a test set that satisfies transition coverage.
3. There is a fault in the `TrafficLight` Java class that may lead to a failure. Where? Indicate a test case that results in failure due to that fault.
4. Write a JUnit test class for the test cases.

Exercise 7

```
public class FixedCapacityStack {
  public FixedCapacityStack(int capacity) { ... }
  public int capacity() { .... }
  public int size() { .... }
  public boolean isEmpty() { ... }
  public boolean isFull() { ... }
  public void push (Object o) throws IllegalStateException { ... }
  public Object pop ()        throws IllegalStateException { ... }
  public Object peek ()       throws IllegalStateException { ... }
}
```

Consider the above skeleton for a fixed capacity stack. The capacity of a stack is specified by the constructor argument. Assume that that `IllegalStateException` is thrown by:

- `pop()` and `peek()` when queue is is empty
- `push()` when queue is full

1. Define a FSM abstraction such that :
    - nodes corresponds to the use of the stack's capacity (empty, full, not empty and not full)
    - edges correspond to calls; add pre-conditions and denote by EXC exceptions output events
2. Characterise test cases in the form of method sequence calls for transition coverage, using one test case per transition. Think of the required state assertions…