Basic Techniques

Software Reliability and Testing - Barbara Russo
SwSE - Software and Systems Engineering Research Group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Models of Analysis and Testing

- Every engineering and mathematical science use models
- They help to formulate a problem and devise solutions:
  - Models must be built the earliest possible
  - They must follow the development evolution

Models of Analysis and Testing

- Models are simpler than the reality as they represent reality with a limited number of factors
- They help factorise the reality and perform program analysis and testing simply and fast

Program analysis and testing

- Program analysis and testing **aim at checking software dependability properties**

Program analysis

- Analysis techniques that do **not need execute code** are the most used
  - Manual inspection of software artefacts and automated analyses; they can be used at any stage of development

Program analysis

- Manual inspection is expensive:
  - Re-inspecting a changed artefact can be too expensive
- Automatic analysis is limited to finite cases
  - Limited to models or a finite number of factors

Program analysis

- Nowadays research focus on dynamic analysis
  - First create scenarios of use
  - Then model program behaviour on those scenarios
  - Pb. It is not possible to re-create all scenarios of use

Testing

- Testing is a **dynamic activity**
- It can be done only when the artefacts to be tested are “executable”
- There are various testing activities depending on the type of artefacts to test
  - Acceptance test
  - System test etc..

Testing as a development technique

- Anticipating testing the earliest possible is one of the practices of agile methods:
  - Test First in XP
- Testing has been also used to develop new code:
  - Test Driven Development

Test Driven Development (TDD)

- Practice for writing unit tests and production code concurrently and at a very fine level of granularity
- Programmers
  - first write a small portion of a unit test, and
  - then they write just enough production code to make that unit test compile and execute

Test Driven Development (TDD)

- This cycle lasts somewhere between 30 seconds and five minutes. Rarely does it grow to ten minutes.
- In each cycle, the tests come first.
- Once a unit test is done, the developer goes on to the next test until they run out of tests for the task they are currently working on

Example

- TextFormatter: A text formatter that takes arbitrary strings and horizontally centers them in a page
- What are the methods:
  - setLineWidth()
  - centerLine()
- What are the parameters?

First we write the test	Then we write the production code

public void testCenterLine(){
Formatter f = new Formatter();
}
does not compile
	class Formatter{ }
	
	compiles and passes

public void testCenterLine(){
Formatter f = new Formatter();
f.setLineWidth(10);
assertEquals(" word ", f.center("word"));
 	
}
 	
does not compile
	class Formatter{
	public void setLineWidth(int width) { }
	public String center(String line) {
	return "";
	}
	}
	
	compiles and fails

	import java.util.Arrays;
	public class Formatter {
	private int width;
	private char spaces[];
	public void setLineWidth(int width) {
	this.width = width;
	spaces = new char[width];
	Arrays.fill(spaces, ' ');
	}
	public String center(String term) {
	StringBuffer b = new StringBuffer();
	int padding = width/2 - term.length();
	b.append(spaces, 0, padding);
	b.append(term);
	b.append(spaces, 0, padding);
	return b.toString();
	}
	}
	
	compiles and unexpectedly fails

	public String center(String term) {
	StringBuffer b = new StringBuffer();
	int padding = (width - term.length()) / 2;
	b.append(spaces, 0, padding);
	b.append(term);
	b.append(spaces, 0, padding);
	return b.toString();
	}
	compiles and passes

public void testCenterLine() {
Formatter f = new Formatter();
f.setLineWidth(10);
assertEquals(" word ", f.center("word"));
}
public void testOddCenterLine() {
Formatter f = new Formatter();
f.setLineWidth(10);
assertEquals(" hello ", f.center("hello"));
}
compiles and fails
	public String center(String term) {
	int remainder = 0;
	StringBuffer b = new StringBuffer();
	int padding = (width - term.length()) / 2;
	remainder = term.length() % 2;
	b.append(spaces, 0, padding);
	b.append(term);
	b.append(spaces, 0, padding + **remainder**);
	return b.toString(); }
	compiles and passes

Examples of techniques of analysis and testing

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Analysis and testing

- Typically they require artefacts to be available
  - We cannot wait until the artefacts are available
  - A thorough analysis or testing is not feasible
- We need to build abstract model of the program or its behaviour
  - They allow starting the analysis earlier
  - They can evolve with development

A model of program execution

- A model of program execution is a representation of a software execution simpler but that preserves some key attributes of it

State Space

- It is a **representation** of the program execution with a sequence of **states and transitions**
- The state space is a set of possible states and transitions
- For almost all programs, the state space is potentially infinite

Abstraction function

- The states are represented in the space by an abstraction function
- The state space is potentially infinite: the abstraction function might suppress some states to create the finite model

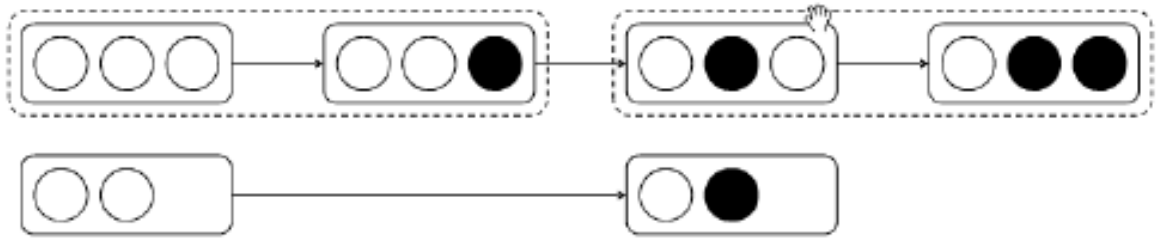
Abstraction function

- **Coarsening**: execution sequences are collapsed into shorter sequences
- **Non determinism**: states have been merged

Effects of abstraction

Assume the third state is neglected

1. Coarsening of execution model



2. Introduction of nondeterminism

Pezze & Young

CFG: an example of modelling technique to design test cases

Software Reliability and Testing - Barbara Russo
SwSE - Software and Systems Engineering Research Group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Control Flow Graphs

- In terms of program execution, CFG states represent where the code is executed
- **Directed graphs**
  - Node= portion of code
  - Directed Edge= flow of execution between two portions

Control Flow Graphs

- In terms of execution, a CFG keeps information of next instruction to be executed and **ignores values of variables**
- Example of **non deterministic abstraction**

```
1 boolean z = FALSE;
2 if(z && y<=2){
3   if(z){y++;}else{y--;}
4 }
```

the CFG of such code also models the non-feasible path

Control flow structure

- The control flow structure is modelled with direct graphs
- A direct graph is a set of arcs and nodes with one defined direction
  - A statement corresponds to a node, a flow of control from a statement to another to an arch
  - There is a start node and an end node
  - Each other node resides on a path between these two
  - Each node has a in-degree and an out degree
  - The start node has zero in - degree

Control flow structure

- A program is transformed in a direct graph called **control flow graph** that depicts the execution control of a program and the instruction to be executed
- It is a static representation of the program
- It makes visible the control structure
- Out-degree = 1 defines **procedural nodes** all the other nodes are called **predicate nodes**

CFG to design test cases

- It neglects data and data structures
- It focuses on data flow
- We can use this information to design test cases
- Let's see how to do it …
- First let's introduce the McCabe complexity measure which will help us to limit the number of test cases

McCabe CC

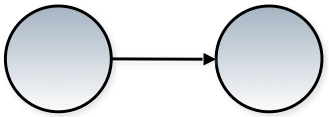
- McCabe Cyclomatic complexity
- Mapping codes to flow graphs
- Mapping flow graphs to numbers

CC definition

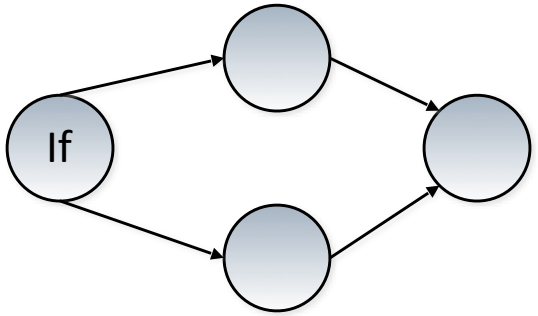
- CC= # of connected regions
- CC=# branches+1
- CC=# elements in a base
- CC=# decision point +1
- CC=#arcs-#nodes+2 (Euler characteristic)

Examples

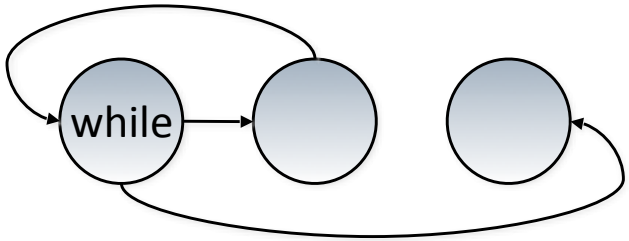
- Sequence



- If … then … else



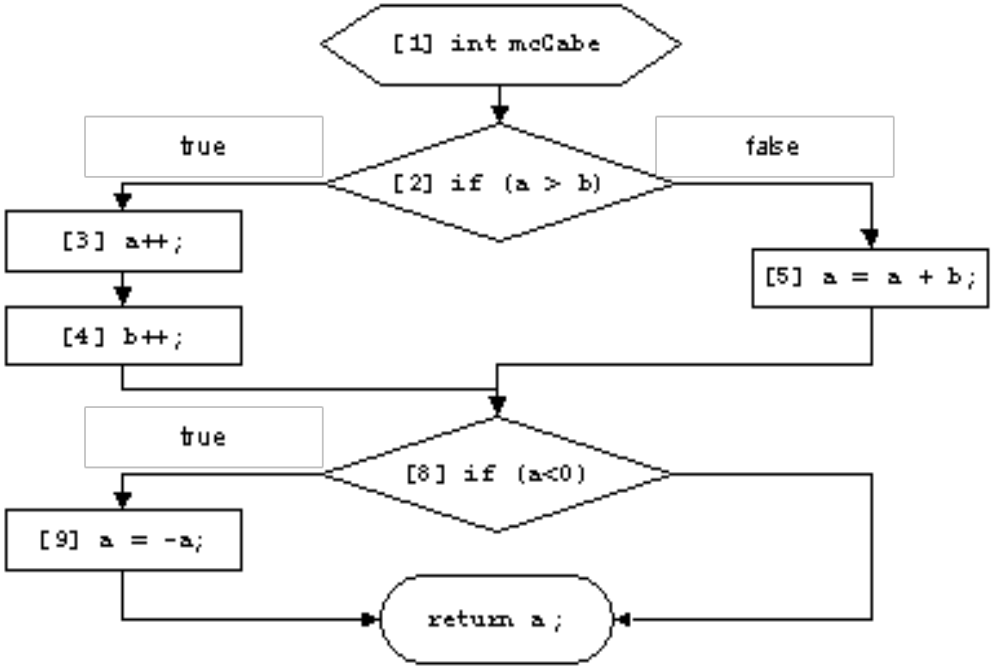
- While

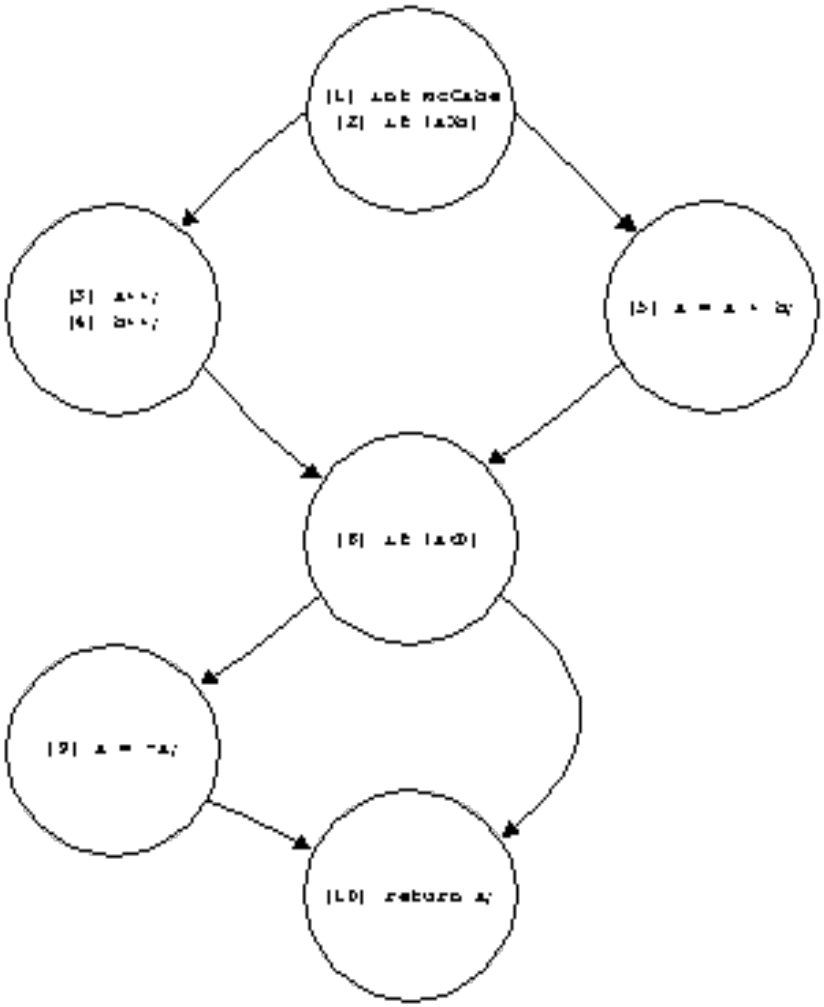


Example

```
 [1]  int mcCabe(int a, int b) {
 [2]    if (a >b) {
 [3]      a++;
 [4]      b--;
 [5]    } else {
 [6]      a=a + b;
 [7]    }
 [8]    if (a < 0) a=-a;
 [9]    return a;
[10]  }
```

Exercise: flow chart and flow graph

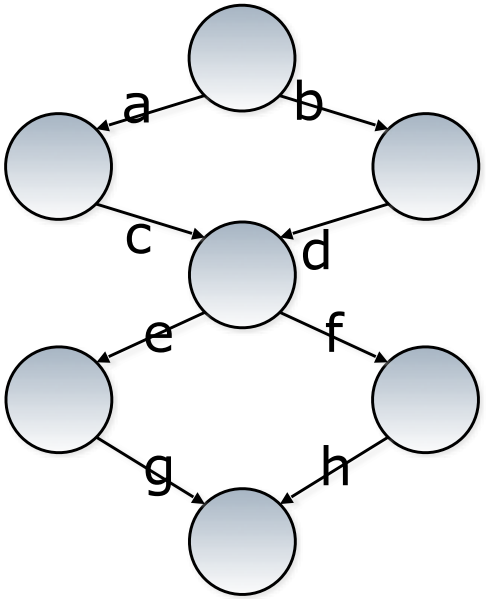




Some comments

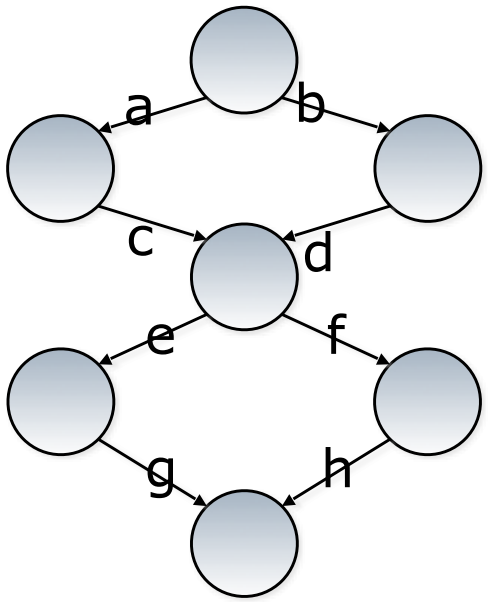
- CC traces only the **control flow**; no reference to the structure of the data (parameters, variable, …)
- We need to **detect all the paths** to cover the whole program: notion of complete path and linear independence
- A **complete path** is a path starting from the starting node and ending to the end node
- One complete path is **linearly independent** from the others if it does not exist a combination of the other complete paths to which is equal
- How to combine paths ...

Describe the base of the following graph



Rule to combine paths:
The arcs go from top to bottom
-a : is the arc in the opposite direction
-aceg: is the opposite complete path of aceg
ab: is first a and then b

Describe the base of the following graph



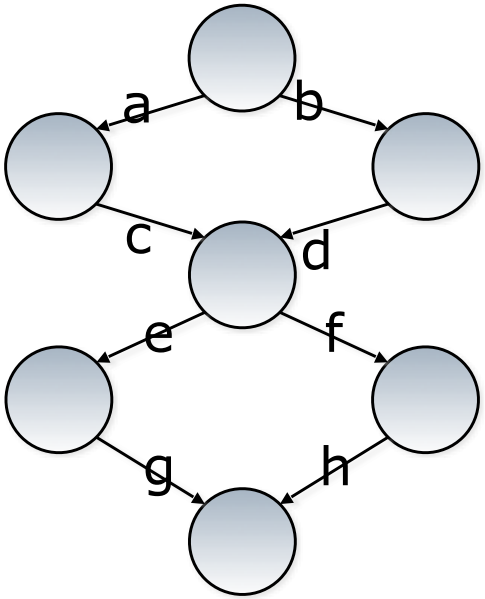
Rule to combine paths:
The arcs go from top to bottom
-a : is the arc in the opposite direction
-aceg: is the opposite complete path of aceg
ab: is first a and then b

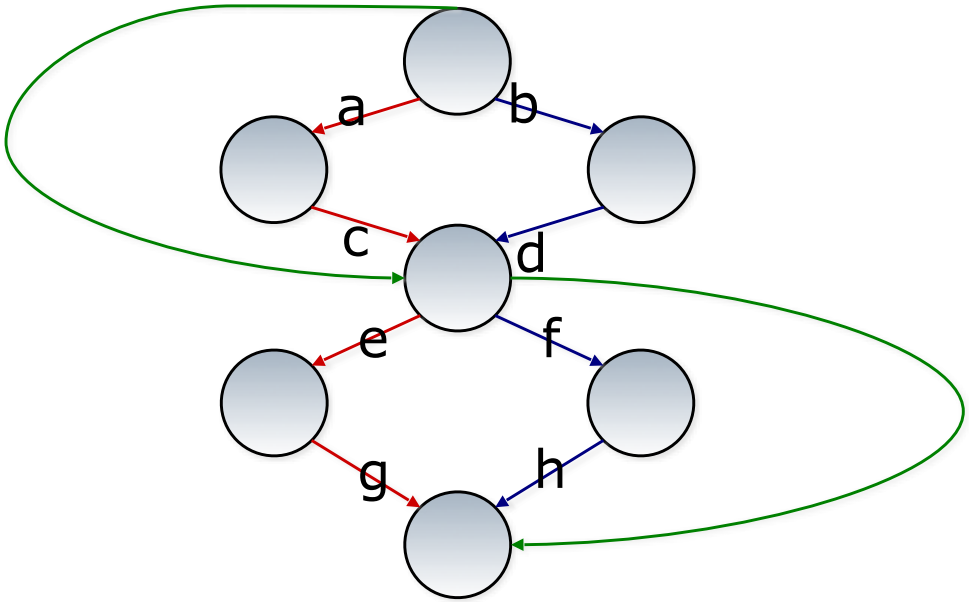
Complete paths:
aceg
bdfh
bdeg
acfh

The path acfh=aceg-bdeg+bdfh

aceg=acfh-fh+eg

Describe the base of the following graph



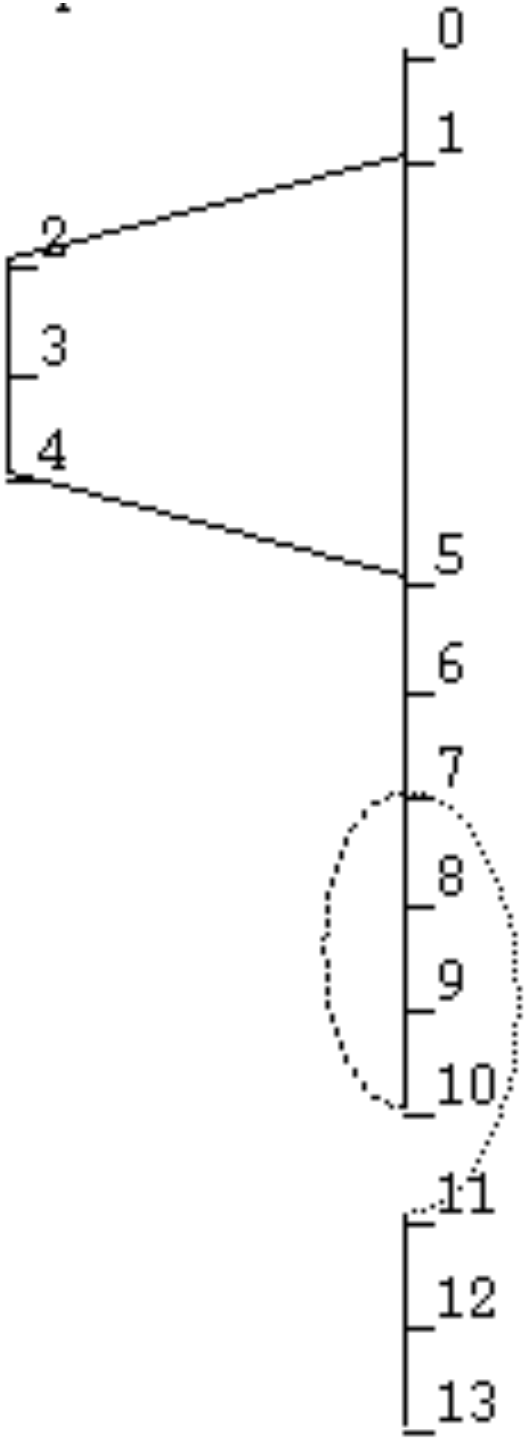


The path acfh=aceg-bdeg+bdfh

Draw the flow graph and Compute the CC

```
2        A0                euclid(int m, int n)
3                          {/* Assuming m and n both greater than 0,
4                            * return their greatest common divisor.
5                            * Enforce m >= n for efficiency.
6                            */
7                              int r;
8        A1                    if (n > m) {
9        A2                        r = m;
10       A3                        m = n;
11       A4                        n = r;
12       A5                    }
13       A6                    r = m % n;          /* m modulo n */
14       A7                    while (r != 0) {
15       A8                        m = n;
16       A9                        n = r;
17       A10                       r = m % n;     /* m modulo n */
18       A11                   }
19       A12                   return n;
20       A13               }
```

Result



Test coverage strategy

- Design test cases so that every node lies on at least one complete path (**statements coverage**)
- Design test cases such that every possible arc is executed at least once (**path coverage**)
- Statement coverage is addressed by finding a set of complete paths such that every node lies on at least one path, but…

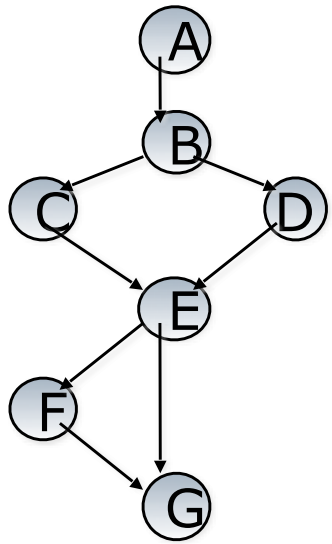
Issues with coverage

- Some path are infeasible
- Some edges are hidden
- Data values are not relevant

Some complete paths may be infeasible

- Infeasible path: a program path that cannot be executed for any input

```
A input(score)
B if score<45
C     then print ('fail')
D     else print ('pass')
E if score >70
F     then print ('with distinction')
G end
```



- The path A-B-C-E-F-G is infeasible and
- It will be never executed
- We create a test case for the non-feasible path: waisting time

Some path are implicit

```
if x < 0 then
    x := -x;
end if
z := x;
```

The else condition is implicit

```
else
    null;
```

A test case exercising only x<0 reaches the 100% statement coverage, but it does not prevent a bug to occur if x >= 0

Solution

- Use condition coverage to uncover the error

Code Example

```
counter:= 0;
found := false;
if number_of_items ≠ 0 then counter :=1;
   while (not found) and counter < number_of_items loop
        if table(counter) = desired_element then
                found := true;
        end if;
        counter := counter + 1;
   end loop;
end if;
if found then write ("the desired element exists in
           the table");
else write ("the desired element does not exists in
        the table");
end if;
```

≤

Typical mistake in search algorithms

- If the the search succeeds for the last item the above code will not find it
- If the search succeeds for the second last item all the edges are exercised with an appropriate test suite, but if an error is due to the last item the tests do not discover it
- T_0={number_of_items=0}
  T_1={number_of_items=3}

Template for path coverage

- Draw the flow graph
- Count the possible independent complete paths
- Create a table with all the possible arcs as column headers
- Decide the number of test cases on the boundary of the input. What are the critical values to use as test inputs?
- Check for path coverage (all the arcs covered by the tests)
  - how many paths do you need to consider? (excluding infeasible and dependent ones?)

Code coverage: an example of analysis technique to verify program correctness

Software Reliability and Testing - Barbara Russo
SwSE - Software and Systems Engineering Research Group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Exercise

- Consider the snippet code in the next slide
    A. Draw the control flow graph
    B. How many test cases would you build using the complete paths? Motivate your answer
    C. Build a test case suite for 100% statement coverage
    D. Is the test case suite in item c) enough for 100% branch coverage? Motivate your answer building the test case suite for 100% branch coverage

Dealing with loops

- Look for conditions that execute loops
  - Zero times
  - A maximum number of times
  - A average number of times (statistical criterion)
- For example, in the search algorithm
  - Skipping the loop (the table is empty)
  - Executing the loop once or twice and then finding the element
  - Searching the entire table without finding the desired element

The power function

```
public class PowerFunction {
    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        int y = Integer.parseInt(args[1]);
        int w = Math.abs(y);
        int z = 1;

        while(w!=0){
            z=z*x;
            w=w-1;
        }
        if(y<0){z=1/z;}
        System.out.println("result is "+z);}
}
```

Program computing Z=X^Y

Issues

All paths

Infeasible path

1 ->2 -> 4 -> 5-> 6

Infinite number of paths :

As many ways to iterate

2 -> (3 -> 2)* as values of Abs(Y)

All branches

Two test cases are enough

Y<0 : 1 -> 2 -> (3 ->2)+ -> 4 -> 5 -> 6

Y≥0 : 1 -> 2 -> (3 -> 2)* -> 4 -> 6

All statements

One test case is enough

Y<0 : 1 -> 2 -> (3 ->2)+ ->4 -> 5 -> 6