Path Testing and Test Coverage

Chapter 9
Structural Testing

- Also known as glass/white/open box testing

- Structural testing is based on using specific knowledge of the program source text to define test cases
  - Contrast with functional testing where the program text is not seen but only hypothesized
Structural Testing

- Structural testing methods are amenable to
  - Rigorous definitions
    - Control flow, data flow, coverage criteria
  - Mathematical analysis
    - Graphs, path analysis
  - Precise measurement
    - Metrics, coverage analysis
What is a program graph?
Given a program written in an imperative programming language

- Its **program graph** is a directed graph in which nodes are statements and statement fragments, and edges represent flow of control

- Two nodes are connected if execution can proceed from one to the other
output ("Enter 3 integers")
input (a, b, c)
output("Side a b c: ", a, b, c)
if (a < b) and (b < a+c) and (c < a+b)
then isTriangle ← true
else isTriangle ← false
fi
if isTriangle
then if (a = b) and (b = c)
else output ("equilateral")
else if (a ≠ b ) and ( a ≠ c ) and ( b ≠ c)
then output ("scalene")
else output("isosceles")
fi
fi
else output ("not a triangle")
fi
What is a DD-path?
A **decision-to-decision** path (DD-Path) is a path chain in a program graph such that

- Initial and terminal nodes are distinct
- Every interior node has indeg = 1 and outdeg = 1
  - The initial node is 2-connected to every other node in the path
  - No instances of 1- or 3-connected nodes occur
What is the definition of node connectedness?

Hint: There are 4-types of connectedness
Connectedness definition – 2

- Two nodes J and K in a directed graph are
  - 0-connected iff no path exists between them

  ![Diagram showing J and K with no path between them]

- 1-connected iff a semi-path but no path exists between them

  ![Diagram showing a semi-path from J to K]
  ![Diagram showing a semi-path from J to K]

  ![Diagram showing a semi-path from J to K]
Connectedness definition – 2

- Two nodes J and K in a directed graph are
  - 2-connected iff a path exists between them
  
    ![Diagram showing a path from J to K](image1)

  - 3-connected iff a path goes from J to K, and a path goes from K to \( n_1 \)

    ![Diagram showing two paths](image2)
DD-Path – formal definition

- A **decision-to-decision** path (DD-Path) is a chain in a program graph such that:
  - **Case 1**: consists of a single node with indeg=0
  - **Case 2**: consists of a single node with outdeg=0
  - **Case 3**: consists of a single node with indeg ≥ 2 or outdeg ≥ 2
  - **Case 4**: consists of a single node with indeg =1, and outdeg = 1
  - **Case 5**: it is a maximal chain of length ≥ 1

- DD-Paths are also known as **segments**
## Triangle program DD-paths

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Path</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First</td>
<td>1</td>
</tr>
<tr>
<td>2,3</td>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>G</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Path</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>H</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>I</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>J</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>L</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>M</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>Last</td>
<td>2</td>
</tr>
</tbody>
</table>
What is a DD-path graph?
Given a program written in an imperative language, its **DD-Path graph** is a directed graph, in which

- Nodes are DD-Paths of its program graph
- Edges represent control flow between successor DD-Paths.

Also known as **Control Flow Graph**
Control Flow Graph Derivation

- Straightforward process
- Some judgment is required
- The last statement in a segment must be
  - a predicate
  - a loop control
  - a break
  - a method exit
Triangle program DD-path graph
public int displayLastMsg(int nToPrint) {
    np = 0;
    if ((msgCounter > 0) && (nToPrint > 0)) {
        for (int j = lastMsg; (( j != 0) && (np < nToPrint)); --j) {
            System.out.println(messageBuffer[j]);
            ++np;
        }
        if (np < nToPrint) {
            for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {
                System.out.println(messageBuffer[j]);
                ++np;
            }
        }
    }
    return np;
}
public int displayLastMsg(int nToPrint) {
    np = 0;
    if (msgCounter > 0) && (nToPrint > 0)) {
        for (int j = lastMsg;
            (j != 0) && (np < nToPrint));
            --j)
        { System.out.println(messageBuffer[j]);
        ++np;
    }
}
### displayLastMsg– Segments part 2

<table>
<thead>
<tr>
<th>Line</th>
<th>Segment</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>G</td>
<td>if (np &lt; nToPrint)</td>
</tr>
<tr>
<td>9a</td>
<td>H</td>
<td>{ for (int j = SIZE;</td>
</tr>
<tr>
<td>9b</td>
<td>I</td>
<td>((j != 0) &amp;&amp;</td>
</tr>
<tr>
<td>9c</td>
<td>J</td>
<td>(np &lt; nToPrint));</td>
</tr>
<tr>
<td>9d</td>
<td>K</td>
<td>--j)</td>
</tr>
<tr>
<td>10</td>
<td>K</td>
<td>} System.out.println(messageBuffer[j]);</td>
</tr>
<tr>
<td>11</td>
<td>K</td>
<td>++np;</td>
</tr>
<tr>
<td>12</td>
<td>L</td>
<td>}</td>
</tr>
<tr>
<td>13</td>
<td>L</td>
<td>}</td>
</tr>
<tr>
<td>14</td>
<td>L</td>
<td>}</td>
</tr>
<tr>
<td>15</td>
<td>L</td>
<td>return np;</td>
</tr>
<tr>
<td>16</td>
<td>L</td>
<td>}</td>
</tr>
</tbody>
</table>
Control flow graphs definition – 1

- Depict which program segments may be followed by others
- A segment is a node in the CFG
- A conditional transfer of control is a **branch** represented by an edge
- An **entry node** (no inbound edges) represents the entry point to a method
- An **exit node** (no outbound edges) represents an exit point of a method
Control flow graphs definition – 2

- An **entry-exit path** is a path from the entry node to the exit node.

- **Path expressions** represent paths as sequences of nodes.

- Loops are represented as segments within parentheses followed by an asterisk.

- There are 22 different entry-exit path expressions in displayLastMsg.
## Entry-Exit paths

<table>
<thead>
<tr>
<th>Entry-exit path expression (part 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
**Entry-Exit paths**

<table>
<thead>
<tr>
<th></th>
<th>A B C D E G H I J L</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>A B C D E G H I J L</td>
</tr>
<tr>
<td>13</td>
<td>A B C D E G H (I J K)* I L</td>
</tr>
<tr>
<td>14</td>
<td>A B C D E G H (I J K)* I J L</td>
</tr>
<tr>
<td>15</td>
<td>A B C (D E F)* D G H I L</td>
</tr>
<tr>
<td>16</td>
<td>A B C (D E F)* D G H I J L</td>
</tr>
<tr>
<td>17</td>
<td>A B C (D E F)* D G H (I J K)* I L</td>
</tr>
<tr>
<td>18</td>
<td>A B C (D E F)* D G H (I J K)* I J L</td>
</tr>
<tr>
<td>19</td>
<td>A B C (D E F)* D E G H I L</td>
</tr>
<tr>
<td>20</td>
<td>A B C (D E F)* D E G H I J L</td>
</tr>
<tr>
<td>21</td>
<td>A B C (D E F)* D E G H (I J K)* I L</td>
</tr>
<tr>
<td>22</td>
<td>A B C (D E F)* D E G H (I J K)* I J L</td>
</tr>
</tbody>
</table>
### Path condition by Segment Name

<table>
<thead>
<tr>
<th>Entry/Exit Path</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 A L</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2 A B L</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>3 A B C D G L</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4 A B C D E G L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5 A B C (D E F)* D G L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/F</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6 A B C (D E F)* D E G L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7 A B C D G H I L</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8 A B C D G H I J L</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>9 A B C D G H (I J K)* I L</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T/F</td>
<td>T/F</td>
<td>T</td>
</tr>
<tr>
<td>10 A B C D G H (I J K)* I J L</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
</tr>
<tr>
<td>11 A B C D E G H I L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>

**x/x Conditions at loop-entry / loop-exit** – is don’t care
## Paths displayLastMsg – decision table – part 2

### Path condition by Segment Name

<table>
<thead>
<tr>
<th>Entry/Exit Path</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 A B C D E G H I J L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>13 A B C D E G H (I J K)* I L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
</tr>
<tr>
<td>14 A B C D E G H (I J K)* I J L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
</tr>
<tr>
<td>15 A B C (D E F)* D G H I L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>16 A B C (D E F)* D G H I J L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>17 A B C (D E F)* D G H (I J K)* I L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
</tr>
<tr>
<td>18 A B C (D E F)* D G H (I J K)* I J L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
</tr>
<tr>
<td>19 A B C (D E F)* D E G H I L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>20 A B C (D E F)* D E G H I J L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>21 A B C (D E F)* D E G H (I J K)* I L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>22 A B C (D E F)* D E G H (I J K)* I J L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**x/x Conditions at loop-entry / loop-exit** – is don’t care
List the program text coverage metrics.
Program text coverage Metrics – 2

- $C_0$ Every Statement
- $C_1$ Every DD-path
- $C_{1p}$ Every predicate to each outcome
- $C_2$ $C_1$ coverage + loop coverage
- $C_d$ $C_1$ coverage + every dependent pair of DD-paths
- $C_{MCC}$ Multiple condition coverage
- $C_{ik}$ Every program path that contains k loop repetitions
- $C_{stat}$ Statistically significant fraction of the paths
- $C_\infty$ Every executable path
Program text coverage models

- What are the common program text coverage models?
Program text coverage models – 2

- Statement Coverage
- Segment Coverage
- Branch Coverage
- Multiple-Condition Coverage
When is statement coverage achieved?
Statement coverage – $C_0 – 2$

- Achieved when all statements in a method have been executed at least once

- A test case that will follow the path expression below will achieve statement coverage in our example

  \[\text{A B C (D E F)* D G H (I J K)* I L}\]

- One test case is enough to achieve statement coverage!
When is segment coverage achieved?
Segment coverage – 2

- Achieved when all segments have been executed at least once
  - Segment coverage counts segments rather than statements
  - May produce drastically different numbers
    - Assume two segments P and Q
    - P has one statement, Q has nine
    - Exercising only one of the segments will give either 10% or 90% statement coverage
    - Segment coverage will be 50% in both cases
Statement coverage problems

- What problems are there with statement coverage?
Statement coverage problems – 2

- Important cases may be missed
  - Predicate may be tested for only one value
    - misses many bugs
  - Loop bodies may only be iterated only once

```java
String s = null;
if (x != y) s = "Hi";
String s2 = s.substring(1);
```

- **What coverage solves this problem?**
  - Define it
Branch coverage – $C_{1p}$

- Achieved when every edge from a node is executed at least once
- At least one true and one false evaluation for each predicate

- How many test cases are required?
Branch coverage – $C_{1p} – 2$

- Can be achieved with $D+1$ paths in a control flow graph with $D$ 2-way branching nodes and no loops
  - Even less if there are loops

- In the Java example `displayLastMsg` branch coverage is achieved with three paths – see next few slides

```
X L
X C (Y F)* Y G L
X C (Y F)* Y G H (Z K)* Z L
```
Java example program displayLastMsg – DD-path graph

X, Y & Z are shorthand for the nodes within the dotted boxes; used for branch testing
Java example program displastLastMsg – aggregate predicate DD-path graph
### Path condition by Segment Name

<table>
<thead>
<tr>
<th>Branch Coverage</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 XL</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2 XL</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>3 XCYGL</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4 XCYGL</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5 XC (YF)* YGL</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6 XC (YF)* YGL</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>F</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7 XCYGHZL</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>8 XCYGHZL</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>9 XCYGH (ZK)* IL</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
</tr>
<tr>
<td>10 XCYGH (ZK)* IL</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>–</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
</tr>
<tr>
<td>11 XCYGHZL</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
</tbody>
</table>

- x/x Conditions at loop-entry / loop-exit
- – is don’t care
### Aggregate Paths – decision table example – part 2

**Path condition by Segment Name**

<table>
<thead>
<tr>
<th>Branch Coverage</th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 X C Y G H Z L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>13 X C Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
</tr>
<tr>
<td>14 X C Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
</tr>
<tr>
<td>15 X C (Y F)* Y G H Z L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>16 X C (Y F)* Y G H Z L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>17 X C (Y F)* Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
</tr>
<tr>
<td>18 X C (Y F)* Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T/F</td>
<td>T/–</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
</tr>
<tr>
<td>19 X C (Y F)* Y G H Z L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>F</td>
<td>–</td>
</tr>
<tr>
<td>20 X C (Y F)* Y G H Z L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>21 X C (Y F)* Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>22 X C (Y F)* Y G H (Z K)* Z L</td>
<td>T</td>
<td>T</td>
<td>T/T</td>
<td>T/F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

**x/x Conditions at loop-entry / loop-exit**  
– is don’t care
What are the problems with branch coverage?
Branch coverage problems – 2

- Ignores implicit paths from compound paths
  - 11 paths in aggregate model vs 22 in full model
Branch coverage problems – 3

- Ignores implicit paths from compound paths
  - 11 paths in aggregate model vs 22 in full model

- Short-circuit evaluation means that many predicates might not be evaluated
  - A compound predicate is treated as a single statement. If $n$ clauses, $2^n$ combinations, but only 2 are tested
Branch coverage problems – 4

- Ignores implicit paths from compound paths
  - 11 paths in aggregate model vs 22 in full model
- Short-circuit evaluation means that many predicates might not be evaluated
  - A compound predicate is treated as a single statement. If \( n \) clauses, \( 2^n \) combinations, but only 2 are tested
- **Only a subset of all entry-exit paths is tested**
  - **Two tests for branch coverage vs 4 tests for path coverage**
    - \( a = b = x = y = 0 \) **and** \( a = x = 0 \land b = y = 1 \)

```java
if (a == b) x++;
if (x == y) x--;
```
Overcoming branch coverage problems

- How do we overcome branch coverage problems?
Overcoming branch coverage problems – 2

- **Use Multiple condition coverage**

  - All true-false combinations of simple conditions in compound predicates are considered at least once
    - Guarantees statement, branch and predicate coverage
    - Does not guarantee path coverage
  
  - A truth table may be necessary
  
  - Not necessarily achievable
    - lazy evaluation – true-true and true-false are impossible
    - mutually exclusive conditions – false-false branch is impossible

```c
if ((x > 0) || (x < 5)) ...
```
Overcoming branch coverage problems – 3

- Can have infeasible paths due to dependencies and redundant predicates
  - Paths perpetual .. motion and free .. lunch are impossible
  - In this case indicates a potential bug
    - At least poor program text

```plaintext
if x = 0 then oof.perpetual
    else oof.free
fi

if x != 0 then oof.motion
    else oof.lunch
fi
```
Dealing with Loops

- Loops are highly fault-prone, so they need to be tested carefully

- Based on the previous slides on testing decisions what would be a simple view of testing a loop?
Dealing with Loops – 2

- Simple view
  - Involves a decision to traverse the loop or not
  - Test as a two way branch

- What would functional testing suggest as a better way of testing?

- What tests does it suggest?
Dealing with Loops – 3

- A bit better
  - Boundary value analysis on the index variable
  - Suggests a zero, one, many tests

- How do we deal with nested loops?
Dealing with Loops – 3

- Nested loops
  - Tested separately starting with the innermost

- Once loops have been tested what can we do with the control flow graph?
Dealing with Loops – 4

- Once loops have been tested
  - They can be condensed to a single node
Condensation graphs

- Condensation graphs are based on removing strong components or DD-paths

- For programs remove structured program constructs
  - One entry, one exit constructs for sequences, choices and loops
  - Each structured component once tested can be replaced by a single node when condensing its graph
Program text that violates proper structure cannot be condensed

- Branches either into or out of the middle of a loop
- Branches either into or out of then and else phrases of if...then...else statements
- Increases the complexity of the program
- Increases the difficulty of testing the program
Cyclomatic number

- The cyclomatic number for a graph is given by
  - $\text{CN}(G) = e - v + 2c$
    - $e$ number of edges
    - $v$ number of vertices
    - $c$ number of connected regions

- For strongly connected graphs, need to add edges from every sink to every source
Cyclomatic number for programs

- For properly structured programs there is only one component with one entry and one exit. There is no edge from exit to entry.

- Definition 1: $CN(G) = e - v + 2$
  - Only 1 component, not strongly connected

- Definition 2: $CN(G) = p + 1$
  - $p$ is the number of predicate nodes with out degree = 2

- Definition 3: $CN(G) = r + 1$
  - $r$ is the number of enclosed regions