CMPT 473
Software Quality Assurance

Graph Coverage

Nick Sumner
Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite **covers / is adequate** for our quality objectives.
Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite *covers/* is *adequate* for our quality objectives.
- So far: Input & Requirements based
Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers / is adequate for our quality objectives.
- So far: Input & Requirements based

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```
Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite *covers/is adequate* for our quality objectives.
- So far: Input & Requirements based

```
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

*Efficiently sort a provided list in under X seconds*

How well can input based techniques test this?
Recall: Coverage/Adequacy

- Can't look at all possible inputs.
- Need to determine if a test suite covers/is adequate for our quality objectives.
- So far: Input & Requirements based

```
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

Efficiently sort a provided list in under X seconds

How well can input based techniques test this?

How might we do better?
White Box / Black Blox

• Considering only the requirements or input is a black box approach
  – Treats the program like an opaque box
  – No deep knowledge of the program's structure
White Box / Black Blox

- Considering only the requirements or input is a black box approach
  - Treats the program like an opaque box
  - No deep knowledge of the program's structure
- Techniques that use artifacts of the program structure are white box approaches
  - They can 'see into' the program's implementation
White Box Testing

- What is a simple approach that solves our problem here?

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```
White Box Testing

- What is a simple approach that solves our problem here?
- **Statement Coverage**
  - How many of the statements did the suite test?

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```
White Box Testing

- What is a simple approach that solves our problem here?
- **Statement Coverage**
  - How many of the statements did the suite test?
- **Branch Coverage**
  - How many of the condition outcomes were tested?

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}```
White Box Testing

• In this course, we'll mostly look at graph coverage based techniques
White Box Testing

- In this course, we'll mostly look at graph coverage based techniques
  - Most commonly used metrics in the real world
White Box Testing

• In this course, we'll mostly look at graph coverage based techniques
  – Most commonly used metrics in the real world
  – Most concepts can be modeled through graphs  
    e.g. programs, protocols, use patterns, designs, ...

So a bit of review...
Graphs

• What is a graph $G$?
Graphs

• What is a graph $G$?
  – A set $N$ of nodes
Graphs

• What is a graph $G$?
  – A set $N$ of nodes
  – A set $E$ of edges
Graphs

- What is a graph $G$?
  - A set $N$ of nodes
  - A set $E$ of edges
- When edges are directed from one node to another, the graph is a *directed graph*
Graphs

- What is a graph $G$?
  - A set $N$ of nodes
  - A set $E$ of edges
- When edges are directed from one node to another, the graph is a directed graph.
- A path is a list of pairwise connected nodes.
Graphs

- What is a graph $G$?
  - A set $N$ of nodes
  - A set $E$ of edges
- When edges are directed from one node to another, the graph is a directed graph
- A path is a list of pairwise connected nodes
Graphs

• What is a graph \( G \)?
  – A set \( N \) of nodes
  – A set \( E \) of edges
• When edges are directed from one node to another, the graph is a directed graph
• A path is a list of pairwise connected nodes
Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!
Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!
- **Control Flow Graphs**
  - **Nodes** comprise the code of a program
Control Flow Graphs (CFGs)

- Programs can be modeled as graphs
  - Used extensively in compilers
  - Also used in testing!

- **Control Flow Graphs**
  - **Nodes** comprise the code of a program
  - **Edges** show the paths that an execution may take through a program
Control Flow Graphs

Example:

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```
Control Flow Graphs

Example:

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

```java
if (list.size() < THRESHOLD) {
    sort1(list);
    sort2(list);
}
return;
```
Control Flow Graphs

Example:

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

```java
if (list.size() < THRESHOLD)
    sort1(list);
else
    sort2(list);
return;
```
Control Flow Graphs

Example:

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

Control Flow Graph:

- `if (list.size() < THRESHOLD)`
  - `size < THRESHOLD`:
    - `sort1(list)`
  - `size >= THRESHOLD`:
    - `sort2(list)`

`return;`
Control Flow Graphs

Example:

```java
void sortEfficiently(List list) {
    if (list.size() < THRESHOLD) {
        sort1(list);
    } else {
        sort2(list);
    }
}
```

![Control Flow Graph Diagram](image-url)
Control Flow Graphs

• Many types of nodes:

```java
list = ...

if (list.size() < THRESHOLD)
    sort1(list);
    sort2(list);
    print(list);
else
    return;

size < THRESHOLD
size >= THRESHOLD
```
Control Flow Graphs

- Many types of nodes:

```java
list = ...

if (list.size() < THRESHOLD)
    sort1(list);
    sort2(list);
    print(list);

return;
```

Entry Node

0 incoming edges

size < THRESHOLD

size >= THRESHOLD

sort1(list);

sort2(list);

print(list);

return;
Control Flow Graphs

• Many types of nodes:

Entry Node

list = ...

if (list.size() < THRESHOLD)

size < THRESHOLD

sort1(list);

size >= THRESHOLD

sort2(list);

print(list);

return;

Exit Node

0 incoming edges

0 outgoing edges
Control Flow Graphs

- Many types of nodes:

  - Entry Node: 0 incoming edges
  - Decision/Branch Node: >1 outgoing edges
  - Exit Node: 0 outgoing edges

```java
list = ...

if (list.size() < THRESHOLD)
  sort1(list);
  sort2(list);
  print(list);
else
  return;
```

- Decision/Branch Node:
  - size < THRESHOLD
  - size >= THRESHOLD

- Exit Node: 0 outgoing edges
Control Flow Graphs

- Many types of nodes:

  - **Entry Node**: 0 incoming edges
  - **Exit Node**: 0 outgoing edges
  - **Decision/Branch Node**: >1 outgoing edges
    - If `size < THRESHOLD`:
      - `sort1(list)`
      - `print(list)`
    - If `size >= THRESHOLD`:
      - `sort2(list)`
  - **Join Node**: >1 incoming edges
  - **Return Node**: 1 outgoing edge
Control Flow Graphs

- Straight-line sequences of code are grouped into basic blocks:

```java
list = ...

if (list.size() < THRESHOLD)
    sort1(list);
    sort2(list);
print(list);
size < THRESHOLD
size >= THRESHOLD

return;
```
Control Flow Graphs

- Straight-line sequences of code are grouped into **basic blocks**:

```java
list = ...
if (list.size() < THRESHOLD)
    sort1(list);
    sort2(list);
print(list);
return;
```

size < THRESHOLD  size >= THRESHOLD

- `sort1(list);`
- `sort2(list);`
- `print(list);`
- `return;`
Control Flow Graphs

- Many patterns arise from common constructs
Control Flow Graphs

- Many patterns arise from common constructs
- What is the CFG for this program?

```c
q = 0;
r = x;
while r >= y {
    r = r - y;
    q = q + 1;
}
```

Tell me how to draw it.
Control Flow Graphs

- What is the CFG?

```c
for (i = 0; i < n; i++) {
    foo(i);
}
```
Control Flow Graphs

- What is the CFG?
  - Don't forget implicit behavior like `default`!

```java
switch (x) {
    case a: foo(x); break;
    case b: bar(x);
    case c: baz(x); break;
}
```
Control Flow Graphs

• What is the CFG?
  – Short circuit operators can lead to subtle behavior!

```java
if (x == 0 || y/x > 1) {
    foo(x, y);
}
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!
    From Ammann & Offutt:

```plaintext
x = 0;
while (x < y) {
  y = f (x, y);
  if (y == 0) {
    break;
  } else if (y < 0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!

From Ammann & Offutt:

```java
x = 0;
while (x < y) {
  y = f (x, y);
  if (y == 0) {
    break;
  } else if (y < 0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```
Control Flow Graphs

- What is the CFG?
  - Control flow can become complex!
  
  From Ammann & Offutt:

```plaintext
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!

From Ammann & Offutt:

```c
x = 0;
while (x < y) {
    y = f(x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
Control Flow Graphs

- What is the CFG?
  - Control flow can become complex!
  
  From Ammann & Offutt:

```plaintext
x = 0;
while (x < y) {
    y = f(x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print(y);
```
Control Flow Graphs

- What is the CFG?
  - Control flow can become complex!

From Ammann & Offutt:

```c
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!
    From Ammann & Offutt:

```
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!

From Ammann & Offutt:

```
x = 0;
while (x < y) {
  y = f (x, y);
  if (y == 0) {
    break;
  } else if (y < 0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```
Control Flow Graphs

- What is the CFG?
  - Control flow can become complex!
    From Ammann & Offutt:

```plaintext
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
Control Flow Graphs

• What is the CFG?
  – Control flow can become complex!
    From Ammann & Offutt:

```c
x = 0;
while (x < y) {
    y = f (x, y);
    if (y == 0) {
        break;
    } else if (y < 0) {
        y = y*2;
        continue;
    }
    x = x + 1;
}
print (y);
```
CFG Coverage

- Statement Coverage → Node Coverage

Try to cover all reachable basic blocks.
CFG Coverage

- Statement Coverage → **Node** Coverage

Thinking in terms of node coverage can be more efficient. **Why?**
CFG Coverage

- Statement Coverage $\rightarrow$ Node Coverage
- Branch Coverage $\rightarrow$ Edge Coverage

Try to cover all reachable paths of length $\leq 1$. 
CFG Coverage

- Statement Coverage $\rightarrow$ Node Coverage
- Branch Coverage $\rightarrow$ Edge Coverage

How do node & edge coverage compare? Why?
CFG Coverage

- Statement Coverage → Node Coverage
- Branch Coverage → Edge Coverage

How do node & edge coverage compare? Why?

Are these notions of coverage enough? Why?
Pragmatic Concerns

- The *goal* is full coverage (of whatever criteria)
Pragmatic Concerns

- The goal is full coverage (of whatever criteria)

Is that reasonable in practice? Why?
Pragmatic Concerns

- The *goal* is full coverage (of whatever criteria)
- We must consider *reachability*
Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider reachability

Try to cover all reachable basic blocks.

Try to cover all reachable paths of length \( \leq 1 \).
Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider reachability
  - Syntactic Reachability
    - Based on the structure of the code

```python
ENTER
...
return;
print("Got Here")
return;
```
Pragmatic Concerns

- The goal is full coverage (of whatever criteria)
- We must consider reachability
  - Syntactic Reachability
    - Based on the structure of the code

```python
ENTER
...
return;
print("Got Here")
return;
```
Pragmatic Concerns

- The *goal* is full coverage (of whatever criteria)
- We must consider *reachability*
  - Syntactic Reachability
    - Based on the structure of the code
  - *Semantic* Reachability
    - Based on the meaning of the code

```java
condition = false;
if (condition)
...
return;
```
Pragmatic Concerns

• The goal is full coverage (of whatever criteria)
• We must consider reachability
  – Syntactic Reachability
    • Based on the structure of the code
  – Semantic Reachability
    • Based on the meaning of the code

```
condition = false;
if (condition)
...
...
return;
```

This can be undecidable!
Pragmatic Concerns

- The *goal* is full coverage (of whatever criteria)
- We must consider *reachability*
  - Syntactic Reachability
    - Based on the structure of the code
  - Semantic Reachability
    - Based on the meaning of the code
- So what do you do in practice?
  - No, really. What have you done in practice?
Pragmatic Concerns

• The goal is full coverage (of whatever criteria)

• We must consider reachability
  – Syntactic Reachability
    • Based on the structure of the code
  – Semantic Reachability
    • Based on the meaning of the code

• So what do you do in practice?
  – No, really. What have you done in practice?
  – Relative degrees of coverage matter (40%? 80%?)
The path taken by each test can matter.

```plaintext
if (condition 1)
  ...
  x = 0;
  ...
  if (condition2)
  ...
  z = y/x;
  ...
  return;
```
• The path taken by each test can matter
The path taken by each test can matter
CFG Coverage

• The path taken by each test can matter

Full edge coverage & no bugs found
CFG Coverage

- The path taken by each test can matter

How can we make sure to find *this* bug?
The path taken by each test can matter

if (condition 1)
...
x = 0;
if (condition2)
...
z = y/x;
return;
The path taken by each test can matter

if (condition 1)
...
x = 0;
if (condition2)
...
z = y/x;
return;
The path taken by each test can matter.

Testing *all paths* exposes this bug.
Path Coverage

- **Complete Path Coverage**
  - Test all paths through the graph
Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?
Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable? Why?
Path Coverage

- **Complete Path Coverage**
  - Test all paths through the graph

Is this reasonable? Why?

Infeasibility
Path Coverage

- Complete Path Coverage
  - Test all paths through the graph

Is this reasonable?
Why?

How many paths?
How does this relate to input based approaches?

Diagram of paths:
- A → B
- B → C
- C → D
- D → E
- E → F
- F → G
- G → H
- H → I
- I → J
- J → K
- K → A
- A → B
- B → C
- C → D
- D → E
- E → F
- F → G
- G → H
- H → I
- I → J
- J → K
- K → A
Path Coverage

- **Complete Path Coverage**
  - Test all paths through the graph

Is this reasonable? Why?

How many paths?
How does this relate to input based approaches?

Intractability
Compromises?

- What could we do instead?
  (How did we handle the input based approaches?)
Compromises?

- **Edge Pair Coverage**
  - Each path of length $\leq 2$ is tested.

- **Specified Path Coverage**
  - Given a number $k$, test $k$ paths
Compromises?

- **Edge Pair Coverage**
  - Each path of length \( \leq 2 \) is tested.

- **Specified Path Coverage**
  - Given a number \( k \), test \( k \) paths

What do these look like?
Compromises?

- **Edge Pair Coverage**
  - Each path of length <= 2 is tested.
- **Specified Path Coverage**
  - Given a number $k$, test $k$ paths

What do these look like? Are they good?
Coping With Loops

- What criteria do you use when testing loops?
Coping With Loops

- What criteria do you use when testing loops?
- **Simple Paths**
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program
Coping With Loops

• What criteria do you use when testing loops?

• Simple Paths
  – A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  – Captures the acyclic behaviors of a program

How many may there be?
Coping With Loops

- What criteria do you use when testing loops?
- **Simple Paths**
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program

How many may there be?
Coping With Loops

- What criteria do you use when testing loops?
- **Simple Paths**
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program

How many may there be?

A B C D
Coping With Loops

- What criteria do you use when testing loops?

- **Simple Paths**
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program

- **Prime Paths**
  - A simple path that is not a subpath of any other simple path
Coping With Loops

- What criteria do you use when testing loops?
- **Simple Paths**
  - A path between nodes is simple if no node appears more than once. (Except maybe the first and last)
  - Captures the acyclic behaviors of a program
- **Prime Paths**
  - A simple path that is not a subpath of any other simple path
Coping With Loops

- Prime Path Coverage
  - Cover all prime paths
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Example from Ammann & Offutt
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Example from Ammann & Offutt
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Example from Ammann & Offutt
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Example from Ammann & Offutt

What are the prime paths?

How many simple paths?

Can you intuitively explain what prime paths capture?
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Are these tests good or bad?
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Are these tests good or bad?

Do they address all of the problems with path coverage?
Coping With Loops

- **Prime Path Coverage**
  - Cover all prime paths

Are these tests good or bad?

Do they address all of the problems with path coverage?

Can you think of things they miss?
How Do They Relate?

All / Complete Path Coverage

Prime Path Coverage

How do they compare w/ edge coverage?
How Do They Relate?

All / Complete Path Coverage

Prime Path Coverage

Edge Coverage

Node Coverage
How Do They Relate?

- All / Complete Path Coverage
- Prime Path Coverage
- Edge Coverage
- Node Coverage

How do they compare with edge pair coverage?
How Do They Relate?

- All / Complete Path Coverage
- Edge Pair Coverage
- Prime Path Coverage
- Edge Coverage
- Node Coverage
How Do They Relate?

- All / Complete Path Coverage
- Edge Pair Coverage
- Prime Path Coverage
- Edge Coverage
- Node Coverage

Consider:

A → B
How Do They Relate?

All / Complete Path Coverage

Edge Pair Coverage

Prime Path Coverage

Edge Coverage

Node Coverage

Why did we look at prime paths at all?
How Do They Relate?

All / Complete Path Coverage

Edge Pair Coverage  

Prime Path Coverage

Edge Coverage

Node Coverage

Why did we look at prime paths at all?

So let's only consider loops....
How Do They Relate?

**Round Trip**

- A prime path starting and ending with the same node

- **All / Complete Path Coverage**
- **Prime Path Coverage**
- **Complete Round Trip Coverage**
- **Simple Round Trip Coverage**
- **Node Coverage**
- **Edge Coverage**
- **Edge Pair Coverage**
How Do They Relate?

- All / Complete Path Coverage
- Edge Pair Coverage
- Prime Path Coverage
- Edge Coverage
- Node Coverage
- Complete Round Trip Coverage

All round trips for each node
How Do They Relate?

- All / Complete Path Coverage
  - Edge Pair Coverage
    - Edge Coverage
      - Node Coverage
  - Prime Path Coverage
    - Complete Round Trip Coverage
    - Simple Round Trip Coverage

Just one round trip for each node
Turning Them Into Tests

• Reconsider:

![Diagram with nodes A, B, C, D, G, E, F, and arrows between them]
Turning Them Into Tests

• Reconsider:

Is this path prime?
Turning Them Into Tests

• Reconsider:

Is this path prime?

Is it still useful?
Turning Them Into Tests

- Reconsider:

  One test may cover multiple prime paths!

  Requirements ≠ Tests
Turning Them Into Tests

- Relaxing our path requirements can help, too
Turning Them Into Tests

- Relaxing our path requirements can help, too
- **Tour**
  - A path $p$ tours path $q$ if $q$ is a subpath of $p$
  - A test covers any prime path it tours
Turning Them Into Tests

- Relaxing our path requirements can help, too

Tour
- A path $p$ tours path $q$ if $q$ is a subpath of $p$
- A test covers any prime path it tours

This is strict! Can we relax it?
Turning Them Into Tests

- Relaxing our path requirements can help, too

- **Tour**
  - A path $p$ tours path $q$ if $q$ is a subpath of $p$
  - A test covers any prime path it tours

- **Tour with sidetrips**
  - Iff every *edge* of $q$ appears in the same order in $p$
  - “Return to where you left”
Turning Them Into Tests

- Relaxing our path requirements can help, too

  **Tour**
  - A path $p$ tours path $q$ if $q$ is a subpath of $p$
  - A test covers any prime path it tours

- **Tour with sidetrips**
  - Iff every *edge* of $q$ appears in the same order in $p$
  - “Return to where you left”

- **Tour with detours**
  - Iff every *node* of $q$ appears in the same order in $p$
Turning Them Into Tests

Strict Tour: ABCDE
Turning Them Into Tests

Strict Tour: ABCDE
Turning Them Into Tests

Strict Tour: ABCDE

With Sidetrips?
Turning Them Into Tests

Strict Tour: ABCDE

With Sidetrips?
Turning Them Into Tests

Strict Tour: ABCDE

With Sidetrips?

With Detours?
Turning Them Into Tests

Strict Tour: ABCDE

With Sidetrips?

With Detours?
Turning Them Into Tests

- Do these relaxations help us with problems we have seen?
Turning Them Into Tests

- Do these relaxations help us with problems we have seen?
- Can you see any problems they may introduce?
Turning Them Into Tests

- Do these relaxations help us with problems we have seen?
- Can you see any problems they may introduce?
- How might this affect how you use them?
Onward

- But sometimes the structure of the CFG is not enough...