

CSE P 504

Advanced topics in Software Systems

Fall 2022

Mutation-based Testing

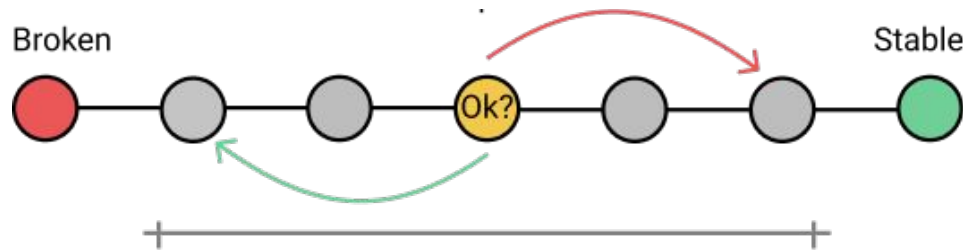
October 24, 2022

Today

- Recap: Git bisect exercise
- Mutation-based testing
 - The basics
 - Productive mutants
 - Mutant subsumption
- Coverage-based vs. mutation-based testing
- In-class exercise 3

Recap: Git bisect

- **Git bisect run-time complexity is always $O(\log(n))$**



Mutation-based testing: the basics

Mutation testing: mutant generation



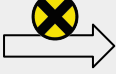
```
public class DrumburgSettings { private static *  
 * // add state variable here  
 *  
 private int condition;  
 *  
 * // add initialization of state variable here  
 *  
 public void DrumburgSettings() {  
     condition = 0;  
 }  
 *  
 * // add initio initialization here  
 *  
 public void initializeInitio 21 {  
     ((DrumburgSettings) getInitio(1)).condition = 1;  
 }  
 *  
 * // add drawing order definition here  
 *  
 * // setting null means the call will not be done  
 *  
 public Color getColor() {  
     return condition == 1  
         ? case 1: return Color.BLUE;  
         : case 2: return Color.RED;  
 }  
 *  
 * // set to null  
 *  
 * // add state variable copy code here  
 *  
 public void copyState() {  
     DrumburgSettings dest = (DrumburgSettings)  
         condition = dest.condition;  
 }  
 *  
 * // add traversal function code here  
 *  
 * // skipping the root state of the call  
 *  
 public void traverseRootCall() {  
     if (condition == 1)  
         condition = condition - 1;  
     } else {  
         state neighbor() = null; getNeighbor();  
         for (int i = 0; i < neighbor.length; i++) {  
             if ((DrumburgSettings) neighbor(i)).condition == 1)  
                 break;  
         }  
     }  
 }  
 *  
 }
```

Program



Mutation testing



Lhs < rhs  *Lhs <= rhs*
Lhs < rhs  *Lhs != rhs*
stmt  *no-op*

Mutation operators

Mutation testing: test creation

```
public void changeSettings(int newI, int newO) {
    // (1) add private variable here
}

/**
 * (2) add initialization of private variable here
 */
public void initializeInstance() {
}

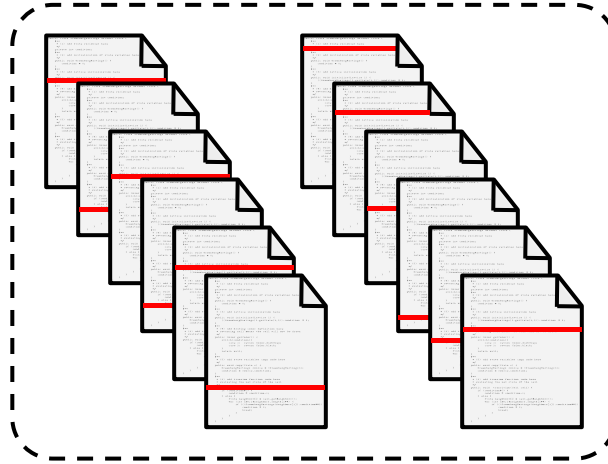
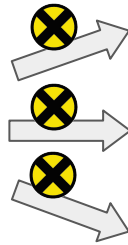
/**
 * (3) add instance initialization here
 */
public void initializeInstance() {
}

/**
 * (4) add clearing state definition here
 * (5) add clearing null before the call will not be done
 */
public void getOutput() {
    // (6) add private variable copy code here
}

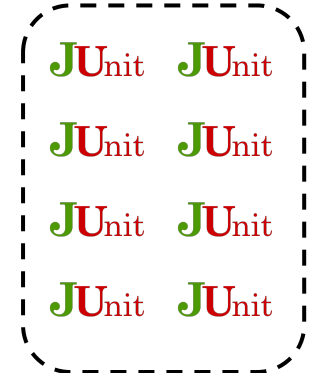
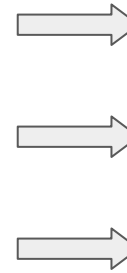
// (7) add private variable copy code here
}

/**
 * (8) add private variable copy code here
 */
public void transformOutput() {
    // (9) add private variable copy code here
}
}
```

Program



Mutants



Tests

Assumptions

- Mutants are coupled to real faults
- Mutant detection is correlated with real-fault detection

https://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf,

https://homes.cs.washington.edu/~rjust/publ/mutation_testing_practices_icse_2021.pdf

Mutation testing: a concrete example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 1:

```
public int min(int a, int b) {  
    return a;  
}
```

Mutation testing: another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 2:

```
public int min(int a, int b) {  
    return b;  
}
```

Mutation testing: yet another example

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 3:

```
public int min(int a, int b) {  
    return a >= b ? a : b;  
}
```


Mutation testing: last example (I promise)

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutant 4:

```
public int min(int a, int b) {  
    return a <= b ? a : b;  
}
```

Mutation testing: exercise



Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

For each mutant, provide a test case that detects it (i.e., passes on the original program but fails on the mutant)

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

M4 cannot be detected (equivalent mutant).

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: exercise

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

Which mutant(s) should we show to a developer?

a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: summary

Original program:

```
public int min(int a, int b) {  
    return a < b ? a : b;  
}
```

Mutants:

M1: return a;

M2: return b;

M3: return a >= b ? a : b;

M4: return a <= b ? a : b;

Redundant

Equivalent

<i>a</i>	<i>b</i>	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation testing: challenges

- Redundant mutants
 - Inflate the mutant detection ratio
 - Hard to assess progress and remaining effort
- Equivalent mutants
 - Max mutant detection ratio \neq 100%
 - Waste resources (CPU and human time)

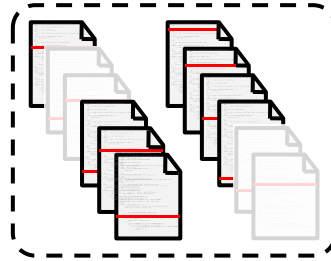
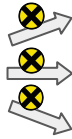
a	b	Original	M1	M2	M3	M4
1	2	1	1	2	2	1
1	1	1	1	1	1	1
2	1	1	2	1	2	1

Mutation Testing vs. Mutation Analysis

Mutation Testing



PROGRAM



MUTANTS



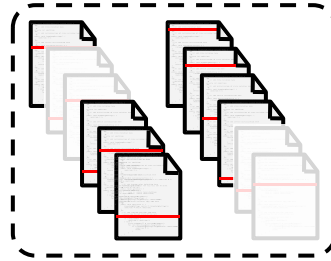
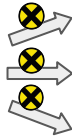
TESTS

Mutation Testing vs. Mutation Analysis

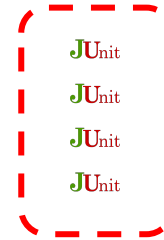
**Mutation
Testing**



PROGRAM



MUTANTS



TESTS

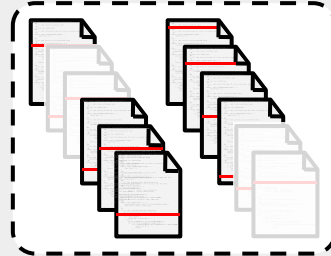
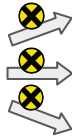
**Primary output is
new tests.**

Mutation Testing vs. Mutation Analysis

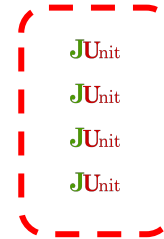
Mutation Testing



PROGRAM



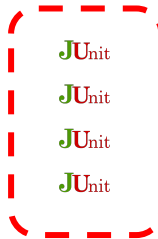
MUTANTS



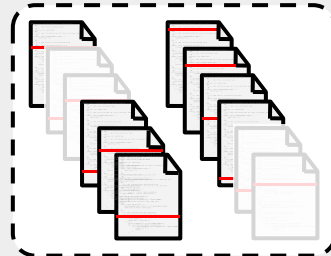
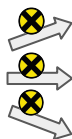
TESTS

Primary output is **new tests.**

Mutation Analysis



TESTS PROGRAM



MUTANTS

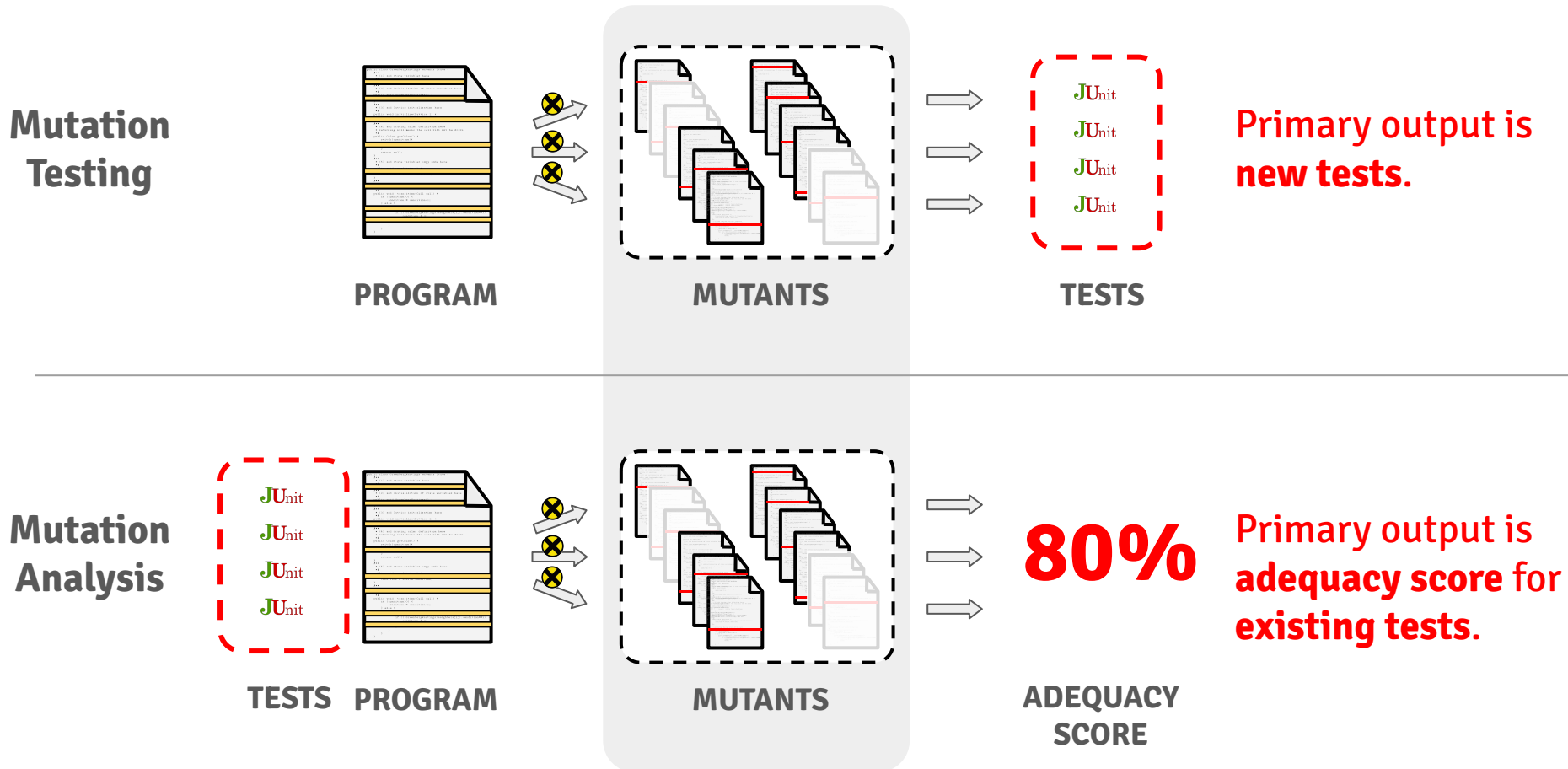


80%

ADEQUACY SCORE

Primary output is **adequacy score for existing tests.**

Mutation Testing vs. Mutation Analysis



How expensive is mutation testing?
Is the mutation score meaningful?

Mutation-based testing: productive mutants

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** \implies **tests**
- **Equivalent** mutants are **bad** \implies **no tests**

A more nuanced view

- **Detectable vs. equivalent** is **too simplistic**
- **Productive mutants** elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

**The core question here concerns test-goal utility
(applies to any adequacy criterion).**

Detectable vs. productive mutants

Historically

- **Detectable** mutants are **good** \implies **tests**
- **Equivalent** mutants are **bad** \implies **no tests**

A more nuanced view

- **Detectable vs. equivalent** is **too simplistic**
- **Productive mutants** elicit effective tests, but
 - detectable mutants can be useless, and
 - equivalent mutants can be useful!

The notion of productive mutants is fuzzy!

A mutant is **productive** if it is

1. **detectable** and **elicits an effective test** or
2. **equivalent** and **advances code quality or knowledge**

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
  if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading 3](#))

Productive mutants: mutation testing at Google

```
int RunMe(int a, int b) {  
  if (a == b || b == 1) {
```

▼ Mutants

14:25, 28 Mar

Changing this 1 line to

```
if (a != b || b == 1) {
```

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

[Please fix](#)

[Not useful](#)

Practical Mutation Testing at Scale: A view from Google ([Reading 3](#))

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant is **detectable**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (1)

Original program

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    double sum = 0;  
    int len = nums.length;  
  
    for (int i = 0; i < len; ++i) {  
        sum = sum * nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is **detectable**, but is it **productive**? **Yes!**

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Is the mutant **detectable**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {
    int len = nums.length;
    double sum = 0;
    double avg = 0;

    for (int i = 0; i < len; ++i) {
        avg = avg + (nums[i] / len);
        sum = sum + nums[i];
    }

    return sum / len;
}
```

Mutant

```
public double getAvg(double[] nums) {
    int len = nums.length;
    double sum = 0;
    double avg = 0;

    for (int i = 0; i < len; ++i) {
        avg = avg * (nums[i] / len);
        sum = sum + nums[i];
    }

    return sum / len;
}
```

The mutant is **not detectable**, but is it **unproductive**?

Detectable vs. productive mutants (2)

Original program

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg + (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

Mutant

```
public double getAvg(double[] nums) {  
    int len = nums.length;  
    double sum = 0;  
    double avg = 0;  
  
    for (int i = 0; i < len; ++i) {  
        avg = avg * (nums[i] / len);  
        sum = sum + nums[i];  
    }  
  
    return sum / len;  
}
```

The mutant is not detectable, but is it unproductive? No!

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

Is the mutant **detectable**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**?

Detectable vs. productive mutants (3)

Original program

```
...  
Set cache = new HashSet(a * b);  
...
```

Mutant

```
...  
Set cache = new HashSet(a + b);  
...
```

The mutant is **detectable**, but is it **productive**? **No!**

Mutation-based testing: mutant subsumption

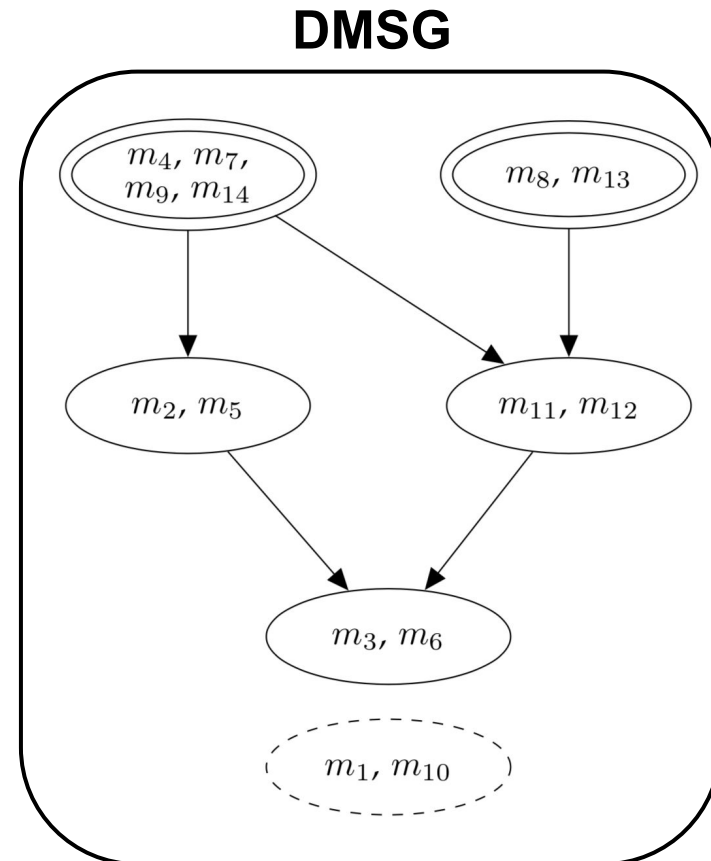
Mutant subsumption

Mutant	Tests					
	MutOp	t_1	t_2	t_3	t_4	
$m_1: < \mapsto !=$		○	○	○	○	Mutant detected (assertion)
$m_2: < \mapsto ==$		○	●	○	●	
$m_3: < \mapsto <=$		★	★	★	★	Mutant detected (exception)
$m_4: < \mapsto >$		○	●	○	○	
$m_5: < \mapsto >=$		○	●	○	●	Mutant not detected
$m_6: < \mapsto \text{true}$		★	★	★	★	
$m_7: < \mapsto \text{false}$		○	●	○	○	
$m_8: < \mapsto !=$		●	○	○	○	
$m_9: < \mapsto ==$		○	●	○	○	
$m_{10}: < \mapsto <=$		○	○	○	○	
$m_{11}: < \mapsto >$		●	●	○	○	
$m_{12}: < \mapsto >=$		●	●	○	○	
$m_{13}: < \mapsto \text{true}$		●	○	○	○	
$m_{14}: < \mapsto \text{false}$		○	●	○	○	

Prioritizing Mutants to Guide Mutation Testing ([Reading 2](#))

DMSG: Dynamic Mutant Subsumption Graph

Mutant	Tests				
	MutOp	t_1	t_2	t_3	t_4
m_1 : $< \mapsto !=$		●	●	●	●
m_2 : $< \mapsto ==$		●	●	●	●
m_3 : $< \mapsto <=$		★	★	★	★
m_4 : $< \mapsto >$		●	●	●	●
m_5 : $< \mapsto >=$		●	●	●	●
m_6 : $< \mapsto \text{true}$		★	★	★	★
m_7 : $< \mapsto \text{false}$		●	●	●	●
m_8 : $< \mapsto !=$		●	●	●	●
m_9 : $< \mapsto ==$		●	●	●	●
m_{10} : $< \mapsto <=$		●	●	●	●
m_{11} : $< \mapsto >$		●	●	●	●
m_{12} : $< \mapsto >=$		●	●	●	●
m_{13} : $< \mapsto \text{true}$		●	●	●	●
m_{14} : $< \mapsto \text{false}$		●	●	●	●



Coverage-based vs. mutation-based testing

See dedicated [Slides \(4 pages\)](#).