Lecture 6 – Regression Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2026 Fall

Recall

- Symbolic Execution
  - Symbols in place of inputs; a path condition per path; an SMT solver returns the input
  - $hash(\alpha) = \beta$: no theory, so the solver says **UNKNOWN**
- Dynamic Symbolic Execution (DSE)
  - Run on concrete values too, and use them where the solver cannot
  - Search heuristics – DFS, CFDS, CGS, learned; loops and data structures
- Hybrid Analysis – blended, heap snapshots, Concerto, dynamic shortcuts

From Making Tests to Keeping Them

- **So far**: how to **make** a test – random, coverage-guided, search-based, symbolic. Every test we made stays in the suite.

From Making Tests to Keeping Them

- **So far**: how to **make** a test – random, coverage-guided, search-based, symbolic. Every test we made stays in the suite.

- The program **changes** every day, and the suite only **grows**. Running all of it after every change becomes too slow.

From Making Tests to Keeping Them

PLRG

- **So far**: how to **make** a test – random, coverage-guided, search-based, symbolic. Every test we made stays in the suite.

- The program **changes** every day, and the suite only **grows**. Running all of it after every change becomes too slow.

- **Today**: which tests to **drop**, which to **rerun** after a change, and in which **order**.

Contents

Regression Fault

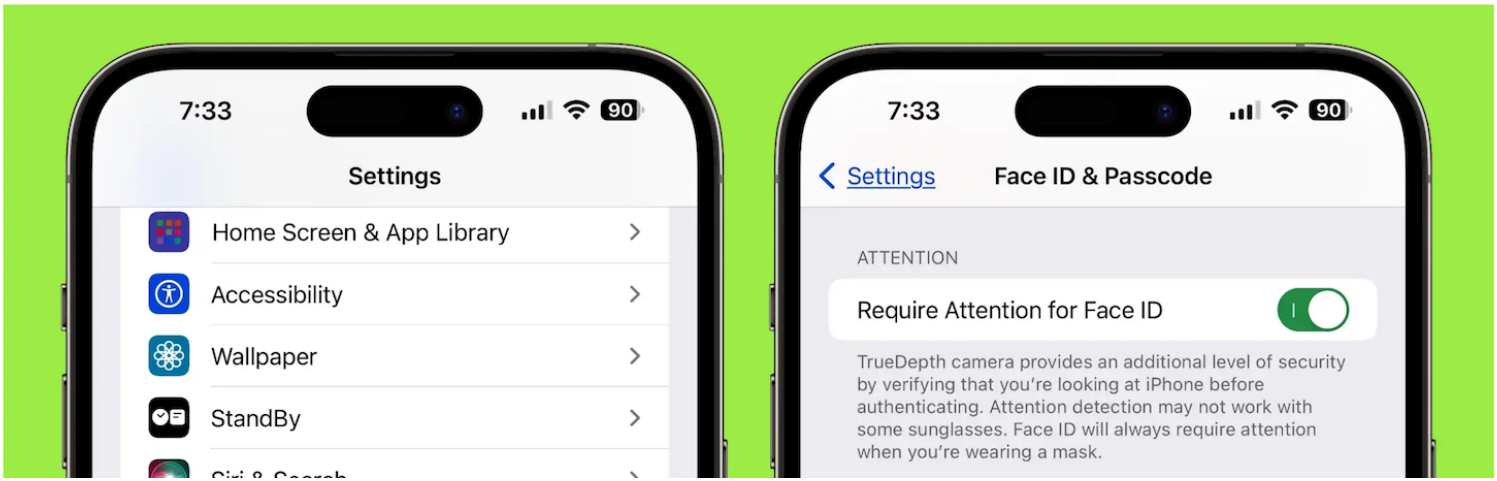
iPhone, iOS 17.4.1 (April 2024)[1]

If your iPhone alarm stopped working in iOS 17.4.1, here's a fix

How-To

Apr 30, 2024

Apple is working to patch the bug but a simple tweak in Settings should help.



[1] **[Macworld, Apr. 30, 2024]** "*If your iPhone alarm stopped working in iOS 17.4.1, here's a fix.*" Apple: "*aware of an issue causing some alarms not play a sound and . . . working on a fix.*"

Regression Fault

Windows 11, June 2026 update (KB5094126)[2]

This article was published on June 21, 2026

DATA AND SECURITY

Microsoft's June update fixed 208 security flaws and introduced a cascade of new bugs across all Windows versions

KB5094126 patched 208 security vulnerabilities on June 9, but the update has triggered Recycle Bin display glitches, BitLocker lockouts on enterprise hardware, OneDrive failures, and system freezes, with the first fix not expected until July 14

June 21, 2026 - 1:59 pm

[2] **[TNW, June 21, 2026]** "*Microsoft's June update fixed 208 security flaws and introduced a cascade of new bugs across all Windows versions.*"

Regression Fault

- **Regression fault** is a fault that occurs when a **change** in the software introduces a new defect or reactivates a defect that had been previously fixed.

- The term **regression** refers to the fact that the software has **regressed** (gone backwards) to an earlier bad state.

Regression Testing

- You are releasing a **new version** of your software.

Regression Testing

PLRG

- You are releasing a **new version** of your software.
- You have tried to thoroughly **test the new features**.

Regression Testing

PLRG

- You are releasing a **new version** of your software.
- You have tried to thoroughly **test the new features**.
- You want to check if you have created any **regression faults**.

Regression Testing

PLRG

- You are releasing a **new version** of your software.
- You have tried to thoroughly **test the new features**.
- You want to check if you have created any **regression faults**.
- How would you do that?

Regression Testing – Retest All

- The simplest way is to **retest all** the test cases.

Regression Testing – Retest All

- The simplest way is to **retest all** the test cases.
- You need to run all tests not only for the **new features** but also for the **old existing features**.

Regression Testing – Retest All

- The simplest way is to **retest all** the test cases.
- You need to run all tests not only for the **new features** but also for the **old existing features**.
- Its main disadvantage is that it is **time-consuming** to run all the test cases for every new version.

Regression Testing – Retest All

- The simplest way is to **retest all** the test cases.
- You need to run all tests not only for the **new features** but also for the **old existing features**.
- Its main disadvantage is that it is **time-consuming** to run all the test cases for every new version.
- It is **critical** in the modern software development process because software is **continuously and rapidly changing**.

Regression Testing – Retest All

"For example, one of our industrial collaborators reports that for one of its products of about **20,000 lines of code**, *the entire test suite requires* **seven weeks** *to run."*

– Rothermel et al.[3]

[3] **[TSE'01]** G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. "*Prioritizing Test Cases for Regression Testing.*"

Regression Testing – Retest All

> *"For example, one of our industrial collaborators reports that for one of its products of about* **20,000 lines of code**, *the entire test suite requires* **seven weeks** *to run."*
>
> *– Rothermel et al.*[3]

The **test suite** becomes **larger** and **larger** as the software evolves with the following factors:

- Long Product History
- Different Configurations
- Types of Test Cases

[3] **[TSE'01]** G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. "*Prioritizing Test Cases for Regression Testing.*"

Regression Testing Techniques

Many techniques have been developed in order to cope with the high cost of retesting all test cases. They can be categorized into:

- **Test Suite Minimization**
- **Test Case Selection**
- **Test Case Prioritization**

Test Suite Minimization

- **Problem** – Regression test suite is **too large**

Test Suite Minimization

- **Problem** – Regression test suite is **too large**
- **Idea** – There might be some test cases that are **redundant**

Test Suite Minimization

- **Problem** – Regression test suite is **too large**
- **Idea** – There might be some test cases that are **redundant**
- **Solution** – **Minimize** regression test suite by removing all the redundant test cases

Test Suite Minimization

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – There might be some test cases that are **redundant**
- **Solution** – **Minimize** regression test suite by removing all the redundant test cases
- Then, what is the definition of a **redundant** test case?

Test Suite Minimization

PLRG

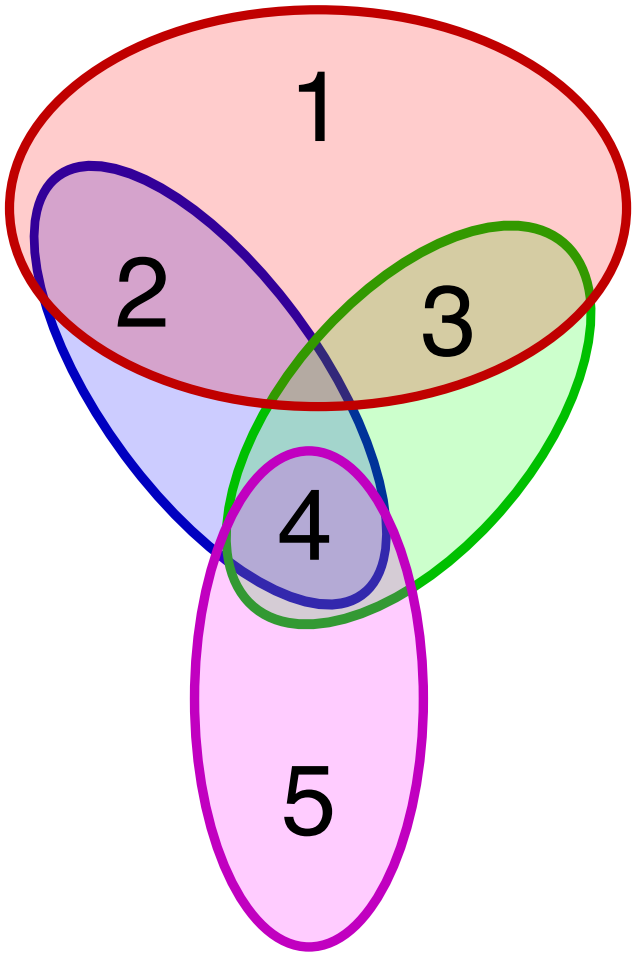
One possible way is to use the **coverage** information.

(e.g., Statements, MC/DC, All-DU-Paths, etc.)

TC	r_1	r_2	r_3	$\cdots$
t_1	✓	✓		$\cdots$
t_2		✓		$\cdots$
t_3		✓	✓	$\cdots$
t_4			✓	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

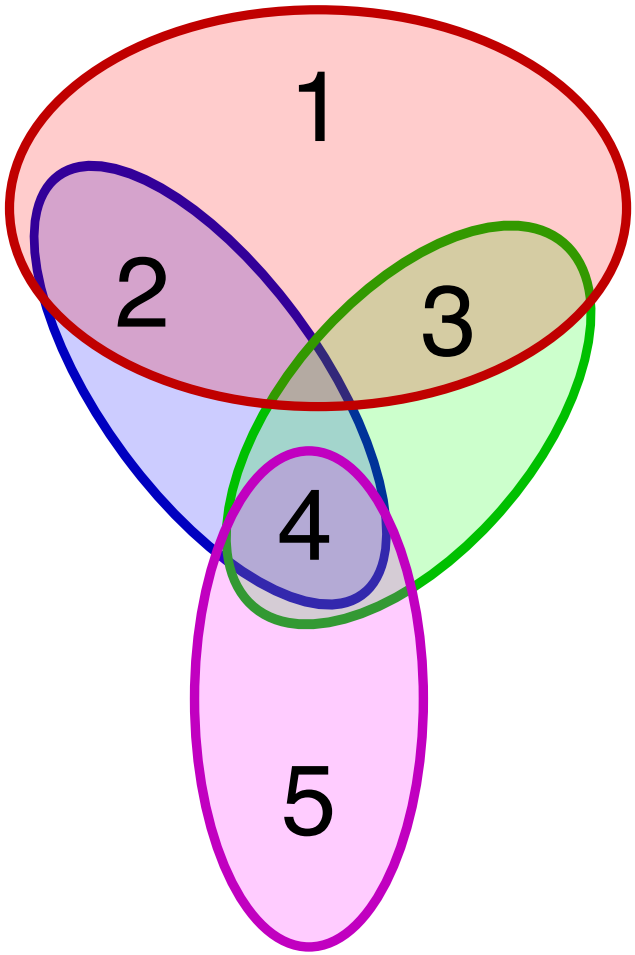
How to **minimize** the test suite as much as possible while **preserving** the coverage information (the set of covered test requirements)?

Test Suite Minimization



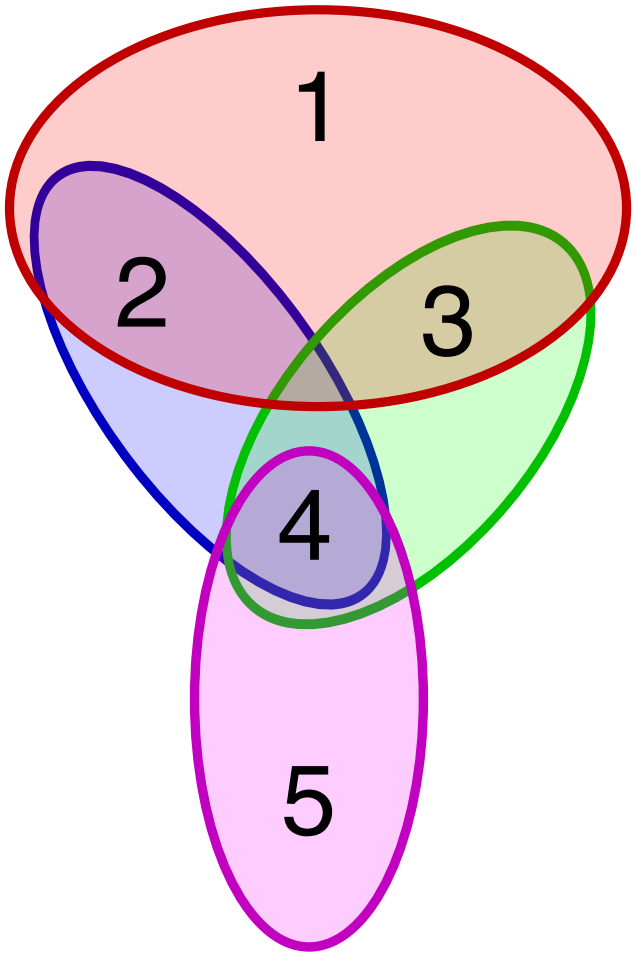
- We can model the **test suite minimization** problem as a **set cover** problem.

Test Suite Minimization



- We can model the **test suite minimization** problem as a **set cover** problem.
- Unfortunately, the set cover problem is **NP-complete**, meaning that there is no known polynomial-time algorithm to solve it.

Test Suite Minimization



- We can model the **test suite minimization** problem as a **set cover** problem.
- Unfortunately, the set cover problem is **NP-complete**, meaning that there is no known polynomial-time algorithm to solve it.
- However, there are many **heuristic** algorithms that can be used to solve the test suite minimization problem.

Greedy Minimization

Algorithm Greedy Minimization

```
1: function GreedyMinimization(T, R)
2:     T' ← ∅
3:     while true do
4:         t ← argmax_{t∈T} |R ∩ TR(t)|
5:         if |R ∩ TR(t)| = 0 then
6:             break
7:         T' ← T' ∪ {t}
8:         R ← R \ TR(t)
9:     return T'
```

Greedy Minimization

Algorithm Greedy Minimization

```
1: function GreedyMinimization(T, R)
2:     T' ← ∅
3:     while true do
4:         t ← argmax_{t∈T} |R ∩ TR(t)|
5:         if |R ∩ TR(t)| = 0 then
6:             break
7:         T' ← T' ∪ {t}
8:         R ← R \ TR(t)
9:     return T'
```

TC	r_1	r_2	r_3	r_4
t_1	✓	✓		
t_2		✓		✓
t_3		✓	✓	✓
t_4			✓	
t_5			✓	✓

$$T' = \varnothing$$
$$R = \{r_1, r_2, r_3, r_4\}$$

Greedy Minimization

Algorithm Greedy Minimization

```
1: function GreedyMinimization(T, R)
2:     T' ← ∅
3:     while true do
4:         t ← argmax_{t∈T} |R ∩ TR(t)|
5:         if |R ∩ TR(t)| = 0 then
6:             break
7:         T' ← T' ∪ {t}
8:         R ← R \ TR(t)
9:     return T'
```

TC	r_1	r_2	r_3	r_4
t_1	✓	✓		
t_2		✓		✓
t_3		✓	✓	✓
t_4			✓	
t_5			✓	✓

$$T' = \varnothing$$
$$R = \{r_1, r_2, r_3, r_4\}$$
$$T' = \{t_3\}$$
$$R = \{r_1\}$$

Greedy Minimization

Algorithm Greedy Minimization

```
1: function GreedyMinimization(T, R)
2:     T' ← ∅
3:     while true do
4:         t ← argmax_{t∈T} |R ∩ TR(t)|
5:         if |R ∩ TR(t)| = 0 then
6:             break
7:         T' ← T' ∪ {t}
8:         R ← R \ TR(t)
9:     return T'
```

TC	r_1	r_2	r_3	r_4
t_1	✓	✓		
t_2		✓		✓
t_3		✓	✓	✓
t_4			✓	
t_5			✓	✓

$$T' = \varnothing$$
$$R = \{r_1, r_2, r_3, r_4\}$$

$$T' = \{t_3\}$$
$$R = \{r_1\}$$

$$T' = \{t_3, t_1\}$$
$$R = \varnothing$$

Greedy Minimization with Cost

PLRG

- However, in fact, test cases have different **costs** to run.

Greedy Minimization with Cost

- However, in fact, test cases have different **costs** to run.

- Is $\{t_3, t_1\}$ still the best solution?

TC	r_1	r_2	r_3	r_4	*Time*
t_1	✓	✓			3
t_2		✓		✓	5
t_3		✓	✓	✓	10
t_4			✓		2
t_5			✓	✓	8

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

1: **function** GreedyMinimization(T, R)
2: $T' \leftarrow \emptyset$
3: **while** `true` **do**
4: $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
5: **if** $|R \cap \text{TR}(t)| = 0$ **then**
6: **break**
7: $T' \leftarrow T' \cup \{t\}$
8: $R \leftarrow R \setminus \text{TR}(t)$
9: **return** T'

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

1: **function** GreedyMinimization(T, R)
2: $T' \leftarrow \emptyset$
3: **while** `true` **do**
4: $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
5: **if** $|R \cap \text{TR}(t)| = 0$ **then**
6: **break**
7: $T' \leftarrow T' \cup \{t\}$
8: $R \leftarrow R \setminus \text{TR}(t)$
9: **return** T'

TC	r_1	r_2	r_3	r_4	*Time*
t_1	✓	✓			3
t_2		✓		✓	5
t_3		✓	✓	✓	10
t_4			✓		2
t_5			✓	✓	8

$T' = \varnothing$ $R = \{r_1, r_2, r_3, r_4\}$

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

1: **function** GreedyMinimization(T, R)
2: $T' \leftarrow \emptyset$
3: **while** `true` **do**
4: $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
5: **if** $|R \cap \text{TR}(t)| = 0$ **then**
6: **break**
7: $T' \leftarrow T' \cup \{t\}$
8: $R \leftarrow R \setminus \text{TR}(t)$
9: **return** T'

TC	r_1	r_2	r_3	r_4	*Time*
t_1	✓	✓			3
t_2		✓		✓	5
t_3		✓	✓	✓	10
t_4			✓		2
t_5			✓	✓	8

$T' = \varnothing$ $R = \{r_1, r_2, r_3, r_4\}$

$T' = \{t_1\}$ $R = \{r_3, r_4\}$

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

1: **function** GreedyMinimization(T, R)
2: $T' \leftarrow \emptyset$
3: **while** `true` **do**
4: $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
5: **if** $|R \cap \text{TR}(t)| = 0$ **then**
6: **break**
7: $T' \leftarrow T' \cup \{t\}$
8: $R \leftarrow R \setminus \text{TR}(t)$
9: **return** T'

TC	r_1	r_2	r_3	r_4	*Time*
t_1	✓	✓			3
t_2		✓		✓	5
t_3		✓	✓	✓	10
t_4			✓		2
t_5			✓	✓	8

$T' = \varnothing$ $R = \{r_1, r_2, r_3, r_4\}$

$T' = \{t_1\}$ $R = \{r_3, r_4\}$

$T' = \{t_1, t_4\}$ $R = \{r_4\}$

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

1: **function** GreedyMinimization(T, R)
2: $T' \leftarrow \emptyset$
3: **while** `true` **do**
4: $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
5: **if** $|R \cap \text{TR}(t)| = 0$ **then**
6: **break**
7: $T' \leftarrow T' \cup \{t\}$
8: $R \leftarrow R \setminus \text{TR}(t)$
9: **return** T'

TC	r_1	r_2	r_3	r_4	*Time*
t_1	✓	✓			3
t_2		✓		✓	5
t_3		✓	✓	✓	10
t_4			✓		2
t_5			✓	✓	8

$T' = \varnothing$ $R = \{r_1, r_2, r_3, r_4\}$

$T' = \{t_1\}$ $R = \{r_3, r_4\}$

$T' = \{t_1, t_4\}$ $R = \{r_4\}$

$T' = \{t_1, t_4, t_2\}$ $R = \varnothing$

Greedy Minimization with Score

- Another possible approach is to use the **score** information and keep the **best test case** for each test requirement according to the score.

Greedy Minimization with Score

- Another possible approach is to use the **score** information and keep the **best test case** for each test requirement according to the score.

- The **score** of each test case can be defined in various ways.

Greedy Minimization with Score

PLRG

- Another possible approach is to use the **score** information and keep the **best test case** for each test requirement according to the score.
- The **score** of each test case can be defined in various ways.
- For example, consider a **conformance test suite** for JavaScript engines: each test case is a JavaScript program with assertions.

Greedy Minimization with Score

PLRG

- Another possible approach is to use the **score** information and keep the **best test case** for each test requirement according to the score.
- The **score** of each test case can be defined in various ways.
- For example, consider a **conformance test suite** for JavaScript engines: each test case is a JavaScript program with assertions.
- Then, which test case is better to keep?

Greedy Minimization with Score

PLRG

- Another possible approach is to use the **score** information and keep the **best test case** for each test requirement according to the score.
- The **score** of each test case can be defined in various ways.
- For example, consider a **conformance test suite** for JavaScript engines: each test case is a JavaScript program with assertions.
- Then, which test case is better to keep?
- We can define the score of each test case based on the **complexity** of the test case (e.g., **size of the program**, etc.) because a simpler test case is more understandable and maintainable.

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \left\{ \right.$$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \begin{cases} r_1 \mapsto t_1 \\ \\ \\ \\ \end{cases}$$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \begin{cases} r_1 \mapsto t_1 \\ r_2 \mapsto t_1 \end{cases}$$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \begin{cases} r_1 \mapsto t_1 \\ r_2 \mapsto t_1 \\ r_3 \mapsto t_4 \end{cases}$$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \left\{ \begin{array}{l} r_1 \mapsto t_1 \\ r_2 \mapsto t_1 \\ r_3 \mapsto t_4 \\ r_4 \mapsto t_5 \end{array} \right\}$$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

```
1: function GreedyMinimization(T, R)
2:     M ← ∅
3:     for t ∈ T do
4:         for r ∈ TR(t) do
5:             if r ∉ M ∨ Score(t) > Score(M[r]) then
6:                 M[r] ← t
7:     T′ ← {t | (r, t) ∈ M}
8:     return T′
```

TC	r_1	r_2	r_3	r_4	*Score*
t_1	✓	✓			4
t_2		✓		✓	3
t_3		✓	✓	✓	2
t_4			✓		9
t_5			✓	✓	5

$$M = \left\{ \begin{array}{l} r_1 \mapsto t_1 \\ r_2 \mapsto t_1 \\ r_3 \mapsto t_4 \\ r_4 \mapsto t_5 \end{array} \right\}$$

$$T' = \{t_1, t_4, t_5\}$$

Greedy Minimization with Score

PLRG

We can minimize the test suite even during the **test case generation**. [4]

[4] **[ICSE'21]** J. Park, S. An, D. Youn, G. Kim, S. Ryu. "*JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification.*"

Greedy Minimization with Score

We can minimize the test suite even during the **test case generation**. [4]

RelationalExpression **:** *RelationalExpression* **<=** *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*).
5. Let *r* be ? IsLessThan(*rval*, *lval*, **false**).
6. If *r* is either **true** or **undefined**, return **false**. Otherwise, return **true**.

[4] **[ICSE'21]** J. Park, S. An, D. Youn, G. Kim, S. Ryu. "*JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification.*"

Greedy Minimization with Score

Tests generated from the spec: keep, for each spec feature, the **smallest** program that covers it.[5]

RelationalExpression **:** *RelationalExpression* **<=** *ShiftExpression*

1. Let *lref* be ? Evaluation of *RelationalExpression*.
2. Let *lval* be ? GetValue(*lref*).
3. Let *rref* be ? Evaluation of *ShiftExpression*.
4. Let *rval* be ? GetValue(*rref*). ▶ `1 <= Symbol()`
5. Let *r* be ? IsLessThan(*rval*, *lval*, **false**).
6. If *r* is either **true** or **undefined**, return **false**. Otherwise, return **true**.

[5] **[ICSE'21]** J. Park, S. An, D. Youn, G. Kim, S. Ryu. "*JEST: N+1-version Differential Testing of Both JavaScript Engines and Specification.*"

Multi-Objective Problem

What if the test suite minimization problem has **multiple objectives**?

Multi-Objective Problem

What if the test suite minimization problem has **multiple objectives**?

- "I have **only 5 hours** as a deadline to run the test suite."

Multi-Objective Problem

What if the test suite minimization problem has **multiple objectives**?

- "I have **only 5 hours** as a deadline to run the test suite."
- "I need to cover **more than one coverage** criterion."

Multi-Objective Problem

PLRG

What if the test suite minimization problem has **multiple objectives**?

- "I have **only 5 hours** as a deadline to run the test suite."
- "I need to cover **more than one coverage** criterion."
- "I want to limit the **fault detection loss** of the minimized suite."

$$\left(1 - \frac{\#\text{ Faults Detected by the Minimized Test Suite}}{\#\text{ Faults Detected by the Original Test Suite}}\right) \times 100\%$$

Multi-Objective Problem – Weighted Sum

- The simplest way: **add the objectives up** with weights, and go back to a single-objective problem.

$$o' = \alpha_1 \times o_1 + \alpha_2 \times o_2 + \cdots + \alpha_n \times o_n \qquad \sum_i \alpha_i = 1$$

Multi-Objective Problem – Weighted Sum

- The simplest way: **add the objectives up** with weights, and go back to a single-objective problem.

$$o' = \alpha_1 \times o_1 + \alpha_2 \times o_2 + \cdots + \alpha_n \times o_n \qquad \sum_i \alpha_i = 1$$

- Then, the greedy algorithm picks **one** test suite – the best one **for these weights**.

Multi-Objective Problem – Weighted Sum

- The simplest way: **add the objectives up** with weights, and go back to a single-objective problem.

$$o' = \alpha_1 \times o_1 + \alpha_2 \times o_2 + \cdots + \alpha_n \times o_n \qquad \sum_i \alpha_i = 1$$

- Then, the greedy algorithm picks **one** test suite – the best one **for these weights**.

- But who decides α? Coverage 0.7 and time 0.3? Or 0.5 and 0.5? Each choice gives a **different** suite, and there is no way to tell which weights are right.

Multi-Objective Problem – Pareto Efficiency

- A suite is **dominated** if another suite is at least as good in **every** objective and better in **one**.

Multi-Objective Problem – Pareto Efficiency

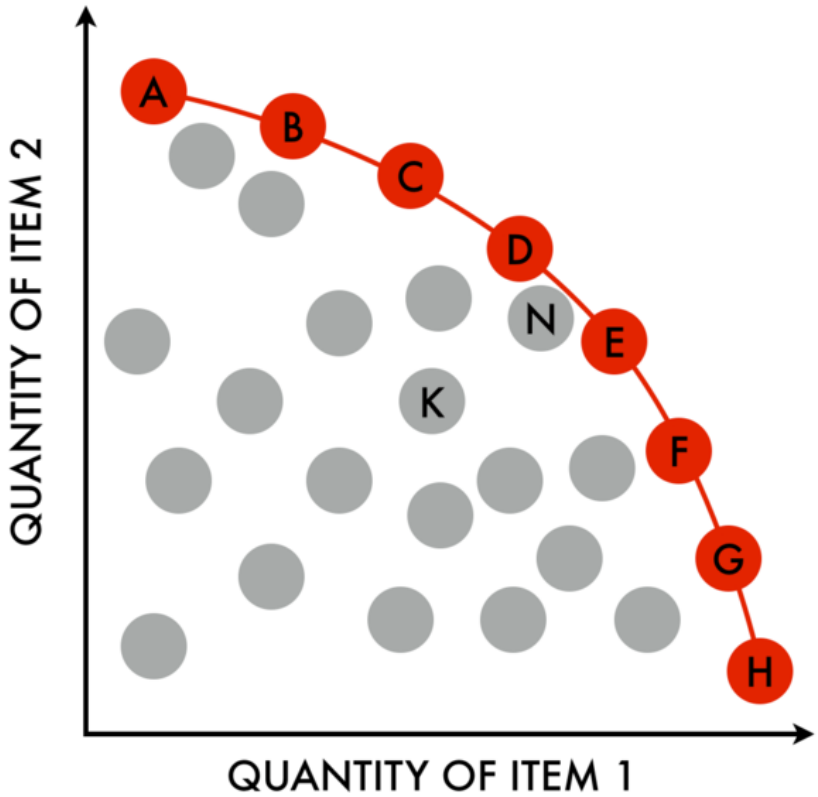
- A suite is **dominated** if another suite is at least as good in **every** objective and better in **one**.
- The suites that **nobody dominates** are **Pareto efficient**. Together they form the **Pareto front**.

Multi-Objective Problem – Pareto Efficiency

- A suite is **dominated** if another suite is at least as good in **every** objective and better in **one**.
- The suites that **nobody dominates** are **Pareto efficient**. Together they form the **Pareto front**.
- Instead of one suite, we get the **whole front** – and decide **later**.

Multi-Objective Problem – Pareto Efficiency

- A suite is **dominated** if another suite is at least as good in **every** objective and better in **one**.
- The suites that **nobody dominates** are **Pareto efficient**. Together they form the **Pareto front**.
- Instead of one suite, we get the **whole front** – and decide **later**.
- "I have **5 hours**": take the front, and pick the point **under 5 hours** with the highest coverage.



Test Case Selection

PLRG

- **Problem** – Regression test suite is **too large**

Test Case Selection

- **Problem** – Regression test suite is **too large**
- **Idea** – Not all tests are **related** to the recent changes in the software.

Test Case Selection

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – Not all tests are **related** to the recent changes in the software.
- **Solution** – Precisely **select** the test cases that actually **execute** the changed parts of the software.

Test Case Selection

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – Not all tests are **related** to the recent changes in the software.
- **Solution** – Precisely **select** the test cases that actually **execute** the changed parts of the software.
- Then, how can we know which test cases are **related** to the recent changes?

Test Case Selection

- We have the original program P and the updated program P'.

Test Case Selection

- We have the original program P and the updated program P'.

- We need to keep track of the **execution trace** of the test cases in the original program P to know which parts of the program are executed by each test case.

Test Case Selection

PLRG

- We have the original program P and the updated program P'.
- We need to keep track of the **execution trace** of the test cases in the original program P to know which parts of the program are executed by each test case.
- Then, we should know **which parts** of the program are **changed** in the updated program P' compared to the original program P.

Test Case Selection – Textual Diff

- Most modern software is developed with a **version control system** (e.g., Git).

Test Case Selection – Textual Diff

- Most modern software is developed with a **version control system** (e.g., Git).
- The easiest way is to use the **diff** command provided by the version control system to know the **changed parts** of the program.

```
191         def codeUnit: AbsCodeUnit
192         def const: AbsConst
193         def math: AbsMath
                                          197 +       def infinity: AbsInfinity
194         def simpleValue: AbsSimpleValue
195         def number: AbsNumber
196         def bigInt: AbsBigInt
```

```
194         def codeUnit: AbsCodeUnit
195         def const: AbsConst
196         def math: AbsMath
198         def simpleValue: AbsSimpleValue
199         def number: AbsNumber
200         def bigInt: AbsBigInt
```

203 src/main/scala/esmeta/analyzer/domain/value/ValueTypeDomain.scala

Viewed

```
@@ -4,7 +4,8 @@ import esmeta.analyzer.*
4   import esmeta.analyzer.domain.*
5   import esmeta.cfg.Func
6   import esmeta.es.*
7 - import esmeta.ir.{COp, Name, VOp,
    MOp}
8   import esmeta.parser.ESValueParser
9   import esmeta.state.*
10  import esmeta.spec.{Grammar => _, *}
```

```
4   import esmeta.analyzer.domain.*
5   import esmeta.cfg.Func
6   import esmeta.es.*
7 + import esmeta.ir.{COp, Name, VOp,
    MOp, UOp}
8 + import esmeta.interpreter.Interpreter
9   import esmeta.parser.ESValueParser
10  import esmeta.state.*
11  import esmeta.spec.{Grammar => _, *}
```

Test Case Selection – Control Flow Graph

A textual diff tells which **lines** changed, not which **tests** reach them.

Test Case Selection – Control Flow Graph

A textual diff tells which **lines** changed, not which **tests** reach them.

P

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

P'

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return 1 - x;
  }
}
```

Test Suite

- $t_1 : x = 1$
- $t_2 : x = -1$

Test Case Selection – Control Flow Graph

A textual diff tells which **lines** changed, not which **tests** reach them.

P

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

P'

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return 1 - x;
  }
}
```

Test Suite

- $t_1 : x = 1$
- $t_2 : x = -1$

Which test case should be **selected**?

Test Case Selection – Control Flow Graph

A textual diff tells which **lines** changed, not which **tests** reach them.

P

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return -x;
  }
}
```

P'

```
function foo(x) {
  if (x >= 0) {
    return x;
  } else {
    return 1 - x;
  }
}
```

Test Suite

- $t_1 : x = 1$
- $t_2 : x = -1$

Which test case should be **selected**?

- A file-level diff says `foo` changed: **both**.
- The trace of t_1 in P never reaches the changed statement: **only** t_2.

Test Case Selection – Safe Selection

- Selection techniques do **not consider** the **cost** of tests. Why?

Test Case Selection – Safe Selection

- Selection techniques do **not consider** the **cost** of tests. Why?

- Because they focus on **safety**: no test related to the change may be missed, or a regression fault slips through.

Test Case Selection – Safe Selection

- Selection techniques do **not consider** the **cost** of tests. Why?
- Because they focus on **safety**: no test related to the change may be missed, or a regression fault slips through.
- Realistically, we only consider whether each part of the program is **executed** by the test cases or not to check the safety according to their **execution traces**.

Test Case Selection – Difficulties

- **Data Collection** – Collecting the **execution traces** of the test cases is not easy. Especially, when the software is **large** and **complex**, and some programs are written in multiple languages.

Test Case Selection – Difficulties

- **Data Collection** – Collecting the **execution traces** of the test cases is not easy. Especially, when the software is **large** and **complex**, and some programs are written in multiple languages.

- **Non-executable modifications** – Some modifications are **not executable** (e.g., configuration changes, etc.)

Test Case Selection – Difficulties

- **Data Collection** – Collecting the **execution traces** of the test cases is not easy. Especially, when the software is **large** and **complex**, and some programs are written in multiple languages.

- **Non-executable modifications** – Some modifications are **not executable** (e.g., configuration changes, etc.)

- **Safety can be expensive** – what if the safe selection is still too **large** to run?

Test Case Selection – Safe vs. Predictive

- **Ekstazi** – **safe** selection that people actually use: track which **files** each test touches at run time, and rerun a test only when one of its files changed. No coverage tool, no version-control hook – a JUnit plug-in.[6]
  - 32 projects, 615 revisions: end-to-end test time −32% on average, −54% for long suites.

[6] **[ISSTA'15]** M. Gligoric, L. Eloussi, D. Marinov. "*Practical Regression Test Selection with Dynamic File Dependencies.*"

[7] **[ICSE-SEIP'19]** M. Machalica, A. Samylkin, M. Porth, S. Chandra. "*Predictive Test Selection.*"

Test Case Selection – Safe vs. Predictive

- **Ekstazi** – **safe** selection that people actually use: track which **files** each test touches at run time, and rerun a test only when one of its files changed. No coverage tool, no version-control hook – a JUnit plug-in.[6]
  - 32 projects, 615 revisions: end-to-end test time -32% on average, -54% for long suites.
- **Predictive test selection** (Facebook) – give up safety, and **learn** from history which tests a change is likely to **fail**.[7]
  - Testing cost **halved**; still reports $> 95\%$ of test failures and $> 99.9\%$ of faulty changes.
  - Not safe – but the **post-commit** run still catches the rest.

[6] **[ISSTA'15]** M. Gligoric, L. Eloussi, D. Marinov. "*Practical Regression Test Selection with Dynamic File Dependencies.*"

[7] **[ICSE-SEIP'19]** M. Machalica, A. Samylkin, M. Porth, S. Chandra. "*Predictive Test Selection.*"

Test Case Prioritization

- **Problem** – Regression test suite is **too large**

Test Case Prioritization

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – Execute tests following the order of their **importance**.

Test Case Prioritization

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – Execute tests following the order of their **importance**.
- **Solution** – **Prioritize** the test suite so that you get the most out of your regression testing whenever it gets stopped.

Test Case Prioritization

PLRG

- **Problem** – Regression test suite is **too large**
- **Idea** – Execute tests following the order of their **importance**.
- **Solution** – **Prioritize** the test suite so that you get the most out of your regression testing whenever it gets stopped.
- Then, how to define the **importance** of each test case?

Simple Approach

Without using any complicated techniques, we can simply prioritize the test using the following criteria:

- **Backwards** – Newer faults are more likely to be detected by newer tests.

Simple Approach

Without using any complicated techniques, we can simply prioritize the test using the following criteria:

- **Backwards** – Newer faults are more likely to be detected by newer tests.
- **Random** – Randomly select the test cases without any bias.

Simple Approach

Without using any complicated techniques, we can simply prioritize the test using the following criteria:

- **Backwards** – Newer faults are more likely to be detected by newer tests.
- **Random** – Randomly select the test cases without any bias.

If we want to prioritize test cases in a smarter way, what should we consider?

Test Case Prioritization

- Suppose we knew about all the faults in advance and we have the information about the **fault detection capability** of each test case. (Impossible but let's pretend)

Test Case Prioritization

- Suppose we knew about all the faults in advance and we have the information about the **fault detection capability** of each test case. (Impossible but let's pretend)

- Which test should we run first if we knew this? What next?

TC	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

Test Case Prioritization

- Suppose we knew about all the faults in advance and we have the information about the **fault detection capability** of each test case. (Impossible but let's pretend)

- Which test should we run first if we knew this? What next?

TC	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

- Obviously, we need to run t_3 first and then run t_5 as the next.

Test Case Prioritization – Surrogate

- Since we cannot know about the faults in advance, we can use a **surrogate** for the fault detection capability (e.g., coverage, etc.)

Test Case Prioritization – Surrogate

- Since we cannot know about the faults in advance, we can use a **surrogate** for the fault detection capability (e.g., coverage, etc.)
- Let's consider the **coverage** information as a **surrogate** for the fault detection capability.

TC	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

Test Case Prioritization – Surrogate

- Since we cannot know about the faults in advance, we can use a **surrogate** for the fault detection capability (e.g., coverage, etc.)
- Let's consider the **coverage** information as a **surrogate** for the fault detection capability.

TC	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

- We still need to run t_3 first and then run t_5 as the next.

Test Case Prioritization – Surrogate

- Since we cannot know about the faults in advance, we can use a **surrogate** for the fault detection capability (e.g., coverage, etc.)
- Let's consider the **coverage** information as a **surrogate** for the fault detection capability.

TC	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

- We still need to run t_3 first and then run t_5 as the next.
- However, since the full coverage does **not guarantee** full fault detection, we still need to run remaining test cases after **resetting** the coverage information after running t_3 and t_5.

Test Case Prioritization – Surrogate

- Since we cannot know about the faults in advance, we can use a **surrogate** for the fault detection capability (e.g., coverage, etc.)
- Let's consider the **coverage** information as a **surrogate** for the fault detection capability.

TC	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}
t_1	✓				✓					
t_2	✓				✓	✓	✓			
t_3	✓	✓	✓	✓	✓	✓	✓			
t_4					✓					
t_5								✓	✓	✓

- We still need to run t_3 first and then run t_5 as the next.
- However, since the full coverage does **not guarantee** full fault detection, we still need to run remaining test cases after **resetting** the coverage information after running t_3 and t_5.
- So, we need to run them in the order of t_2, t_1, and t_4.

Average Percentage of Faults Detected (APFD)

PLRG

- We can measure the effectiveness of the test case prioritization based on the **average percentage of faults detected** (APFD).[8]

[8] **[TSE'01]** G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. "*Prioritizing Test Cases for Regression Testing.*"

Average Percentage of Faults Detected (APFD)

PLRG

- We can measure the effectiveness of the test case prioritization based on the **average percentage of faults detected** (APFD).[8]
- Intuitively, APFD evaluates the effectiveness of the test case based on the **area** under the curve of the **fault detection rate**.

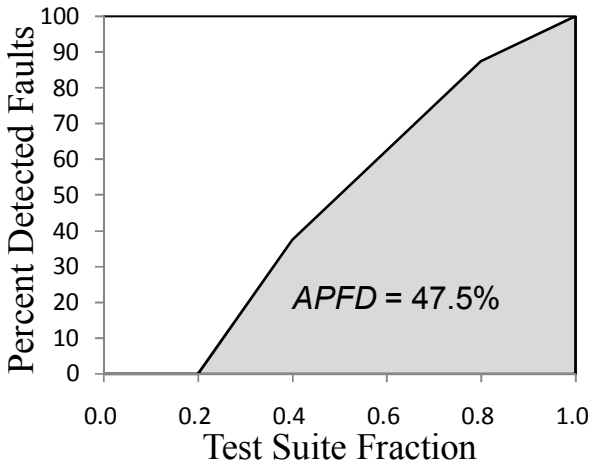
[8] **[TSE'01]** G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. "*Prioritizing Test Cases for Regression Testing.*"

Average Percentage of Faults Detected (APFD)

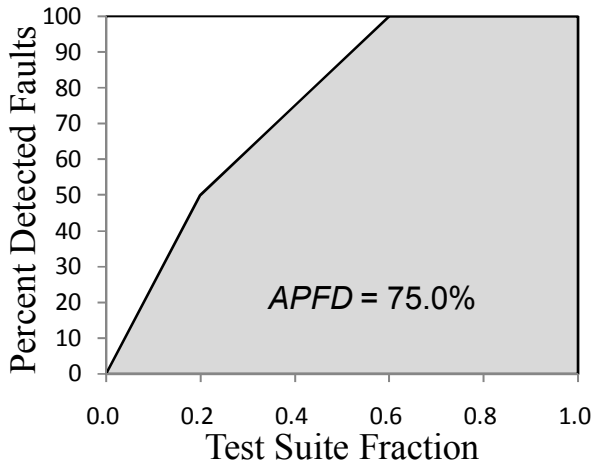
- We can measure the effectiveness of the test case prioritization based on the **average percentage of faults detected** (APFD).[8]
- Intuitively, APFD evaluates the effectiveness of the test case based on the **area** under the curve of the **fault detection rate**.

Example on test suite and faults exposed

Test Case	Fault							
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
t_A	•		•					•
t_B								
t_C		•		•	•			•
t_D	•	•				•		
t_E			•				•	



(a) *APFD* for test suite T_1



(b) *APFD* for test suite T_2

[8] **[TSE'01]** G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold. "*Prioritizing Test Cases for Regression Testing.*"

Average Percentage of Faults Detected (APFD)

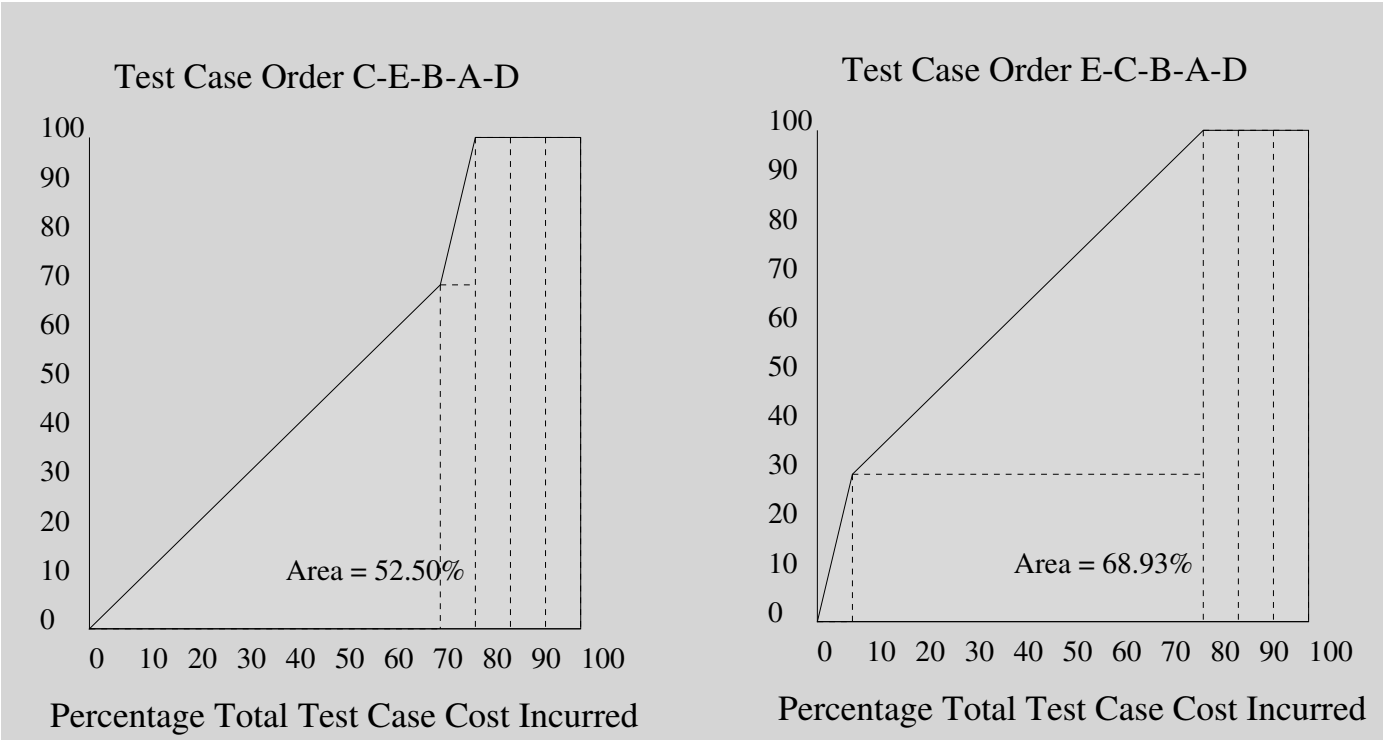
PLRG

- **APFDc** is a variant of APFD that weighs each test case by its **cost**: the x-axis is the test cost incurred, not the test count.[9]

[9] **[ICSE'01]** S. Elbaum, A. Malishevsky, G. Rothermel. "*Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization.*"

Average Percentage of Faults Detected (APFD)

- **APFDc** is a variant of APFD that weighs each test case by its **cost**: the x-axis is the test cost incurred, not the test count.[9]



[9] **[ICSE'01]** S. Elbaum, A. Malishevsky, G. Rothermel. "*Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization.*"

Test Case Prioritization – Clustering

- A technique that **groups** objects such that objects in the same group are the most similar to each other.

Test Case Prioritization – Clustering

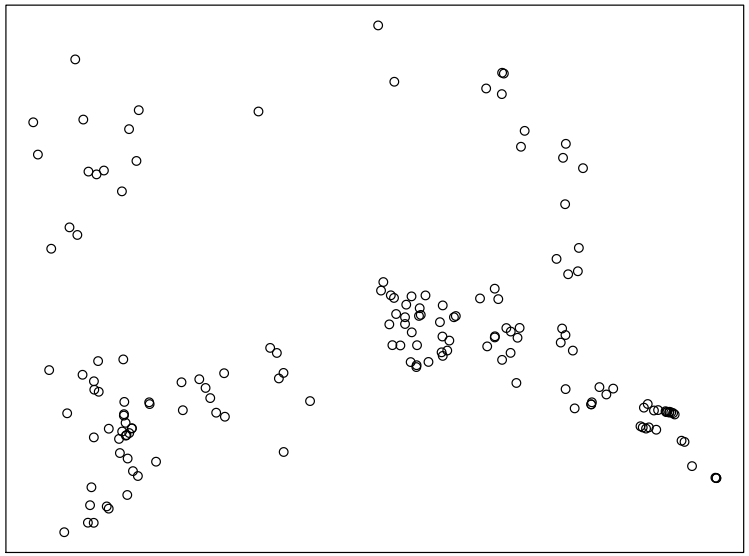
- A technique that **groups** objects such that objects in the same group are the most similar to each other.
- Reduces the conceptual size of test suites by **clustering** test cases that are similar to each other.

Test Case Prioritization – Clustering

- A technique that **groups** objects such that objects in the same group are the most similar to each other.
- Reduces the conceptual size of test suites by **clustering** test cases that are similar to each other.
- Provides insights into what is the **most common behavior** of the test cases.

Test Case Prioritization – Clustering

- A technique that **groups** objects such that objects in the same group are the most similar to each other.
- Reduces the conceptual size of test suites by **clustering** test cases that are similar to each other.
- Provides insights into what is the **most common behavior** of the test cases.
- We can define a **distance** function between test cases based on diverse properties (e.g., coverage, etc.) and **visualize** the clustering results.



Test Case Prioritization – Clustering

Cluster the test cases by their **coverage** (agglomerative hierarchical clustering), then let an expert rank the **clusters**, not the tests.[10]

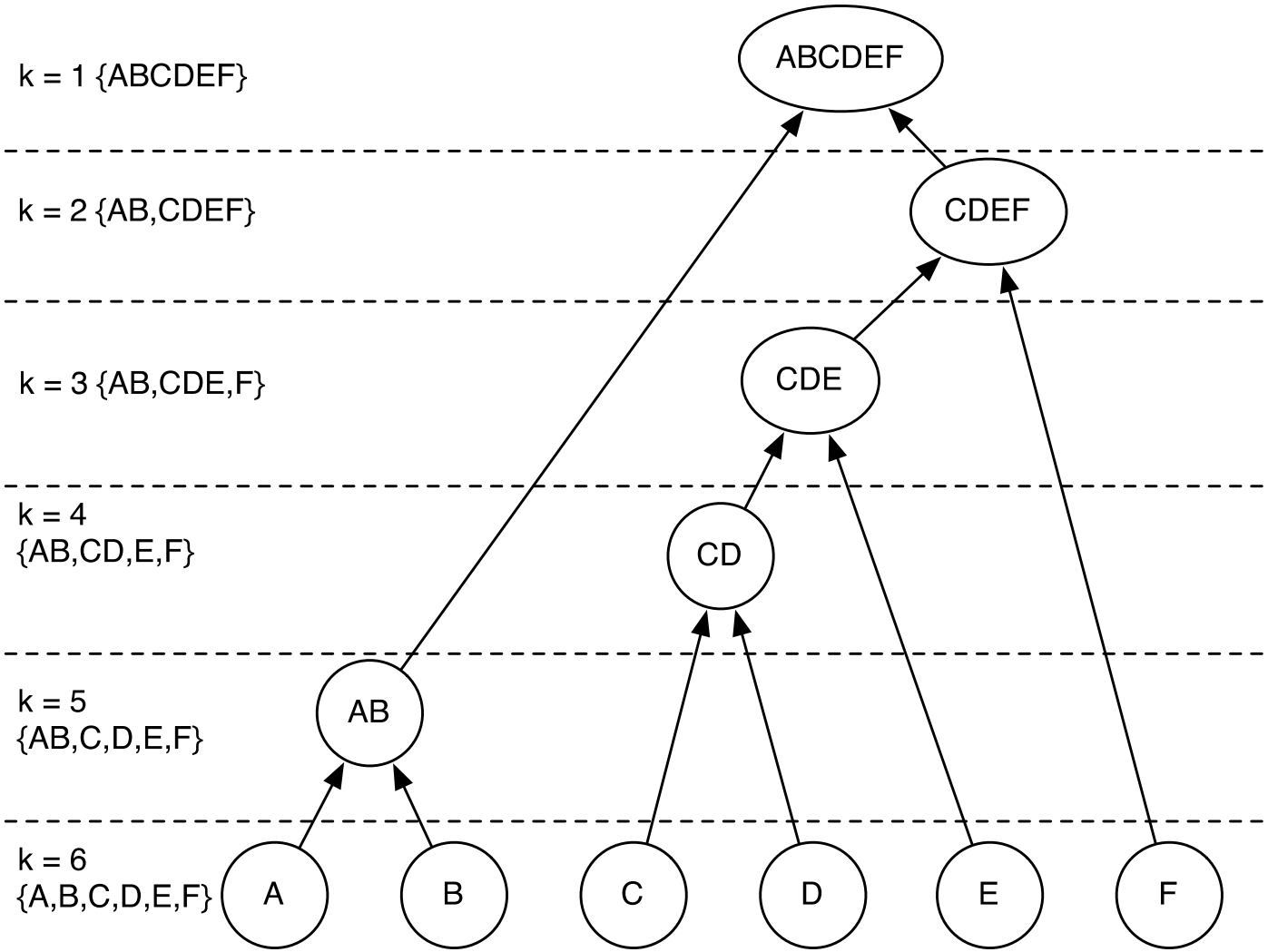
Algorithm 1: Agglomerative Hierarchical Clustering
Input: A set of n test cases, T
Output: A dendrogram, D, representing the clusters

(1) Form n clusters, each with one test case
(2) $C \leftarrow \{\}$
(3) Add clusters to C
(4) Insert n clusters as leaf node into D
(5) **while** there is more than one cluster
(6) Find a pair of clusters with minimum distance
(7) Merge the pair into a new cluster, c_{new}
(8) Remove the pair of test cases from C
(9) Add c_{new} to C
(10) Insert c_{new} as a parent node of the pair into D
(11) **return** D

[10] **[ISSTA'09]** S. Yoo, M. Harman, P. Tonella, A. Susi. "*Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge.*"

Test Case Prioritization – Clustering

Cut the **dendrogram** at a height to get the clusters.[11]



[11] **[ISSTA'09]** S. Yoo, M. Harman, P. Tonella, A. Susi. "*Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge.*"

Test Case Prioritization – Clustering

Interleave: order the clusters, then take one test from each in turn.[12]

Algorithm 2: Interleaved Clusters Prioritisation
Input: An ordered set of k ordered clusters, OOC
Output: An ordered set of test cases, OTC

(1) $OTC =<>$
(2) $i \leftarrow 1$
(3) **while** OOC is not empty
(4) Append $OOC_i(1)$ to OTC
(5) Remove $OOC_i(1)$ from OOC_i
(6) **if** OOC_i is empty **then** Remove OOC_i from OOC
(7) $i \leftarrow (i + 1) \bmod k$
(8) **return** OOC

$$OOC = [\quad \{A, B\}, \quad \{C, D, E\}, \quad \{F\} \quad]$$

[12] **[ISSTA'09]** S. Yoo, M. Harman, P. Tonella, A. Susi. "*Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge.*"

Test Case Prioritization – Similarity

Clustering makes sense if we can measure **similarity** between test cases.

[13] **[ICSE'19]** E. Cruciani, B. Miranda, R. Verdecchia, A. Bertolino. "*Scalable Approaches for Test Suite Reduction.*"

Test Case Prioritization – Similarity

Clustering makes sense if we can measure **similarity** between test cases. **FAST-R**: embed each test as a vector of its **tokens**, and sample tests that are far apart – no coverage needed.[13]

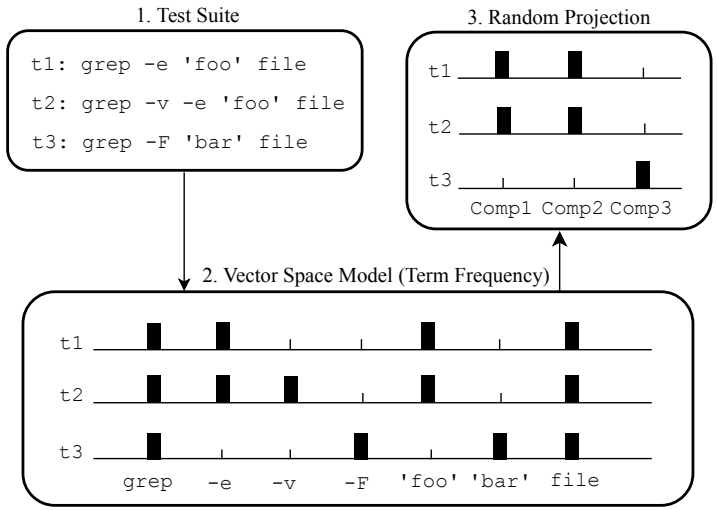


Fig. 1: Visual representation of *FAST-R* preparation phase.

State of the art methods are still largely based on **syntactic** measures (e.g., code coverage, etc.)

[13] **[ICSE'19]** E. Cruciani, B. Miranda, R. Verdecchia, A. Bertolino. "*Scalable Approaches for Test Suite Reduction.*"

Test Case Prioritization – Similarity

Clustering makes sense if we can measure **similarity** between test cases. **FAST-R**: embed each test as a vector of its **tokens**, and sample tests that are far apart – no coverage needed.[13]

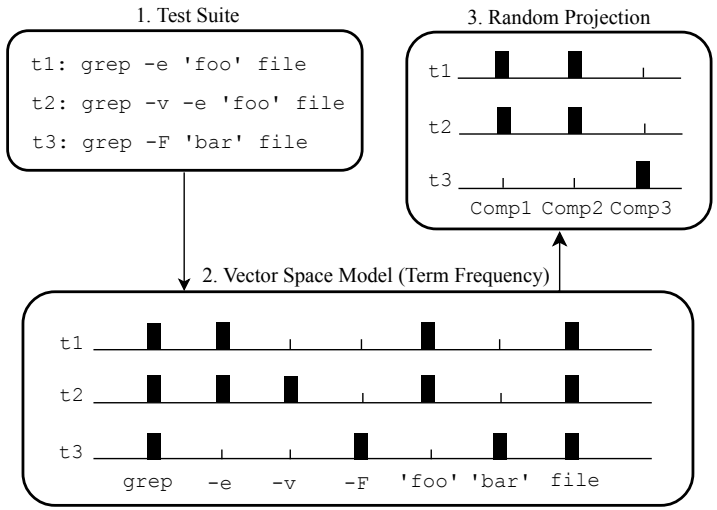


Fig. 1: Visual representation of *FAST-R* preparation phase.

State of the art methods are still largely based on **syntactic** measures (e.g., code coverage, etc.) Can we define a **semantic** measure of similarity between test cases?

[13] **[ICSE'19]** E. Cruciani, B. Miranda, R. Verdecchia, A. Bertolino. "*Scalable Approaches for Test Suite Reduction.*"

Test Case Prioritization – Learning in CI

- In CI there is a new commit every few minutes and **no time** to collect coverage. What is left? The **history** of each test.

[14] **[ISSTA'17]** H. Spieker, A. Gotlieb, D. Marijan, M. Mossige. "*Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration.*"

Test Case Prioritization – Learning in CI

- In CI there is a new commit every few minutes and **no time** to collect coverage. What is left? The **history** of each test.
- **RETECS** – a **reinforcement learning** agent ranks each test from three numbers: its **duration**, when it **last ran**, and its recent **failures**.[14]
  - **Reward**: failures found early in this cycle. The agent learns from every CI run – no model of the code at all.
  - Three industrial datasets (ABB, Google): after a few hundred cycles it beats the fixed heuristics.

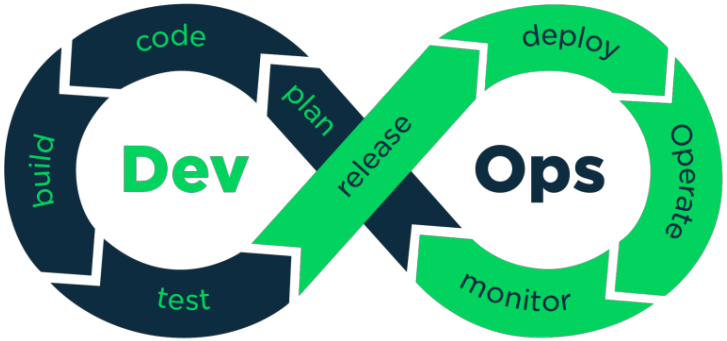
[14] **[ISSTA'17]** H. Spieker, A. Gotlieb, D. Marijan, M. Mossige. "*Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration.*"

Test Case Prioritization – Learning in CI

- In CI there is a new commit every few minutes and **no time** to collect coverage. What is left? The **history** of each test.
- **RETECS** – a **reinforcement learning** agent ranks each test from three numbers: its **duration**, when it **last ran**, and its recent **failures**.[14]
  - **Reward**: failures found early in this cycle. The agent learns from every CI run – no model of the code at all.
  - Three industrial datasets (ABB, Google): after a few hundred cycles it beats the fixed heuristics.
- The surrogate is no longer coverage. It is **"this test failed recently"** – and that is what the next slide scales up to commits.

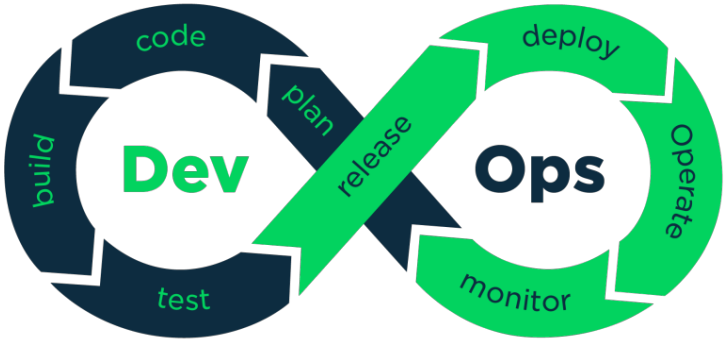
[14] **[ISSTA'17]** H. Spieker, A. Gotlieb, D. Marijan, M. Mossige. "*Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration.*"

Regression Testing in Practice



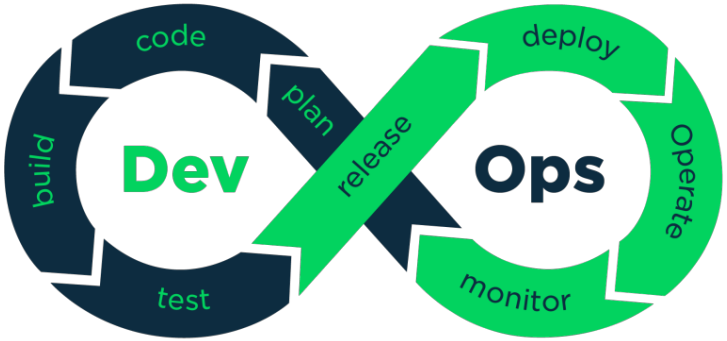
- A **DevOps** concept popularized by Google, more commonly and also known as: **Continuous Integration and Deployment (CI/CD)**

Regression Testing in Practice



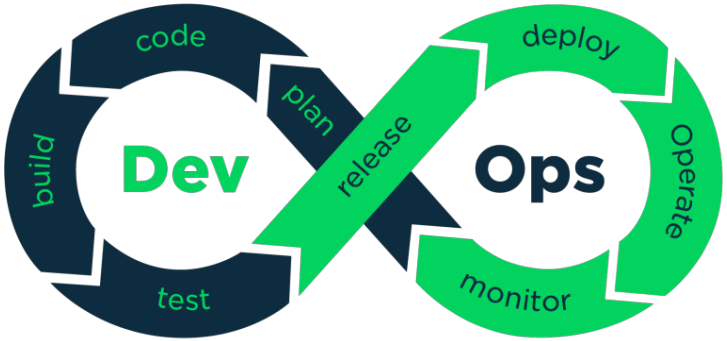
- A **DevOps** concept popularized by Google, more commonly and also known as: **Continuous Integration and Deployment (CI/CD)**
- Newest version of software is automatically deployed whenever **all tests pass**.

Regression Testing in Practice



- A **DevOps** concept popularized by Google, more commonly and also known as: **Continuous Integration and Deployment (CI/CD)**
- Newest version of software is automatically deployed whenever **all tests pass**.
- Developers usually ensure that their commits are correct by executing test cases that are directly relevant at their **local machines**. This is sometimes called **pre-commit testing**.

Regression Testing in Practice



- A **DevOps** concept popularized by Google, more commonly and also known as: **Continuous Integration and Deployment (CI/CD)**
- Newest version of software is automatically deployed whenever **all tests pass**.
- Developers usually ensure that their commits are correct by executing test cases that are directly relevant at their **local machines**. This is sometimes called **pre-commit testing**.
- Once changes are merged, the **CI system** automatically executes all relevant test cases, to ensure that individual changes correctly work with each other. This is sometimes called **post-commit testing**.

Regression Testing in Practice

- All regression testing techniques are limited by the **cost** of running the test cases.

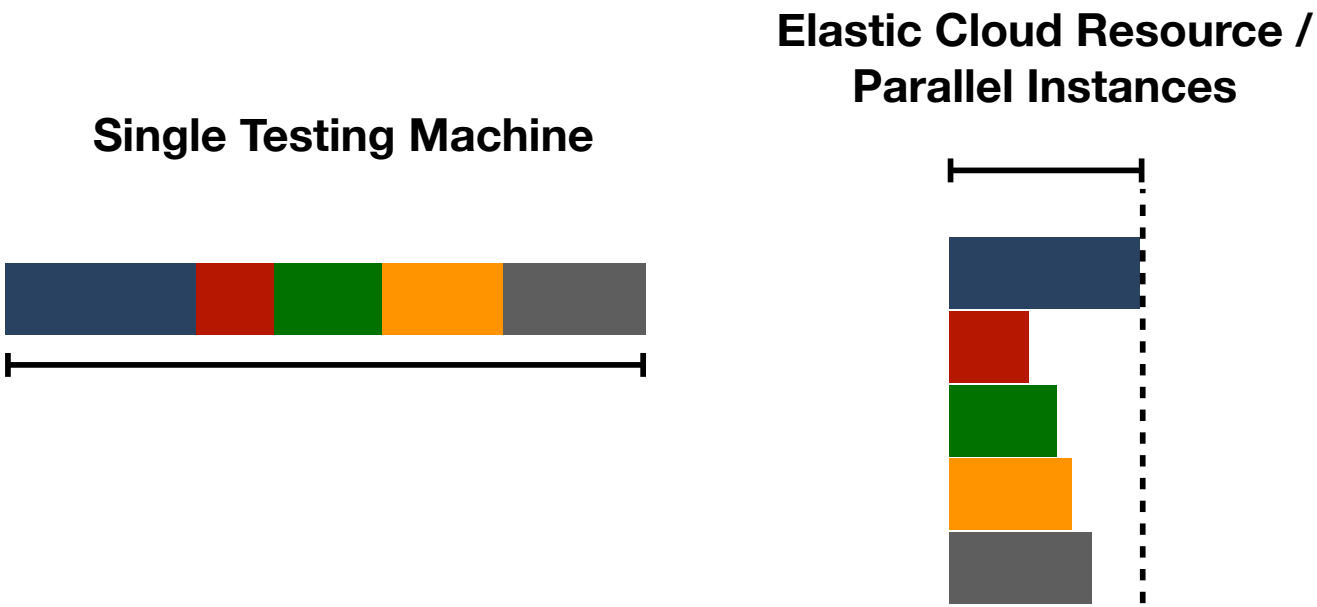
Regression Testing in Practice

PLRG

- All regression testing techniques are limited by the **cost** of running the test cases.
- In practice, we execute test cases in a **distributed** manner to minimize the cost of running the test cases.

Regression Testing in Practice

- All regression testing techniques are limited by the **cost** of running the test cases.
- In practice, we execute test cases in a **distributed** manner to minimize the cost of running the test cases.
- Then, the testing time is only limited by the **slowest** test case.



Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.

[15] **[Velocity'11]** J. Jenkins. "*Velocity Culture.*" Amazon.com.

[16] **[ICSE-SEIP'17]** A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco. "*Taming Google-Scale Continuous Testing.*"

[17] **[ICSE'18]** J. Liang, S. Elbaum, G. Rothermel. "*Redefining Prioritization: Continuous Prioritization for Continuous Integration.*"

Redefining Prioritization

PLRG

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon deployed to production every **11.6 seconds** on average, already in 2011.[15]

[15] **[Velocity'11]** J. Jenkins. "*Velocity Culture.*" Amazon.com.

[16] **[ICSE-SEIP'17]** A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco. "*Taming Google-Scale Continuous Testing.*"

[17] **[ICSE'18]** J. Liang, S. Elbaum, G. Rothermel. "*Redefining Prioritization: Continuous Prioritization for Continuous Integration.*"

Redefining Prioritization

PLRG

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon deployed to production every **11.6 seconds** on average, already in 2011.[15]
  - At Google, developers have waited **up to 9 hours** for the test results of a change.[16]

[15] **[Velocity'11]** J. Jenkins. "*Velocity Culture.*" Amazon.com.

[16] **[ICSE-SEIP'17]** A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco. "*Taming Google-Scale Continuous Testing.*"

[17] **[ICSE'18]** J. Liang, S. Elbaum, G. Rothermel. "*Redefining Prioritization: Continuous Prioritization for Continuous Integration.*"

Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon deployed to production every **11.6 seconds** on average, already in 2011.[15]
  - At Google, developers have waited **up to 9 hours** for the test results of a change.[16]
- **Prioritizing test cases** within one test suite makes little difference at this scale.

[15] **[Velocity'11]** J. Jenkins. "*Velocity Culture.*" Amazon.com.

[16] **[ICSE-SEIP'17]** A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco. "*Taming Google-Scale Continuous Testing.*"

[17] **[ICSE'18]** J. Liang, S. Elbaum, G. Rothermel. "*Redefining Prioritization: Continuous Prioritization for Continuous Integration.*"

Redefining Prioritization

PLRG

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon deployed to production every **11.6 seconds** on average, already in 2011.[15]
  - At Google, developers have waited **up to 9 hours** for the test results of a change.[16]
- **Prioritizing test cases** within one test suite makes little difference at this scale.
- Instead, **prioritize the commits** to test.[17]

---

[15] **[Velocity'11]** J. Jenkins. "*Velocity Culture.*" Amazon.com.

[16] **[ICSE-SEIP'17]** A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, J. Micco. "*Taming Google-Scale Continuous Testing.*"

[17] **[ICSE'18]** J. Liang, S. Elbaum, G. Rothermel. "*Redefining Prioritization: Continuous Prioritization for Continuous Integration.*"

Redefining Prioritization

- The technique is history-based prioritization: **commits** that touch code whose tests have **recently failed** are tested first. If tests really fail, this ensures **quicker feedback** to the responsible developers.

Redefining Prioritization

PLRG

- The technique is history-based prioritization: **commits** that touch code whose tests have **recently failed** are tested first. If tests really fail, this ensures **quicker feedback** to the responsible developers.

- Assumptions

  - Commits are **independent** from each other. In high volume environment, this is not unrealistic.

  - **Relationships** between code and test cases are known in advance (i.e., which test covers which parts of the code).

Summary

- **Minimization**: Keyword is **redundancy** – **Minimize** test suite by removing redundant test cases according to the definition of redundancy.
- **Selection**: Keyword is **safety** – **Select** test cases that are related to the recent changes conservatively to avoid possible regression faults.
- **Prioritization**: Keyword is **surrogate** – **Prioritize** test cases based on the surrogate for the fault detection capability to maximize the fault detection rate early.

Next Lecture

- Mutation Testing

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr