Lecture 9 – Regression Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Recall

- Mutation Testing
  - Fundamental Hypotheses
  - Overall Process
  - Mutation Generation
  - Kill vs Alive
  - Equivalent Mutants
  - How to Kill A Mutant
  - Scalability
  - Higher Order Mutants
  - Tools
- Test Flakiness

Contents

Contents

Regression Fault

Inacio Ribeiro Original Poster Jun 1, 2023

Bug after updating Google Chrome to version 114.0.5735.90

After updating Google Chrome to version 114.0.5735.90 my website started experiencing issues with displaying information. It seems to be related to the CSS applied by the browser, but I'm not certain.

The strange thing is that the release notes for this version only mention security updates. Is anyone else facing this same issue?

https://support.google.com/chrome/thread/218644651/

Regression Fault

PLRG

The current update of IOS 17.0.2 is full of bugs

device- IPAD 9th generation
I have updated it now while studying when I am opening a pdf it always opens from the 1st page, earlier it used open the same page where I used to leave.
Take a screenshot then copy and delete the screenshot and then paste that in the pdf file.. now it is not getting pasted..earlier it used to get copied.

https://discussions.apple.com/thread/255162058

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 realizing a **new version** of your software.

Regression Testing

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

Regression Testing

- You are realizing 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

- You are realizing 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."*

– G.Rothermel, R.H. Untch & M.J. Harrold, *Prioritizing Test Cases for Regression Testing, TSE'21*

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."*

– G.Rothermel, R.H. Untch & M.J. Harrold, *Prioritizing Test Cases for Regression Testing, TSE'21*

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

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

Regression Testing Techniques

PLRG

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

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

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

- Then, what is the definition of a **redundant** test case?

Test Suite Minimization

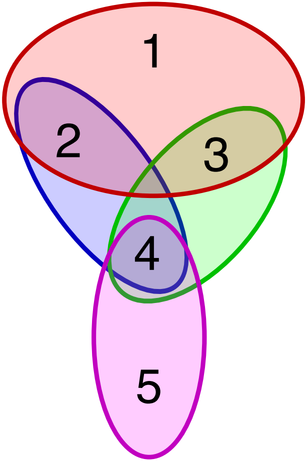
One possible way is to use the **coverage** information.

(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			✓	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

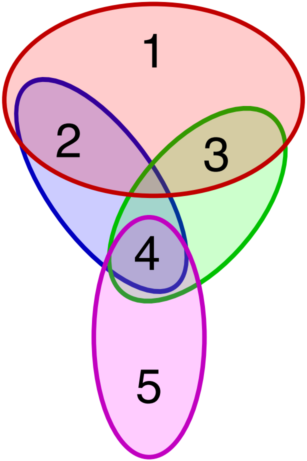
How to **minimize** the test suite as much as possible while **preserving** the coverage information (the set of covered test requirements)?

Test Suite Minimization



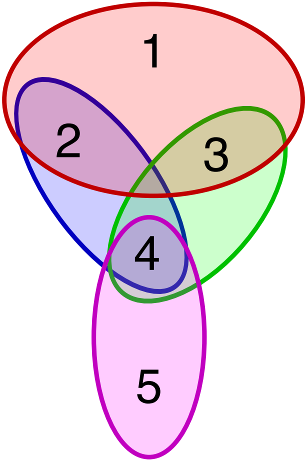
- We can model the **test suite minimization** problem as a **set cover** problem.

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

function $\textsc{GreedyMinimization}(T, R)$
    $T' \leftarrow \emptyset$
    while `true` **do**
        $t \leftarrow \text{argmax}_{t \in T} |R \cap \text{TR}(t)|$
        if $|R \cap \text{TR}(t)| = 0$ **then**
            break
        $T' \leftarrow T' \cup \{t\}$
        $R \leftarrow R \setminus \text{TR}(t)$
    return T'

Greedy Minimization

Algorithm Greedy Minimization

```
function GreedyMinimization(T, R)
    T' ← ∅
    while true do
        t ← argmax_{t∈T} |R ∩ TR(t)|
        if |R ∩ TR(t)| = 0 then
            break
        T' ← T' ∪ {t}
        R ← R \ TR(t)
    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

function GreedyMinimization(T, R)
  $T' \leftarrow \emptyset$
  while `true` **do**
    $t \leftarrow \text{argmax}_{t \in T} |R \cap \text{TR}(t)|$
    if $|R \cap \text{TR}(t)| = 0$ **then**
      break
    $T' \leftarrow T' \cup \{t\}$
    $R \leftarrow R \setminus \text{TR}(t)$
  return T'

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

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

Greedy Minimization

PLRG

Algorithm Greedy Minimization

function GreedyMinimization(T, R)
    $T' \leftarrow \emptyset$
    while `true` **do**
        $t \leftarrow \text{argmax}_{t \in T} |R \cap \text{TR}(t)|$
        if $|R \cap \text{TR}(t)| = 0$ **then**
            break
        $T' \leftarrow T' \cup \{t\}$
        $R \leftarrow R \setminus \text{TR}(t)$
    return T'

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

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

$T' = \{t_3\}$
$R = \{r_3\}$

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

Greedy Minimization with Cost

- 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 it still $\{t_3, t_1\}$ 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

```
function GreedyMinimization(T, R)
    T' ← ∅
    while true do
        t ← argmax_{t∈T} |R∩TR(t)| / Time(t)
        if |R ∩ TR(t)| = 0 then
            break
        T' ← T' ∪ {t}
        R ← R \ TR(t)
    return T'
```

Greedy Minimization with Cost

PLRG

Algorithm Greedy Minimization with Cost

function $\textsc{GreedyMinimization}(T, R)$
  $T' \leftarrow \emptyset$
  while `true` **do**
    $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
    if $|R \cap \text{TR}(t)| = 0$ **then**
      break
    $T' \leftarrow T' \cup \{t\}$
    $R \leftarrow R \setminus \text{TR}(t)$
  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

$T' = \varnothing$

$R = \{r_1, r_2, r_3, r_4\}$

Algorithm Greedy Minimization with Cost

function GREEDYMINIMIZATION(T, R)
    $T' \leftarrow \emptyset$
    while `true` **do**
        $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
        if $|R \cap \text{TR}(t)| = 0$ **then**
            break
        $T' \leftarrow T' \cup \{t\}$
        $R \leftarrow R \setminus \text{TR}(t)$
    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

```
function GreedyMinimization(T, R)
    T' ← ∅
    while true do
        t ← argmax_{t∈T} |R∩TR(t)| / Time(t)
        if |R ∩ TR(t)| = 0 then
            break
        T' ← T' ∪ {t}
        R ← R \ TR(t)
    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

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

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

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

Greedy Minimization with Cost

Algorithm Greedy Minimization with Cost

function GREEDYMINIMIZATION(T, R)
  $T' \leftarrow \emptyset$
  while `true` **do**
    $t \leftarrow \text{argmax}_{t \in T} \frac{|R \cap \text{TR}(t)|}{\text{Time}(t)}$
    if $|R \cap \text{TR}(t)| = 0$ **then**
      break
    $T' \leftarrow T' \cup \{t\}$
    $R \leftarrow R \setminus \text{TR}(t)$
  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

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

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

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

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

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 **conformance test suite** for a JavaScript engines, and each test case is a JavaScript program with assertions.

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.
- For example, consider **conformance test suite** for a JavaScript engines, and each test case is a JavaScript program with assertions.
- Then, which test case is better to keep?

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.
- For example, consider **conformance test suite** for a JavaScript engines, and 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

```
function GreedyMinimization(T, R)
    M ← ∅
    for t ∈ T do
        for r ∈ TR(t) do
            if r ∉ M ∨ Score(t) > Score(M[r]) then
                M[r] ← t
    T′ ← {t | (r, t) ∈ M}
    return T′
```

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

function GreedyMinimization(T, R)
    $M \leftarrow \emptyset$
    for $t \in T$ **do**
        for $r \in \text{TR}(t)$ **do**
            if $r \notin M \vee \text{Score}(t) > \text{Score}(M[r])$ **then**
                $M[r] \leftarrow t$
    $T' \leftarrow \{t \mid (r, t) \in M\}$
    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

$M = \{$

Greedy Minimization with Score

Algorithm Greedy Minimization with Score

function GreedyMinimization(T, R)
  $M \leftarrow \emptyset$
  for $t \in T$ **do**
    for $r \in \text{TR}(t)$ **do**
      if $r \notin M \lor \text{Score}(t) > \text{Score}(M[r])$ **then**
        $M[r] \leftarrow t$
  $T' \leftarrow \{t \mid (r, t) \in M\}$
  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

function $\textsc{GreedyMinimization}(T, R)$
  $M \leftarrow \emptyset$
  for $t \in T$ **do**
    for $r \in \mathrm{TR}(t)$ **do**
      if $r \notin M \lor \mathrm{Score}(t) > \mathrm{Score}(M[r])$ **then**
        $M[r] \leftarrow t$
  $T' \leftarrow \{t \mid (r, t) \in M\}$
  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

PLRG

Algorithm Greedy Minimization with Score

```
function GreedyMinimization(T, R)
    M ← ∅
    for t ∈ T do
        for r ∈ TR(t) do
            if r ∉ M ∨ Score(t) > Score(M[r]) then
                M[r] ← t
    T′ ← {t | (r, t) ∈ M}
    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

```
function GreedyMinimization(T, R)
    M ← ∅
    for t ∈ T do
        for r ∈ TR(t) do
            if r ∉ M ∨ Score(t) > Score(M[r]) then
                M[r] ← t
    T′ ← {t | (r, t) ∈ M}
    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

function GreedyMinimization(T, R)
  $M \leftarrow \emptyset$
  for $t \in T$ **do**
    for $r \in \mathsf{TR}(t)$ **do**
      if $r \notin M \vee \mathsf{Score}(t) > \mathsf{Score}(M[r])$ **then**
        $M[r] \leftarrow t$
  $T' \leftarrow \{t \mid (r, t) \in M\}$
  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

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

Greedy Minimization with Score

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

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**.

[ICSE'21] J. Park et al., "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**.

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**.

[ICSE'21] J. Park et al., "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

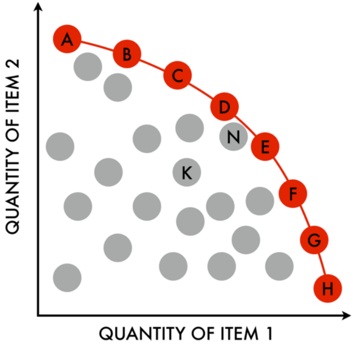
What if the test suite minimization problem has **multiple objectives**?

- "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 consider the **fault detection capability** of the test suite."

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

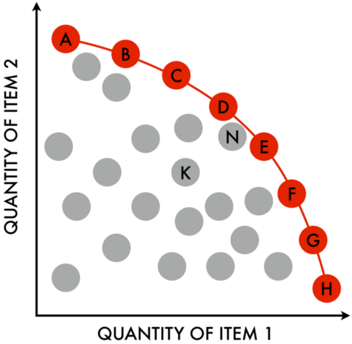
Multi-Objective Problem

- Let's consider the **Pareto efficiency** of the test suite minimization.



Multi-Objective Problem

- Let's consider the **Pareto efficiency** of the test suite minimization.

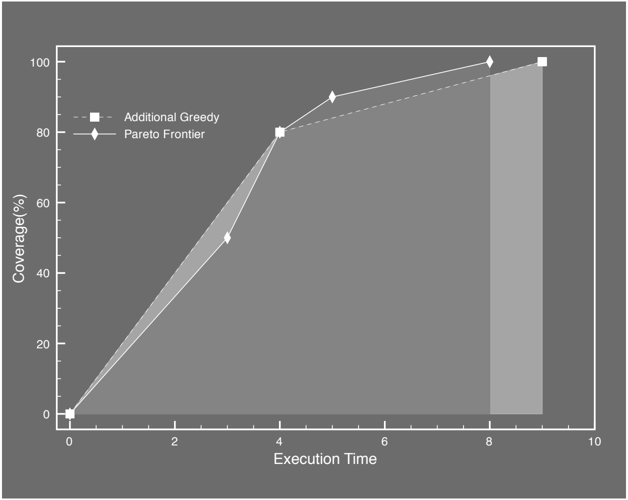


- And, we can consider the **weighted sum** of the objectives.

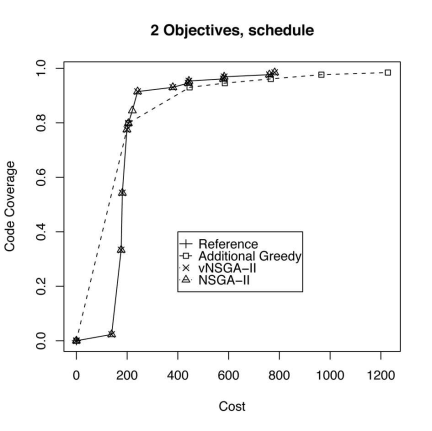
$$o' = \alpha_1 \times o_1 + \alpha_2 \times o_2 + \cdots \alpha_n \times o_n$$

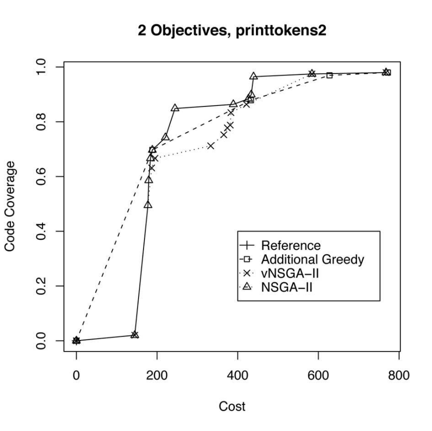
where α_i is the weight of the i-th objective such that $\sum_{1 \leq i \leq n} \alpha_i = 1$, and o_i is the value of the i-th objective.

Multi-Objective Problem

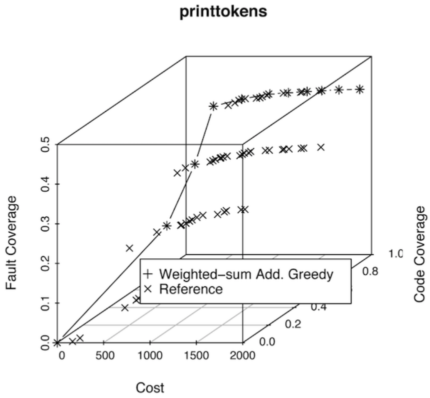


Multi-Objective Problem





Multi-Objective Problem



Test Case Selection

- **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

- **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

- **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

- 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 the **which parts** of the program are **changed** in the updated program P' compared to the original program P.

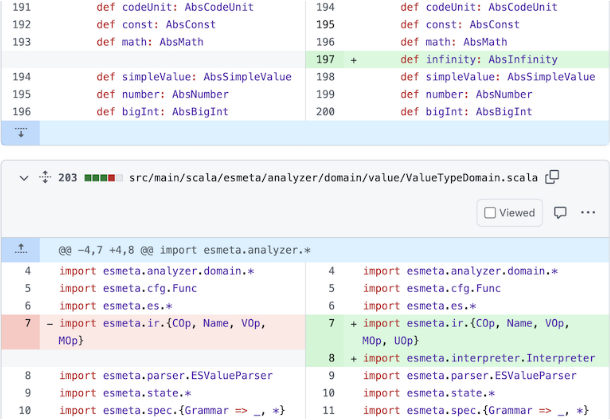
Test Case Selection – Textual Diff

- Most of the modern software are developed using the **version control system** (e.g., Git, SVN, etc.).

Test Case Selection – Textual Diff

PLRG

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



Test Case Selection – Control Flow Graph

However, the **textual diff** is not enough to precisely know the **changed parts** of the program.

Test Case Selection – Control Flow Graph

However, the **textual diff** is not enough to precisely know the **changed parts** of the program.

P

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

P'

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

Test Suite

- $t_1 : x = 0$
- $t_2 : x = 1$

Test Case Selection – Control Flow Graph

However, the **textual diff** is not enough to precisely know the **changed parts** of the program.

P

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

P'

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

Test Suite

- $t_1 : x = 0$
- $t_2 : x = 1$

Which test case should be **selected**?

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?

- It is due to that they focus on **safety** (i.e., not missing any tests that are related to the recent modification to avoid regression faults)

Test Case Selection – Safe Selection

- Selection techniques do **not consider** the **cost** of tests. Why?
- It is due to that they focus on **safety** (i.e., not missing any tests that are related to the recent modification to avoid regression faults)
- 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 safe selection is still too **expensive** to run all the selected test cases?

Test Case Prioritization

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

Test Case Prioritization

- **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

- **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)

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

Average Percentage of Faults Detected (APFD)

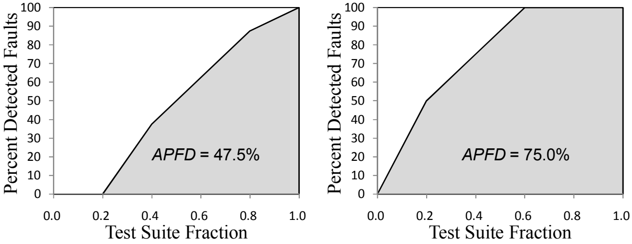
- We can measure the effectiveness of the test case prioritization based on the **average percentage of faults detected** (APFD).
- Intuitively, APFD evaluates the effectiveness of the test case based on the **area** under the curve of the **fault detection rate**.

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).
- 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			•				•	



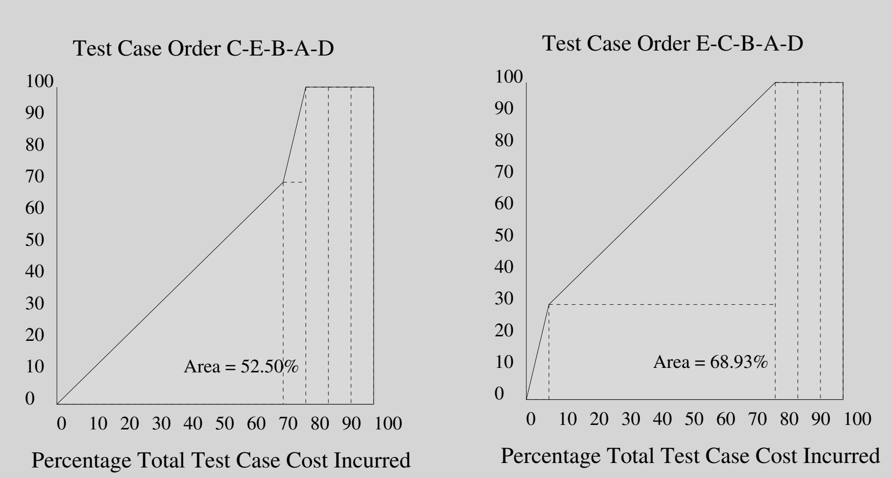
(a) *APFD* for test suite T_1 (b) *APFD* for test suite T_2

Average Percentage of Faults Detected (APFD)

- APFDc (by Elbaum et al., 2001) is a variant of APFD that considers the **cost** of the test cases.

Average Percentage of Faults Detected (APFD)

- APFDc (by Elbaum et al., 2001) is a variant of APFD that considers the **cost** of the test cases.



Cost-Cognisant Test Case Prioritisation, ICSE, Malishevsky et al., 2006

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

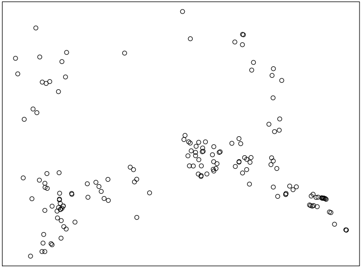
- A technique that **groups** objects such that objects in the same group are the most similar to each other.
- Reduces the conceptual size of test suites by **clustering** test cases that are similar to each other.

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

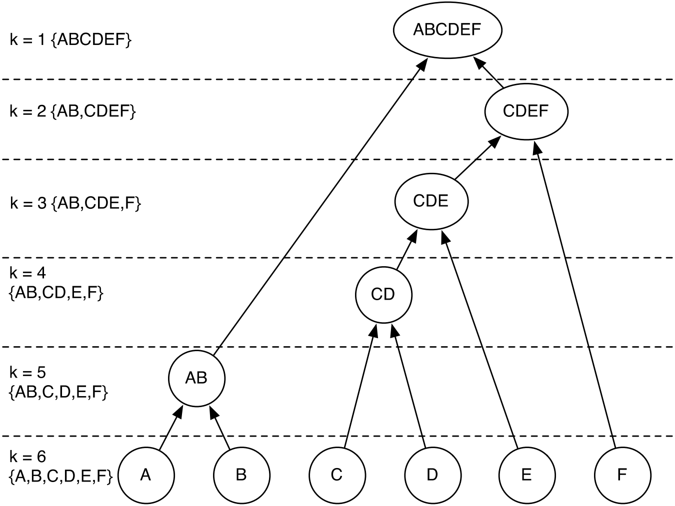
[ISSTA'09] Yoo et al., "Clustering Test Cases to Achieve Effective & Scalable Prioritisation Incorporating Expert Knowledge"

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

```
Form n clusters, each with one test case
C ← {}
Add clusters to C
Insert n clusters as leaf node into D
while there is more than one cluster
    Find a pair of clusters with minimum distance
    Merge the pair into a new cluster, c_new
    Remove the pair of test cases from C
    Add c_new to C
    Insert c_new as a parent node of the pair into D
return D
```

Test Case Prioritization – Clustering

[ISSTA'09] Yoo et al., "Clustering Test Cases to Achieve Effective & Scalable Prioritisation Incorporating Expert Knowledge"



Test Case Prioritization – Clustering

[ISSTA'09] Yoo et al., "Clustering Test Cases to Achieve Effective & Scalable Prioritisation Incorporating Expert Knowledge"

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

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

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

Test Case Prioritization – Similarity

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

Test Case Prioritization – Similarity

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

[ICSE'19] Cruciani et al., "Scalable Approaches for Test Suite Reduction"

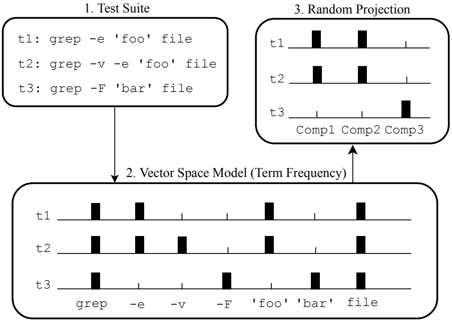


Fig. 1: Visual representation of *FAST-R* preparation phase.

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

Test Case Prioritization – Similarity

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

[ICSE'19] Cruciani et al., "Scalable Approaches for Test Suite Reduction"

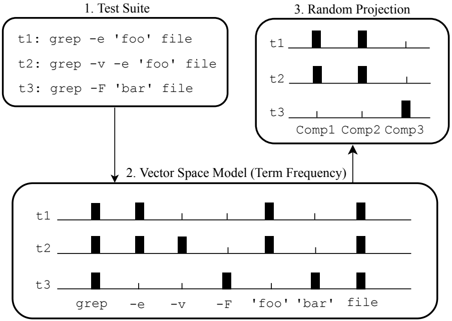
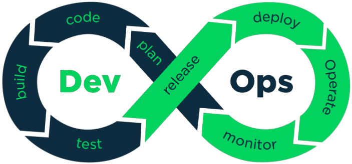


Fig. 1: Visual representation of *FAST-R* preparation phase.

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?

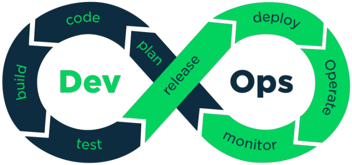
Regression Testing in Practice



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

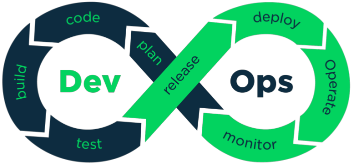
Regression Testing in Practice

PLRG



- 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**.

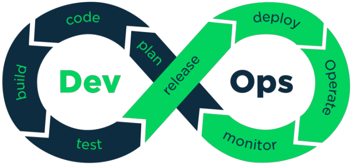
Regression Testing in Practice



- 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

PLRG



- 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

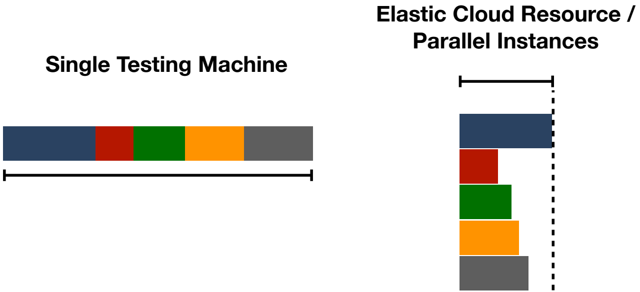
- All regression testing techniques are limited by the **cost** of running the test cases.

Regression Testing in Practice

- 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**.

Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon engineers conduct 136,000 system deployments per day: one in every 12 seconds.

Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon engineers conduct 136,000 system deployments per day: one in every 12 seconds.
  - Google engineers have to wait up to 9 hours to receive test results from the CI pipeline.

Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
  - Amazon engineers conduct 136,000 system deployments per day: one in every 12 seconds.
  - Google engineers have to wait up to 9 hours to receive test results from the CI pipeline.
- **Prioritising test cases** within test suites makes little impact at this scale.

Redefining Prioritization

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded with commits**.
    - Amazon engineers conduct 136,000 system deployments per day: one in every 12 seconds.
    - Google engineers have to wait up to 9 hours to receive test results from the CI pipeline.
- **Prioritising test cases** within test suites makes little impact at this scale.
- Instead, **prioritising commits** to test has been proposed.

Redefining Prioritization

- The proposed technique is essentially history based prioritisation: **commits** relevant to test cases that have **recently failed** are given higher priority. If tests really fail, this ensures **quicker feedback** to the responsible developers.

Redefining Prioritization

- The proposed technique is essentially history based prioritisation: **commits** relevant to test cases that have **recently failed** are given higher priority. 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

- Fault Localization

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