Lecture 10 – Fault Localization

AAA705: Software Testing and Quality Assurance

Jihyeok Park

2024 Spring

Recall

- **Regression Testing**
  - Regression Fault
  - Test Suite Minimization
  - Test Case Selection
  - Test Case Prioritization
  - Regression Testing in Practice

Fault Localization

- We found a bug by running a test suite!
- Before fixing the bug, we need to know **where it is**.
- **Fault Localization** is the process of identifying the **location** of a fault in the code.

Fault Localization

There are several basic **manual approaches** to fault localization:

- **Logging** – Inserting print statements to trace the program execution.
- **Assertions** – Inserting assertions to check the program state.
- **Breakpoints** – Using a debugger to pause the program execution and inspect the state.
- **Profiling** – Profiling the execution speed and memory usage of the program typically to find performance and memory usage bugs.

Contents

Contents

Motivation

PLRG

Following HTML code caused Firefox to crash. What is the actual cause?

```
<td align=left valign=top>
<SELECT NAME="op sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All <OPTION VALUE="Windows 3.1">Windows 3.1
<OPTION VALUE="Windows 95">Windows 95 <OPTION VALUE="Windows 98">Windows 98
<OPTION VALUE="Windows ME">Windows ME <OPTION VALUE="Windows 2000">Windows 2000
<OPTION VALUE="Windows NT">Windows NT <OPTION VALUE="Mac System 7">Mac System 7
<OPTION VALUE="Mac System 7.5">Mac System 7.5 <OPTION VALUE="Mac System 7.6.1">Mac System 7.6.1
<OPTION VALUE="Mac System 8.0">Mac System 8.0 <OPTION VALUE="Mac System 8.5">Mac System 8.5
<OPTION VALUE="Mac System 8.6">Mac System 8.6 <OPTION VALUE="Mac System 9.x">Mac System 9.x
<OPTION VALUE="MacOS X">MacOS X <OPTION VALUE="Linux">Linux
<OPTION VALUE="BSDI">BSDI <OPTION VALUE="FreeBSD">FreeBSD
<OPTION VALUE="NetBSD">NetBSD <OPTION VALUE="OpenBSD">OpenBSD
<OPTION VALUE="AIX">AIX <OPTION VALUE="BeOS">BeOS
<OPTION VALUE="HP-UX">HP-UX <OPTION VALUE="IRIX">IRIX
<OPTION VALUE="Neutrino">Neutrino <OPTION VALUE="OpenVMS">OpenVMS
<OPTION VALUE="OS/2">OS/2 <OPTION VALUE="OSF/1">OSF/1
<OPTION VALUE="Solaris">Solaris <OPTION VALUE="SunOS">SunOS
<OPTION VALUE="other">other</SELECT></td> <td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">-- <OPTION VALUE="P1">P1
<OPTION VALUE="P2">P2 <OPTION VALUE="P3">P3
<OPTION VALUE="P4">P4 <OPTION VALUE="P5">P5</SELECT> </td> <td align=left valign=top>
<SELECT NAME="bug severity" MULTIPLE SIZE=7> <OPTION VALUE="blocker">blocker
<OPTION VALUE="critical">critical <OPTION VALUE="major">major <OPTION VALUE="normal">normal
<OPTION VALUE="minor">minor <OPTION VALUE="trivial">trivial
<OPTION VALUE="enhancement">enhancement </ SELECT>
</tr> </table>
```

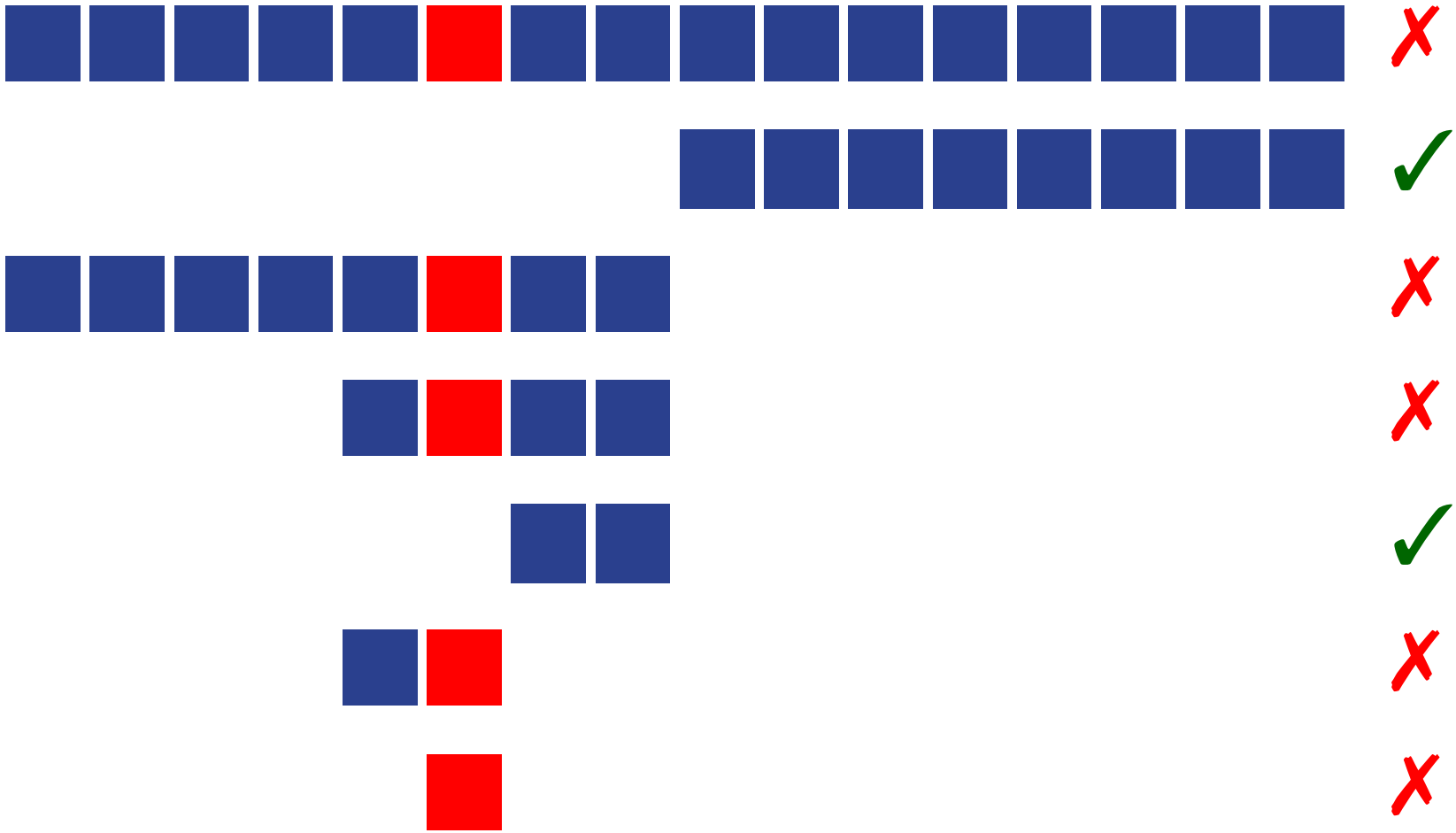
Delta Debugging (DD)

PLRG

- **Not all parts** of the test case are necessary to cause the failure.
- The core idea of **delta debugging (DD)** is to **simplify** the test case that causes the failure to **keep only the necessary parts** by **removing unnecessary parts**.
- The simplified test case has the following benefits:
  - **Ease of communication** – A simplified test case is easier to communicate.
  - **Easier debugging** – A smaller test case result in smaller states and shorter executions.
  - **Identify duplicates** – Simplified test cases subsume several duplicates.

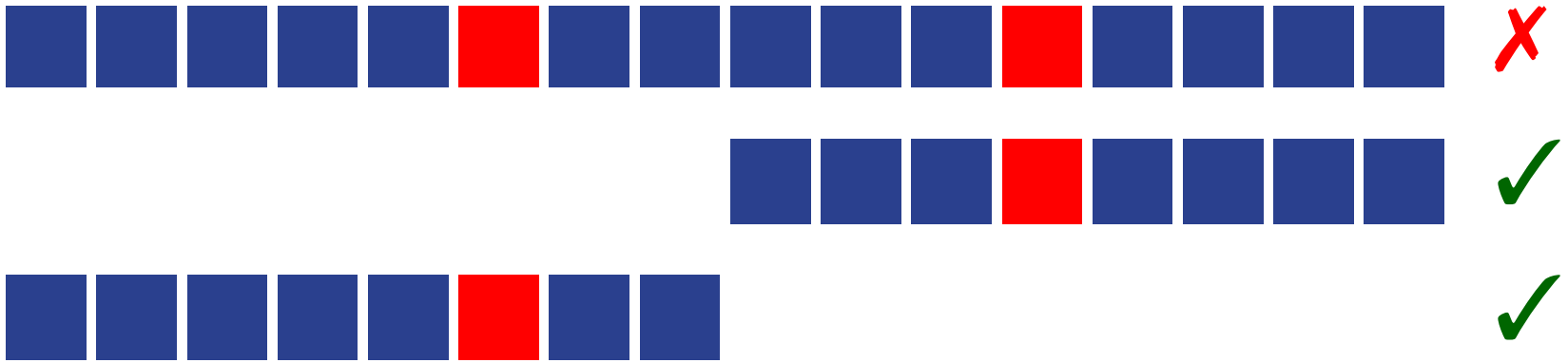
Delta Debugging

Basic approach of delta debugging is just **binary search**:



Delta Debugging

However, what if the failure is caused by a combination of multiple parts?



Recursive Delta Debugging – ddmin

PLRG

Recursively split the test into smaller parts and perform DD on each part.

Algorithm Recursive Delta Debugging – ddmin

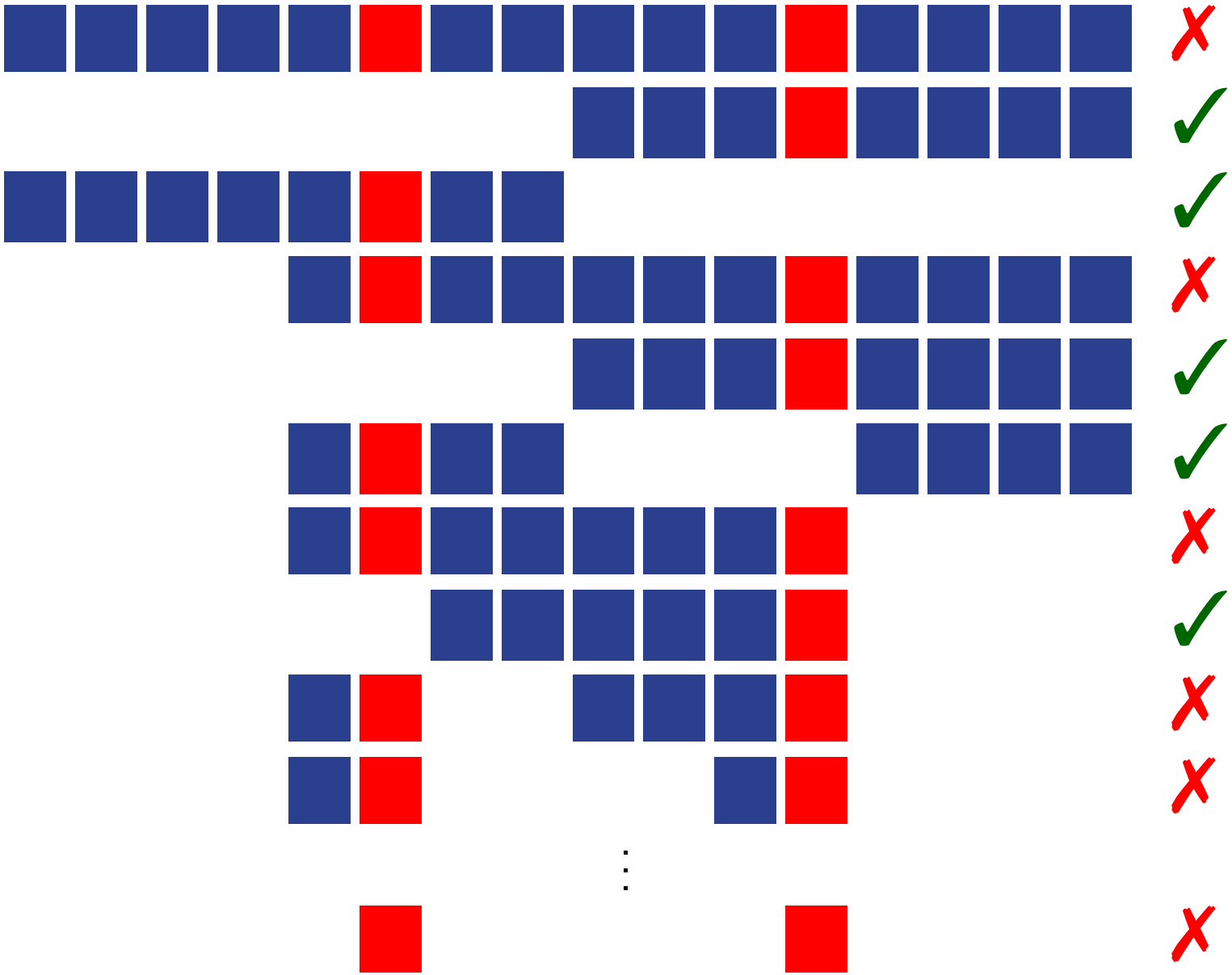
Input: A test case t that causes a failure

Output: A minimal test case that still causes the failure t'

```
1:  function DDMIN(t)
2:      L ← the list of elements in t
3:      n ← 2
4:      while n ≥ |L| do
5:          ⟨c_1, ..., c_n⟩ ← split L into n chunks
6:          if ∃i ∈ [1, n]. test(L \ c_i) = ✗ then
7:              ⟨L, n⟩ ← ⟨L \ c_i, n − 1⟩
8:          else if Possible to split c_i into two chunks c_i' and c_i'' then
9:              ⟨L, n⟩ ← ⟨⟨c_1', c_1'', ..., c_n', c_n''⟩, 2n⟩
10:         else
11:             break
12:     return a test case t' that corresponds to L
```

A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering, 28(2):183–200, Feb. 2002

Recursive Delta Debugging – ddmin



Hierarchical Delta Debugging (HDD)

PLRG

- What if the test case is **highly structured**?
- For example, if our target software is a JavaScript **interpreter**, test cases are JavaScript **programs** following the JavaScript **grammar**.
- If we delete a part of the program regardless of the grammar, the resulting program **most likely** will be an **invalid** program.
- **Hierarchical Delta Debugging (HDD)** is a variant of DD that takes the structure of the test case into account. It recursively goes into deeper nested structures instead of just splitting the test case into chunks.
    - G. Misherghi and Z. Su. HDD: Hierarchical delta debugging. In Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pages 142–151, New York, NY, USA, 2006. ACM.

Hierarchical Delta Debugging (HDD)

First, we can consider the **function-level** structure of the test case.

```
int copy(double to[], double from[], int count)
{
  int n= (count+7)/8;
  switch (count%8) do {
    case 0: *to++ = *from++;
    case 7: *to++ = *from++;
    case 6: *to++ = *from++;
    case 5: *to++ = *from++;
    case 4: *to++ = *from++;
    case 3: *to++ = *from++;
    case 2: *to++ = *from++;
    case 1: *to++ = *from++;
  } while (--n > 0);
  return (int)mult(to,2);
}
```

```
double mult( double z[], int n )
{
  int i;
  int j;
  for (j= 0; j< n; j++) {
    i= i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
  return z[n];
}
int main( int argc, char *argv[] )
{
  double x[20], y[20];
  double *px= x;
  while (px < x + 20)
    *px++ = (px-x)*(20+1.0);
  return copy(y,x,20);
}
```

Hierarchical Delta Debugging (HDD)

PLRG

Then, let's consider the **statement-level** structure of the test case.

```
double mult( double z[], int n )
{
  int i;
  int j;
  for (j= 0; j< n; j++) {
    i= i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
  return z[n];
}
```

Hierarchical Delta Debugging (HDD)

PLRG

Then, let's consider the **expression-level** structure of the test case.

```
double mult( double z[], int n )
{
  int i;
  int j;
  for (j= 0; j< n; j++) {
    i= i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

Hierarchical Delta Debugging (HDD)

PLRG

The final result of HDD is as follows:

```
double mult( double z[], int n )
{
  int i;
  int j;
  for (;;) {
    i= i+j+1;
    z[i]=z[i]*(z[0]+0);
  }
}
```

Probabilistic Delta Debugging (ProbDD)

PLRG

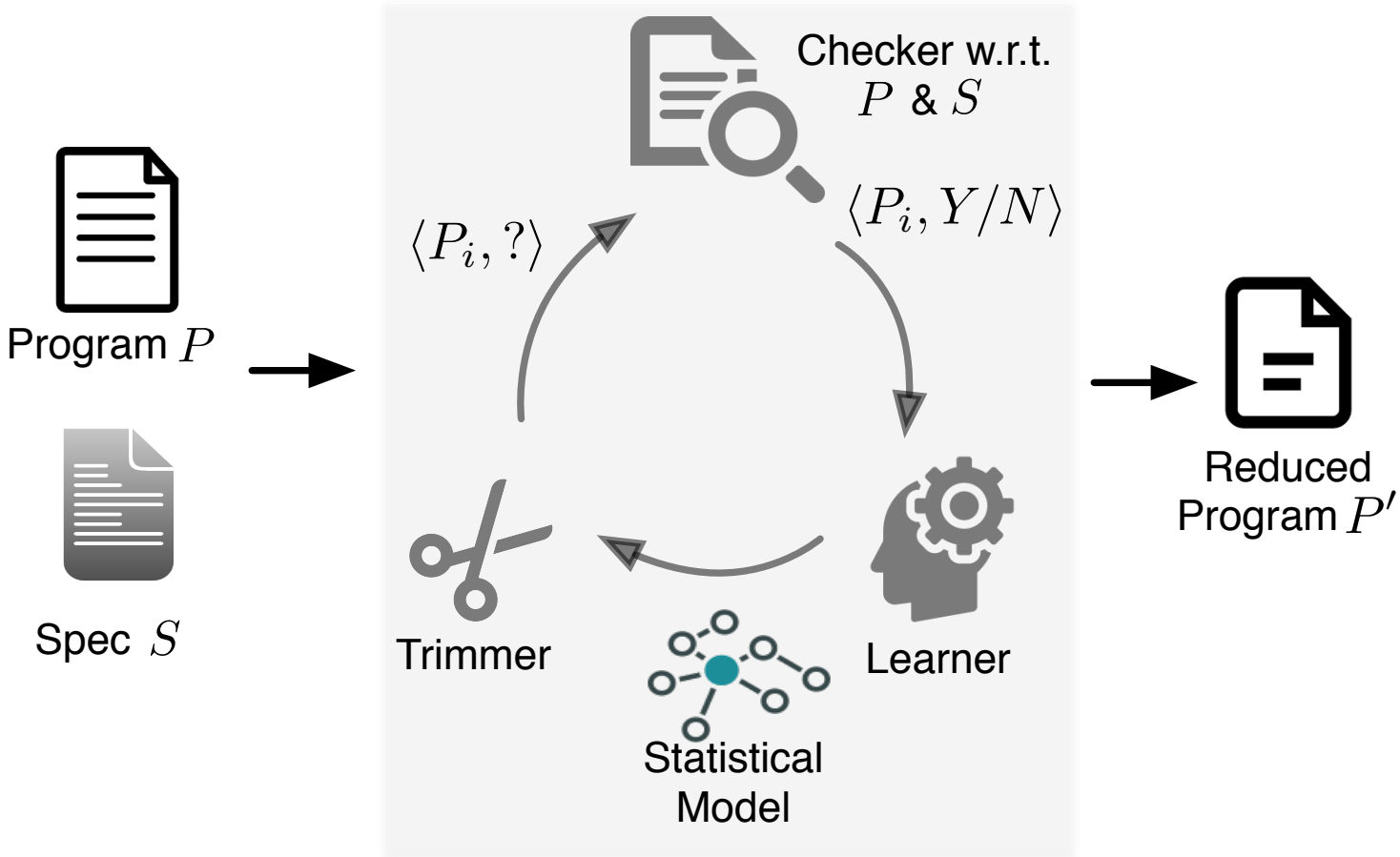
- The fixed way to split the test case may be **not the best way** to find the minimal test case.
- Then, let's consider a **probabilistic** way to split the test case.
- **Probabilistic Delta Debugging (ProbDD)** is a variant of DD that **randomly selects** the sequential parts to select the next part and **updates** the **probabilistic model** based on the previous results.
    - G. Wang, R. Shen, J. Chen, Y Xiong, and L Zhang. Probabilistic delta debugging. In Proceedings of the 2021 Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021.

Probabilistic Delta Debugging (ProbDD)

	s1	s2	s3	s4	s5	s6	s7	s8	
	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	0.2500	
1	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.3657	0.3657	0.3657	0.2500	0.2500	0.2500	0.2500	0.3657	
2	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.3657	0.3657	0.3657	0	0	0	0	0.3657	
3	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.3657	0.3657	0	0	0	0	0.6119	
4	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0.6119	0.6119	0	0	0	0	0.6119	
5	s1	s2	s3	s4	s5	s6	s7	s8	T
	0.6119	0	0.6119	0	0	0	0	0.6119	
6	s1	s2	s3	s4	s5	s6	s7	s8	F
	0.6119	0	1	0	0	0	0	0.6119	
7	s1	s2	s3	s4	s5	s6	s7	s8	T
	0	0	1	0	0	0	0	0.6119	
8	s1	s2	s3	s4	s5	s6	s7	s8	F
	0	0	1	0	0	0	0	1	

Delta Debugging for Program Debloating

- We can utilize DD technique to **debloat** a program.
  - K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective Program Debloating via Reinforcement Learning. In proceedings of the 2018 Conference on Computer and Communications Security, CCS 2018.



Delta Debugging for Program Debloating

PLRG

Let's consider the following program:

```
int f1() { return 0; }

int f2() { return 1; }

int f3() { return 1; }

int f4() { return 1; }

int f5() { return 1; }

int f6() { return 1; }

int f7() { return 1; }

int main() { return f1(); }
```

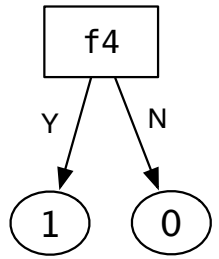
Delta Debugging for Program Debloating

PLRG

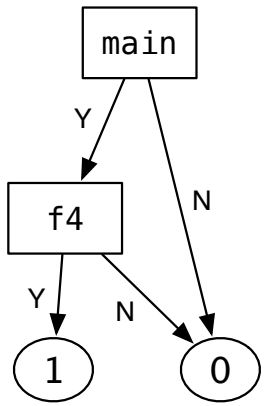
	f1	f2	f3	f4	f5	f6	f7	main	✔
1	f1	f2	f3	f4	f5	f6	f7	main	✘
2	f1	f2	f3	f4	f5	f6	f7	main	✘
3	f1	f2	f3	f4	f5	f6	f7	main	✘
4	f1	f2	f3	f4	f5	f6	f7	main	✘
5	f1	f2	f3	f4	f5	f6	f7	main	✘
6	f1	f2	f3	f4	f5	f6	f7	main	✘
7	f1	f2	f3	f4	f5	f6	f7	main	✘
8	f1	f2	f3	f4	f5	f6	f7	main	✔
9	f1	f2	f3	f4	f5	f6	f7	main	✔
10	f1	f2	f3	f4	f5	f6	f7	main	✘
11	f1	f2	f3	f4	f5	f6	f7	main	✘
12	f1	f2	f3	f4	f5	f6	f7	main	✘
13	f1	f2	f3	f4	f5	f6	f7	main	✘
14	f1	f2	f3	f4	f5	f6	f7	main	✘
15	f1	f2	f3	f4	f5	f6	f7	main	✔
16	f1	f2	f3	f4	f5	f6	f7	main	✔

Delta Debugging for Program Debloating

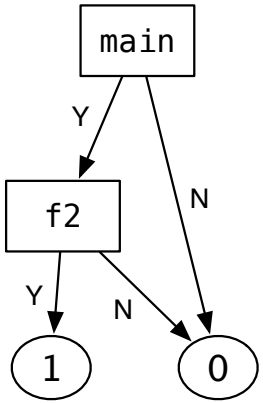
	f1	f2	f3	f4	f5	f6	f7	main	✔
1	f1	f2	f3	f4	f5	f6	f7	main	✘
2	f1	f2	f3	f4	f5	f6	f7	main	✘
3	f1	f2	f3	f4	f5	f6	f7	main	✘
4	f1	f2	f3	f4	f5	f6	f7	main	✔
5	f1	f2	f3	f4	f5	f6	f7	main	✔
6	f1	f2	f3	f4	f5	f6	f7	main	✘
7	f1	f2	f3	f4	f5	f6	f7	main	✔
8	f1	f2	f3	f4	f5	f6	f7	main	✔
9	f1	f2	f3	f4	f5	f6	f7	main	✘
10	f1	f2	f3	f4	f5	f6	f7	main	✘



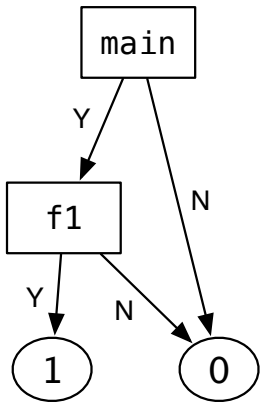
Iteration 1



Iteration 2



Iteration 3



Iteration 6

Delta Debugging in Practice

PLRG

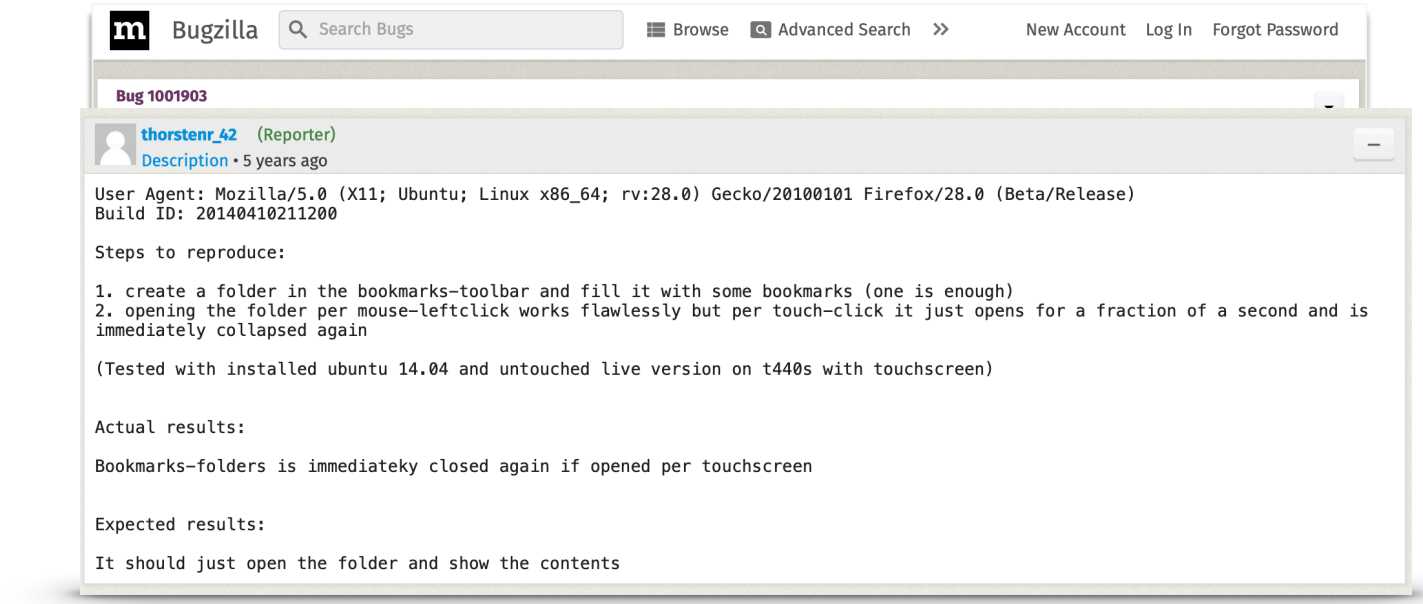
- DD.py – a Python implementation of Delta Debugging
- Lithium – a Python implementation of a Hierarchical Delta Debugging
- C-Reduce – a tool that reduces source files written in C/C++ using the Delta algorithms

Contents

Motivation

Can we utilize **information** in the **bug report** for **fault localization**?

For example, this is a real bug report for Firefox:



https://bugzilla.mozilla.org/show_bug.cgi?id=1001903

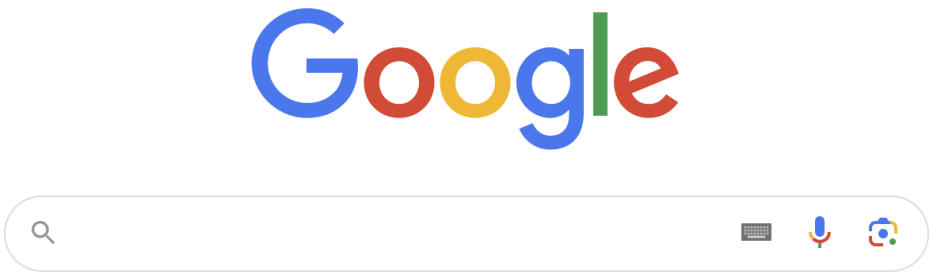
Motivation

- After **creating** a **folder** in the **bookmark toolbar** and a **bookmark** inside the **folder**, mouse left-click successfully opens the folder but **touch-click** fails to **open** it: it **opens** but immediately **closes** itself.

- While we do not know the internal structure of the Firefox source code, we **can guess** which parts of the code are **most likely to be related** to the fault.

Information Retrieval (IR) based Fault Localization

- **Information Retrieval (IR)** techniques help a user to quickly obtain resources relevant to an information need, from a large collection of information resources.



- **Fault localization** can be thought of as finding a resource (a program element) that is relevant to an information need (the reported symptoms of the failure), from a large collection of information resources (the entire system).
  - The **bug report** becomes our **query**.
  - The entire **source code** becomes our **collection of documents**.
  - **Fault localization** is to **find** the source code that matches the bug report the best.

Vector Space Model (VSM)

- There are many ways to represent queries and documents in IR.
- **Vector Space Model (VSM)** is one of the most popular ways to represent queries and documents as vectors in a high-dimensional space.
- It allows us to calculate the **similarity** between the query and the documents easily.

Vector Space Model (VSM)

PLRG

- First, define the **vocabulary**, a set of meaningful terms of the system.
- Given a vocabulary with N terms, we represent both the query and the documents as vectors:

$$\begin{aligned} d_j &= (w_{1,j}, w_{2,j}, \ldots, w_{N,j}) && \text{for document } j \\ q &= (w_{1,q}, w_{2,q}, \ldots, w_{N,q}) && \text{for query} \end{aligned}$$

- The **dimensionality** of these vectors is equal to the number of terms in the vocabulary.
- If the term w_2 **appears** in the document d_j and the query q, then $w_{2,j}$ and $w_{2,q}$ are **non-zero**, otherwise, they are **zero**.
- There are many ways to set the non-zero values. Let's study **tf-idf**.

Tf-Idf

- **Term-Frequency Inverse Document Frequency (Tf-Idf)** is a numerical statistic that reflects how important a term is to a document in a collection or corpus.
- **Term Frequency (tf)**, $tf(t, d)$, is the number of times the term t appears in the document d divided by the total number of terms in the document d.

$$tf(t, d) = \frac{f_{t,d}}{|d|}$$

  where $f_{t,d}$ is the number of times t appears in d
- **Inverse Document Frequency (idf)**, $idf(t, D)$, is the logarithm of the ratio of the total number of documents in the corpus D to the number of documents containing the term t.

$$idf(t, D) = \log\left(\frac{|D|}{|\{d \in D \mid t \in d\}|}\right)$$

Tf-Idf

- **Term-Frequency Inverse Document Frequency (Tf-Idf)** is just a product of the **term frequency** and the **inverse document frequency**.

$$\text{Tf-Idf}(t, d, D) = tf(t, d) \times idf(t, D)$$

- A **higher** Tf-Idf(t, d, D) value means that the term t occupies a **higher proportion** in the document d and is **less common** in the corpus D.

- A **lower** Tf-Idf(t, d, D) value means that the term t occupies a **lower proportion** in the document d and is **more common** in the corpus D.

Cleansing Bug Reports

PLRG

- We follow the standard text **cleansing process** used in many IR and machine learning applications.
  - **Tokenization** – Break down the bug report to a list of tokens.
  - **Remove punctuation** – For example, “file’s” becomes “file”.
  - **Case normalization** – For example, “File” becomes “file”.
  - **Stop word filtering** – some words are extremely common in English (e.g., “a”, “an”, “the”) and best to be removed. Such words are called **stop words**.
  - **Stemming** – For example, “running” becomes “run”.
- There are many widely used libraries that will perform these tasks.

Cleansing Source Code

PLRG

- Similarly, we need to do the cleansing process for the source code.
- In source code, **reserved keywords** can be considered as **stop words**.
- In addition, we need to apply additional normalization for identifiers:
  - **CamelCase** normalization – For example, "getFileName" becomes "get", "file", and "name".
  - **SnakeCase** normalization – For example, "get_file_name" becomes "get", "file", and "name".

VSM and Similarity

PLRG

- After cleansing, using a predefined vocabulary, we can calculate the Tf-Idf values for each term t (word) and each document d_j (source code file) or the query q (bug report).
- Then, we can represent the bug report and the source code as **vectors** in a high-dimensional space.
- The **similarity** between the bug report and the source code can be calculated using the **cosine similarity**:

$$\cos(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{N} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{N} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{N} w_{i,q}^2}}$$

- Finally, we can select the source code file that has the **highest cosine similarity** to the bug report as the **fault location**.

Strengths and Weaknesses

- **Strengths**
  - **Simple** and **easy** to implement.
  - **Intuitive** and **easy to understand**.
- **Weaknesses**
  - Requires a **high-quality** and **detailed** bug report.
  - Suffers from the **inherent limit to the accuracy** because the document to be matched (i.e., the unit of localization) needs to be of certain size (otherwise lexical similarity becomes more random).

Contents

Motivation

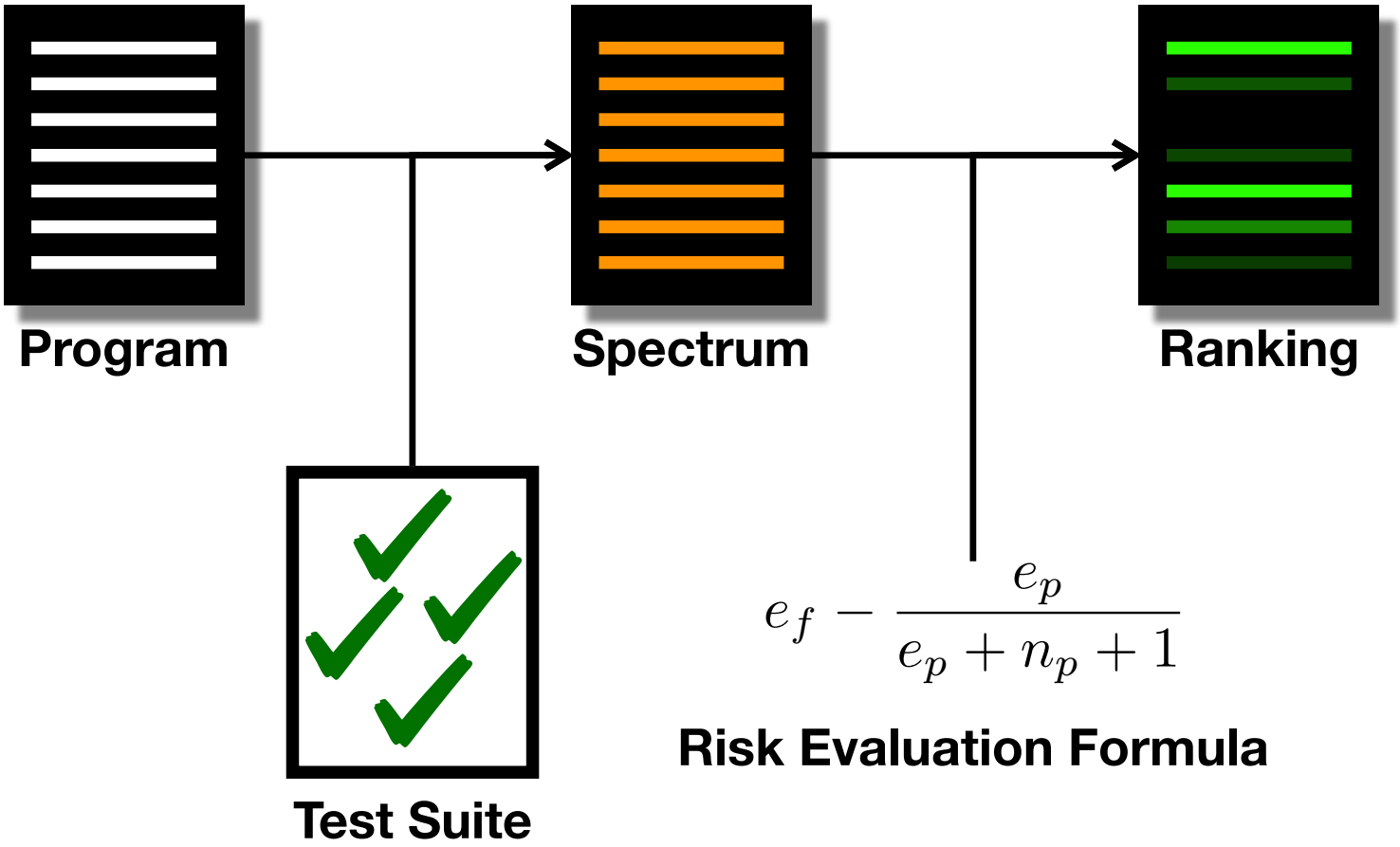
PLRG

- **Intuition** – If a program has a **faulty line**, a test case that **executes** the faulty line is more likely to **fail** than a test case that does not execute the faulty line.

- It means that we want to utilize **statistical information** about the program's execution to **localize** the fault.

- We need to reflect this intuition to a **computable form** to automate the fault localization process.

Spectrum-based Fault Localization (SBFL)



Spectrum-based Fault Localization (SBFL)

PLRG

- **Program Spectrum** – for each structural unit (e.g., statement, branch, predicate), summarize the test result and coverage into a tuple of the following four numbers:
  - e_p – # test cases that **execute** the unit e and **pass**.
  - e_f – # test cases that **execute** the unit e and **fail**.
  - n_p – # test cases that do **not execute** the unit e and **pass**.
  - n_f – # test cases that do **not execute** the unit e and **fail**.

Tarantula (2001)

The following is the initial formula for SBFL called **Tarantula**:

$$Tarantula(e) = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f} + \frac{e_p}{e_p+n_p}}$$

The following part represents the how many test cases are **associated** with the unit e among the **failing** test cases:

$$\frac{e_f}{e_f + n_f}$$

On the other hand, the following part represents the how many test cases are **associated** with the unit e among the **passing** test cases:

$$\frac{e_p}{e_p + n_p}$$

Tarantula (2001)

While it originally was meant as a **visualization** technique, it has been widely used as a fault localization technique.

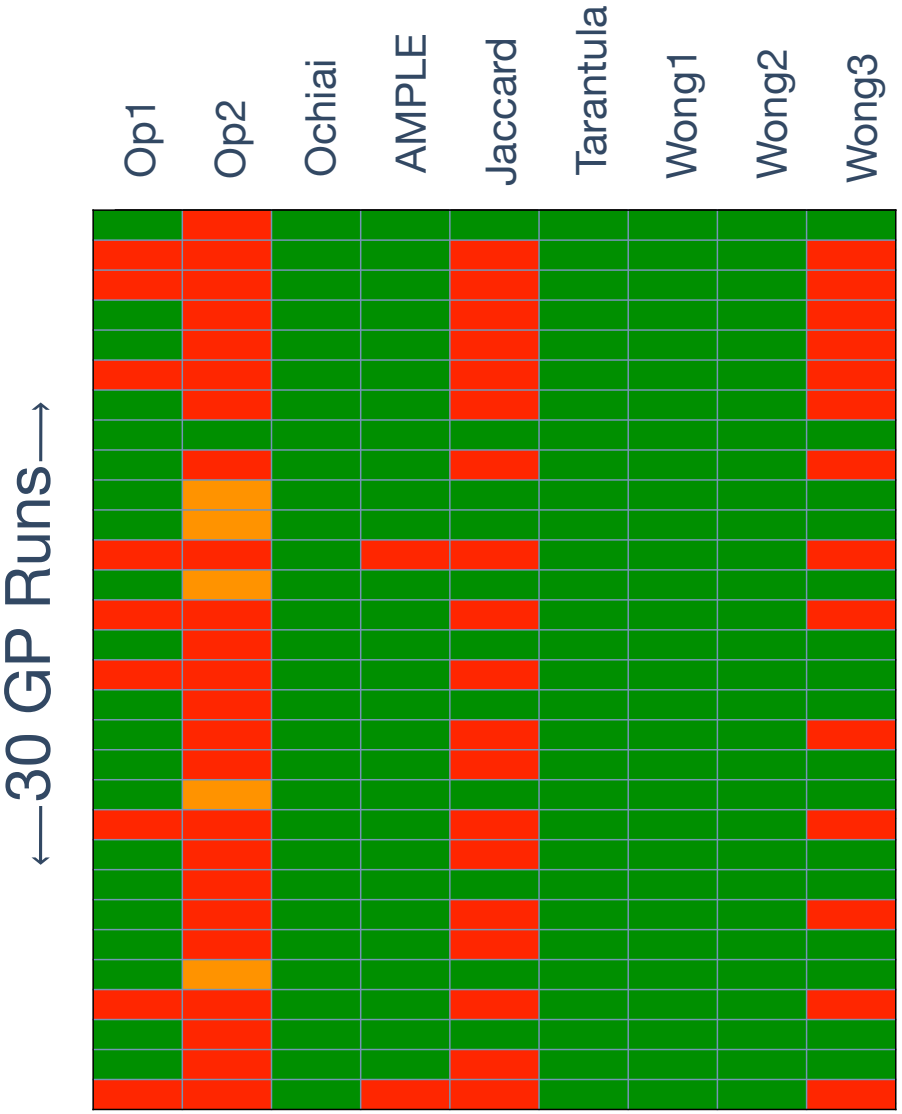
Diverse Risk Evaluation Formulae

While over 30 formulae have been proposed, none of them is universally superior to others and no best formula for all types of faults.

Name	Formula	Name	Formula
$ER1_a$	$\begin{cases} -1 & \text{if } n_f > 0 \\ n_p & \text{otherwise} \end{cases}$	$ER1_b$	$e_f - \frac{e_p}{e_p+n_p+1}$
$ER5_a$	$e_f - \frac{e_f}{e_p+n_p+1}$	$ER5_b$	$\frac{e_f}{e_f+n_f+e_p+n_p}$
$ER5_c$	$\begin{cases} 0 & \text{if } e_f < F \\ 1 & \text{otherwise} \end{cases}$	GP_2	$2(e_f+\sqrt{e_p+n_p})+\sqrt{e_p}$
Ochiai	$\frac{e_f}{(e_f+n_f)(e_f+e_p)}$	GP_3	$\sqrt{\lvert e_f^2-\sqrt{e_p}\rvert}$
Jaccard	$\frac{e_f}{e_f+n_f+e_p}$	GP_{13}	$e_f(1+\frac{1}{2e_p+e_f})$
AMPLE	$\lvert\frac{e_f}{F}-\frac{e_p}{P}\rvert$	GP_{19}	$e_f\sqrt{\lvert e_p-e_f+F-P\rvert}$
Hamann	$\frac{e_f+n_p-e_p-n_f}{P+F}$	Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$
Dice	$\frac{2e_f}{e_f+e_p+n_f}$	RusselRao	$\frac{e_f}{e_p+e_f+n_p+n_f}$
M1	$\frac{e_f+n_p}{n_f+e_p}$	SørensenDice	$\frac{2e_f}{2e_f+e_p+n_f}$
M2	$\frac{e_f}{e_f+n_p+2n_f+2e_p}$	Kulczynski1	$\frac{e_f}{n_f+e_p}$
Hamming	e_f+n_p	Kulczynski2	$\frac{1}{2}(\frac{e_f}{e_f+n_f}+\frac{e_f}{e_f+e_p})$
Goodman	$\frac{2e_f-n_f-e_p}{2e_f+n_f+e_p}$	SimpleMatching	$\frac{e_f+n_p}{e_p+e_f+n_p+n_f}$
Euclid	$\sqrt{e_f+n_p}$	RogersTanimoto	$\frac{e_f+n_p}{e_f+n_p+2n_f+2e_p}$
Wong1	e_f	Sokal	$\frac{2e_f+2n_p}{2_ef+2n_p+n_f+e_p}$
Wong2	e_f-e_p	Anderberg	$\frac{e_f}{e_f+2e_p+2n_f}$

Name	Formula
Wong3	$e_f-h, h=\begin{cases} e_p & \text{if } e_p \le 2 \\ 2+0.1(e_p-2) & \text{if } 2<e_p\le 10 \\ 2.8+0.01(e_p-10) & \text{if } e_p>10 \end{cases}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(ef+e_p)(n_f+n_p)(e_f+n_p)(e_p+n_f)}}$
Zoltar	$\frac{e_f}{e_f+e_p+n_f+\frac{10000 n_f e_p}{e_f}}$

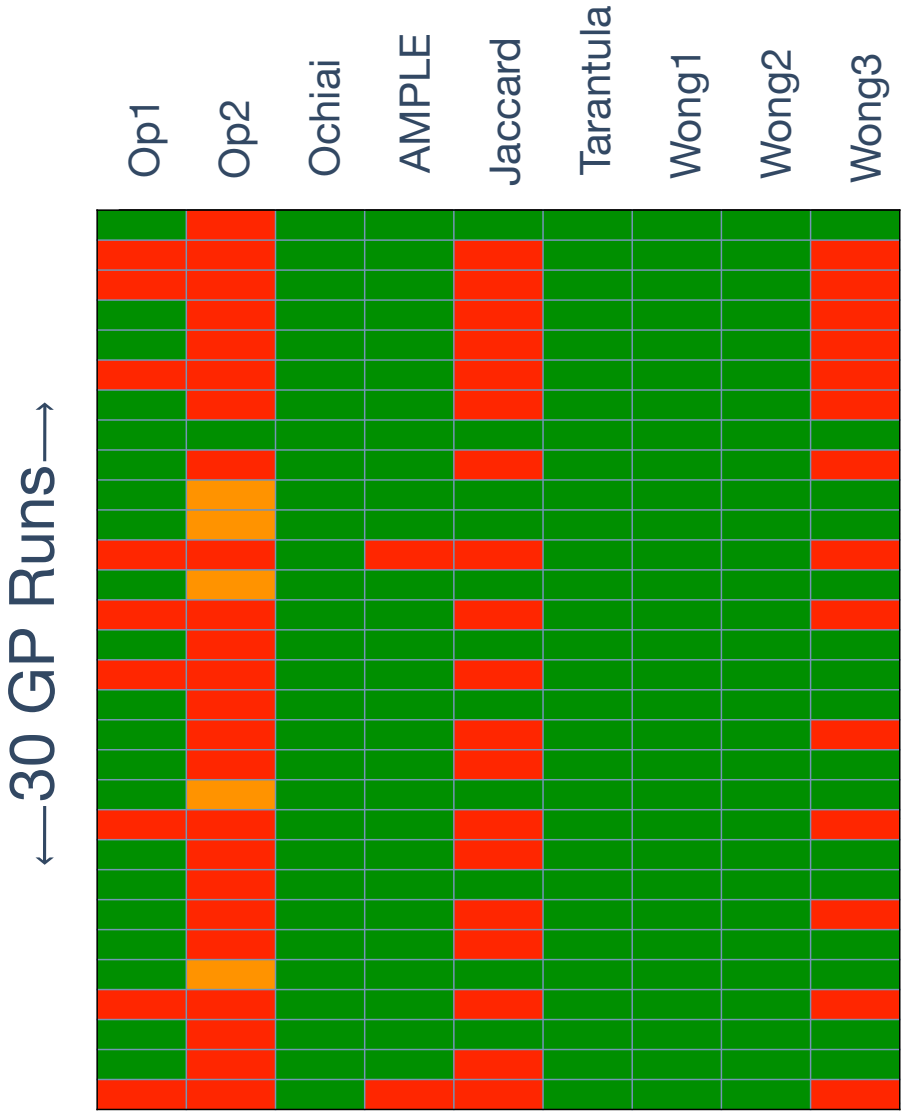
Genetic Algorithm for SBFL



4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

- Let's apply the **Genetic Algorithm (GA)** to **optimize** the formula for SBFL.
- **30 formulae** are generated.
  - Green – GP **outperforms** the other.
  - Orange – GP **exactly matches** the other.
  - Red – the other **outperforms** GP.

Genetic Algorithm for SBFL



4 Unix tools w/ 92 faults: 20 for training, 72 for evaluation.

- GP **completely outperforms** Ochiai, Tarantula, Wong 1, and Wong 2.
- GP **mostly outperforms** AMPLE.
- Op1, Jaccard, and Wong 3 are **tough to beat**.
- Op2 is very good but it is **not impossible to do better**.

Genetic Algorithm for SBFL

PLRG

ID	Refined Formula	ID	Refined Formula
GP01	$e_f(n_p + e_f(1+\sqrt{e_f}))$	GP16	$\sqrt{e_f^{\frac{3}{2}} + n_p}$
GP02	$2(e_f + \sqrt{n_p}) + \sqrt{e_p}$	GP17	$\frac{2e_f+n_f}{e_f-n_p} + \frac{n_p}{\sqrt{e_f}} - e_f - e_f^2$
GP03	$\sqrt{\lvert e_f^2 - \sqrt{e_p}\rvert}$	GP18	$e_f^3 + 2n_p$
GP04	$\sqrt{\lvert\frac{n_p}{e_p-n_p} - e_f\rvert}$	GP19	$e_f\sqrt{\lvert e_p - e_f + n_f - n_p\rvert}$
GP05	$\frac{(e_f+n_p)\sqrt{e_f}}{(e_f+e_p)(n_pn_f+\sqrt{e_p})(e_p+n_p)\sqrt{\lvert e_p-n_p\rvert}}$	**GP20**	$2(e_f + \frac{n_p}{e_p+n_p})$
GP06	$e_f n_p$	GP21	$\sqrt{e_f + \sqrt{e_f + n_p}}$
GP07	$2e_f(1 + e_f + \frac{1}{2n_p}) + (1+\sqrt{2})\sqrt{n_p}$	GP22	$e_f^2 + e_f + \sqrt{n_p}$
GP08	$e_f^2(2e_p + 2e_f + 3n_p)$	GP23	$\sqrt{e_f}(e_f^2 + \frac{n_p}{e_f} + \sqrt{n_p} + n_f + n_p)$
GP09	$\frac{e_f\sqrt{n_p}}{n_p+n_p} + n_p + e_f + e_f^3$	GP24	$e_f + \sqrt{n_p}$
GP10	$\sqrt{\lvert e_f - \frac{1}{n_p}\rvert}$	GP25	$e_f^2 + \sqrt{n_p} + \frac{\sqrt{e_f}}{\sqrt{\lvert e_p-n_p\rvert}} + \frac{n_p}{(e_f-n_p)}$
GP11	$e_f^2(e_f^2 + \sqrt{n_p})$	**GP26**	$2e_f^2 + \sqrt{n_p}$
GP12	$\sqrt{e_p + e_f + n_p - \sqrt{e_p}}$	GP27	$\frac{n_p\sqrt{(n_pn_f-e_f)}}{e_f+n_pn_f}$
GP13	$e_f(1 + \frac{1}{2e_p+e_f})$	GP28	$e_f(e_f + \sqrt{n_p} + 1)$
GP14	$e_f + \sqrt{n_p}$	GP29	$e_f(2e_f^2 + e_f + n_p) + \frac{(e_f-n_p)\sqrt{n_pe_f}}{e_p-n_p}$
GP15	$e_f + \sqrt{n_f + \sqrt{n_p}}$	GP30	$\sqrt{\lvert e_f - \frac{n_f-n_p}{e_f+n_f}\rvert}$

Theoretical Analysis

PLRG

- Given a test suite and a formula R, a program P consists of n elements $(s_1, s_2, \ldots, s_n)$.
- The set of elements can be divided into three subsets:
  - S_B^R – set of elements **ranked higher** than the faulty element s_f.

$$S_B^R = \{s_i \mid 1 \leq i \leq n \wedge R(s_i) > R(s_f)\}$$

  - S_F^R – set of elements **tied** to the faulty element s_f.

$$S_F^R = \{s_i \mid 1 \leq i \leq n \wedge R(s_i) = R(s_f)\}$$

  - S_A^R – set of elements **ranked lower** than the faulty element s_f.

$$S_A^R = \{s_i \mid 1 \leq i \leq n \wedge R(s_i) < R(s_f)\}$$

Theoretical Analysis

- Formula R_1 **dominates** formula R_2 (i.e., $R_1 \rightarrow R_2$) if, for each element s_i, R_1 gives a **better ranking comparison result** with the faulty element s_f than R_2.

$$S_B^{R_1} \subseteq S_B^{R_2} \wedge S_A^{R_2} \subseteq S_A^{R_1}$$

- Formula R_1 **equivalent** to formula R_2 (i.e., $R_1 \equiv R_2$) if they give the **same ranking comparison result** with the faulty element s_f.

$$S_B^{R_1} = S_B^{R_2} \wedge S_F^{R_1} = S_F^{R_2} \wedge S_A^{R_1} = S_A^{R_2}$$

- It means that $R_1 \equiv R_2$ if and only if $R_1 \rightarrow R_2$ and $R_2 \rightarrow R_1$.

Method-level Aggregation

```
public void testMe(int a) {
  util();
  if (a % 3 == 0) {
    ... // faulty code
  } else {
    ...
  }
}
public void util() {
  ...
}
```

- The `util` method is called by the `testMe` method.
- If we target **functions** as the unit of localization, the `util` method **might be tied** to the `testMe` method having the faulty code.
- **Method-level aggregation** is a technique to **aggregate** the statement- or branch-level ranks to the method level to get a more **accurate** fault localization result.

J. Sohn and S. Yoo. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In Proceedings of the 2017 International Symposium on Software Testing and Analysis, ISSTA 2017.

Hybrid SBFL

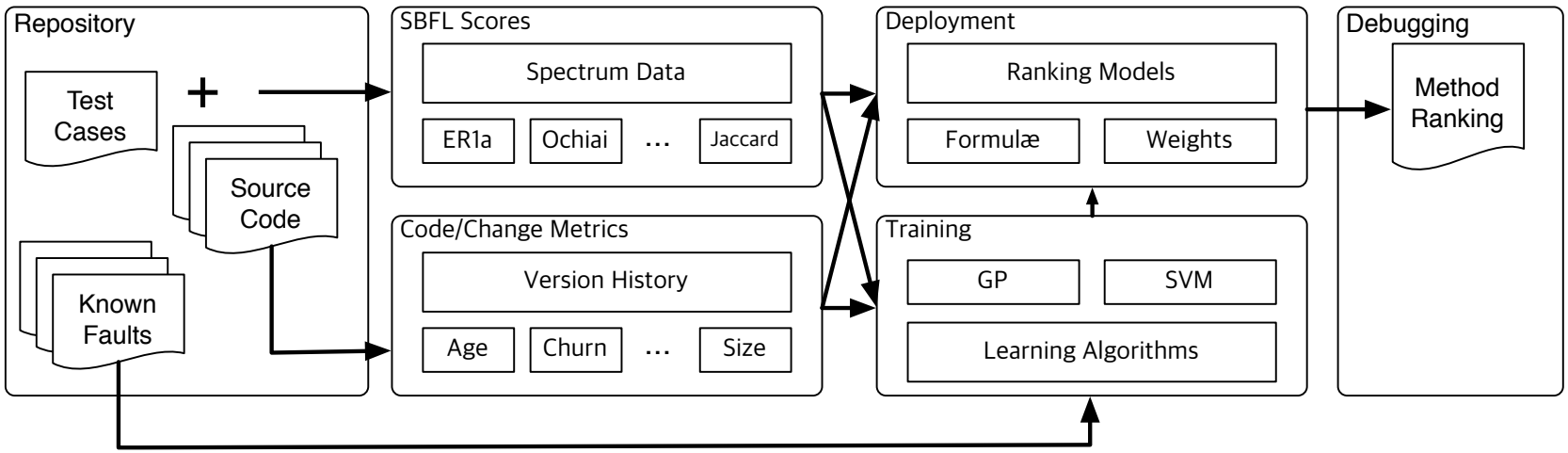
- People are realising that a **single formula** is **strongly limited**.
- Hybridization
  - Use **multiple formulae** at the same time.
  - Use SBFL formulae with some **additional input features**.
- As we accept more diverse input, fault localization become a learning problem, instead of design-a-technique problem.

Hybrid SBFL

- **Xuan & Monperrus (ICSME 2014)**
  - Simultaneously use **25 SBFL formulae**, using the **weighted sum** method.
  - Use a **learning-to-rank** technique to train the raking model (i.e., to optimize the weights that yield best ranks for training set faults)
- **Le et al. (ISSTA 2016)**
  - In addition to **35 SBFL formulae**, **Savant** use **invariant differences**.
  - Use Daikon to infer method invariants twice: with passing tests, and with failing test.
  - For the faulty method, two sets of invariant will tend to be more different.

Hybrid SBFL

J. Sohn and S. Yoo. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In Proceedings of the 2017 International Symposium on Software Testing and Analysis, ISSTA 2017.



- **Code and Change Metrics Features**
  - **Age** – how long has the given element existed in the code?
  - **Churn** – how frequently has the given element been changed?
  - **Complexity** – how complex is the given element?

Strengths and Weaknesses

- **Strengths**
  - Only requires what is already there: **coverage** and **test results**.
  - Relatively **intuitive**.
- **Weaknesses**
  - **Single formulae** are usually **limited**.
  - Does not work against **omission faults**
  - Does not work well against **multiple faults**.

Contents

Mutation-based Fault Localization (MBFL)

PLRG

- **Mutation testing** is a technique to evaluate the quality of a test suite by generating **mutants** of the program and checking whether the test suite can **kill** the mutants.

- Can we use **mutants** for **fault localization**?

- What would happen if we **mutate** a program having a **known fault**?

Mutation-based Fault Localization (MBFL)

PLRG

- Consider a **faulty program** P.
- Let m_f be a mutant of P that mutates the **faulty statement** and m_c be a mutant of P that mutates a **correct statement**.
- **Conjecture 1**: **failing tests** are more likely to **pass** on m_f than m_c.
- **Conjecture 2**: **passing tests** are more likely to **fail** on m_c than m_f.

Mutation-based Fault Localization (MBFL)

S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. International Conference on Software Testing, Verification, and Validation, ICST 2014. (**MUSE** – **MU**tation-ba**SE**d fault localization technique)

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

Mutation-based Fault Localization (MBFL)

S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. International Conference on Software Testing, Verification, and Validation, ICST 2014. (**MUSE** – **MU**tation-ba**SE**d fault localization technique)

Proportion of test cases that mutant *m* turns from fail to pass

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

Mutation-based Fault Localization (MBFL)

S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. International Conference on Software Testing, Verification, and Validation, ICST 2014. (**MUSE** – **MU**tation-ba**SE**d fault localization technique)

Proportion of test cases that mutant *m* turns from fail to pass

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

Proportion of test cases that mutant *m* turns from pass to fail

Mutation-based Fault Localization (MBFL)

S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. International Conference on Software Testing, Verification, and Validation, ICST 2014. (**MUSE** – **MU**tation-ba**SE**d fault localization technique)

Proportion of test cases that mutant *m* turns from fail to pass

$$\mu(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_P(s) \cap p_m|}{|f_P|} - \alpha \cdot \frac{|p_P(s) \cap f_m|}{|p_P|} \right)$$

Proportion of test cases that mutant *m* turns from pass to fail

Proportion of test cases that mutant *m* turns from pass to fail

where α is the **balancing factor**:

$$\alpha = \frac{f2p}{|mut(P)| \cdot |f_P|} \cdot \frac{|mut(P)| \cdot |p_P|}{p2f}$$

Summary

PLRG

1. Delta Debugging (DD)
   - Recursive Delta Debugging – `ddmin`
   - Hierarchical Delta Debugging
   - Probabilistic Delta Debugging (ProbDD)
   - Delta Debugging for Program Debloating
2. Information Retrieval based Fault Localization (IRFL)
   - Vector Space Model (VSM)
   - Tf-Idf
   - Cleansing Bug Reports and Source Code
   - VSM and Similarity
3. Spectrum-based Fault Localization (SBFL)
   - Genetic Algorithm for SBFL
   - Theoretical Analysis
   - Method-level Aggregation
   - Hybrid SBFL
4. Mutation-based Fault Localization (MBFL)

Next Lecture

- Testing Oracles

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