Survey of Software Bug Localization Using Dynamic Analysis

DR. Hua Jie Lee (Jason)

Senior Software Test Engineer
DOLBY LABORATORIES, Australia

February 21, 2013
Outline

1. Introduction
2. Background
3. Test Coverage Information Approaches
4. Slicing Approaches
5. Statistical Approaches
6. State-based Approaches
7. Machine Learning Approaches
8. Practical Tools
9. Recent Bug Localization Research Trend
10. Summary
11. Challenges
Common Terms Used I

1. Statement – program statements
2. Predicate – specific program point such as conditions of if-statements, return expression of functions etc.
3. Metric – numeric function and as a standard of measurement. Not related to metric spaces (where it defines the distance between any two points in the space)
4. Rank percentages – percentages of the program code need/need not be examined to locate software bug
Common Terms Used II

1. **Bug** – code segment of program code which causes failure of the program
2. **Single bug** – also known as one-bug
3. **Fault** – incorrect statement or data definition in a program
4. **Failure** – deviation of the observed behaviour of a program, or a system, from its specification
Binary weighting approach – using program spectra information without introducing any weights

e_{s,t} – test execution coverage of statement s for a respective test t

DU-PAIR refers to associating a conditional branch with statements that exist before the conditional statement or by the conditional statement itself
Common Terms Used IV

1. Graph – collection of vertices and edges
2. Vertices – better known as nodes
3. Edges – connector of pairs of vertices
Introduction

- Software bugs cost U.S economy USD 59.5 billion dollars (0.6% of GDP) in 2002 [Hailpern et al. ’02].
- Dynamic analysis - one of the software bug localization techniques which could improve efficiency in locating bugs.
## Software Bug Incidents I

<table>
<thead>
<tr>
<th>YEAR</th>
<th>COMPANY</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013</td>
<td>Network manufacturers</td>
<td>40–50 million devices vulnerable to the attack from bugs found in networking equipment (UPnP standard)</td>
</tr>
<tr>
<td>2012</td>
<td>Knight Capital</td>
<td>Computer bug costs the loss of USD 440 million where a series of automatic order of company’s fund were executed instead of over period of days</td>
</tr>
</tbody>
</table>
## Software Bug Incidents II

<table>
<thead>
<tr>
<th>YEAR</th>
<th>COMPANY</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>Apple</td>
<td>Alarm not working on the 1 Jan 2011. Caused customers missed flights and to work. Suspected to be the bug in the complexity of the algorithm of week-number</td>
</tr>
<tr>
<td>2010</td>
<td>NAB</td>
<td>NAB system crashed. Delayed payments to customers. Due to the bug in the batch processing software code that contained instructions on how the system operate in the batch processing operations</td>
</tr>
</tbody>
</table>
Software Bug Incidents III

<table>
<thead>
<tr>
<th>YEAR</th>
<th>COMPANY</th>
<th>OUTCOME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1990</td>
<td>AT &amp; T Wireless</td>
<td>Massive shutdown of their network [Neumann, 1990] due to a single line of code (bug) which is part of software upgrade to speed up calling</td>
</tr>
<tr>
<td>1962</td>
<td>NASA</td>
<td>Incorrect signals guidance (due to a missing superscript bar in a written formula) causes the launch of first Mariner mission Mariner I rocket to off course</td>
</tr>
</tbody>
</table>

And the list goes on, and on ...
Background

1. Static analysis [Weiser et al. ’79, Weiser et al. 81, Weiser et al. 82].

2. Dynamic analysis
   - State-based Approaches[Zeller ’01, Zeller et al. ’05]
What is Program Spectra I?

1. A type of dynamic analysis approach
2. A path (or program) spectrum is a collection of data that provides a specific view of the dynamic behavior of software.
3. Contains information about which part of a program (individual statements, basic blocks or functions) was executed during the execution of several test cases.
What is Program Spectra II?

1. \( a_{np} \) – number of pass test cases not executing respective statements
2. \( a_{nf} \) – number of fail test cases not executing respective statements
3. \( a_{ep} \) – number of pass test cases executing respective statements
4. \( a_{ef} \) – number of fail test cases executing respective statements
5. \( totF \) – total number of fail test cases
6. \( totP \) – total number of pass test cases
Example of test coverage information (frequency counts) with tests $T_1 \ldots T_5$

<table>
<thead>
<tr>
<th>S</th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_1</td>
<td>60</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>38</td>
</tr>
<tr>
<td>S_2</td>
<td>70</td>
<td>65</td>
<td>90</td>
<td>52</td>
<td>45</td>
</tr>
<tr>
<td>S_3</td>
<td>25</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>S_4</td>
<td>59</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>S_5</td>
<td>0</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>57</td>
</tr>
</tbody>
</table>

Test Result: Fail, Fail, Fail, Pass, Pass
Example of test coverage information (frequency counts) with tests $T_1 \ldots T_5$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>60</td>
<td>2</td>
<td>100</td>
<td>50</td>
<td>38</td>
</tr>
<tr>
<td>$S_2$</td>
<td>70</td>
<td>65</td>
<td>90</td>
<td>52</td>
<td>45</td>
</tr>
<tr>
<td>$S_3$</td>
<td>25</td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>$S_4$</td>
<td>59</td>
<td>0</td>
<td>37</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>$S_5$</td>
<td>0</td>
<td>80</td>
<td>0</td>
<td>0</td>
<td>57</td>
</tr>
</tbody>
</table>

Test Result | Fail | Fail | Fail | Pass | Pass

Example of test coverage information (binary) and program spectra with tests $T_1 \ldots T_5$

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$a_{np}$</th>
<th>$a_{nf}$</th>
<th>$a_{ep}$</th>
<th>$a_{ef}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Test Result | Fail | Fail | Fail | Pass | Pass
Definition of various ranking metrics used 1

Numerous metrics proposed to predict which statements are most likely to be buggy.

- Software automated debugging tools (Tarantula (Tar), Ample, Wong3, Wong4, Zoltar, CBI Inc),
- Self-organizing-map (Hamann, Kulczynski2 (Kul2)),
- Zoology (Ochiai), and
- Botany (Jaccard, Sorensen-Dice(Sor), Anderberg, Rogers and Tanimoto (Rog), Russell and Rao(Rus), Simple-Matching, Ochiai2).

- Most spectra metrics formula used primarily on $a_{ef}$, secondarily on $a_{ep}$. 

More Spectra Metrics
Benchmarks

Description of Siemens Test Suite, Unix, Concordance and Space

<table>
<thead>
<tr>
<th>Program</th>
<th>Ver</th>
<th>1 Bug</th>
<th>2 Bugs</th>
<th>3 Bugs</th>
<th>LOC</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>41</td>
<td>37</td>
<td>604</td>
<td>—</td>
<td>173</td>
<td>1608</td>
</tr>
<tr>
<td>schedule</td>
<td>9</td>
<td>8</td>
<td>—</td>
<td>—</td>
<td>410</td>
<td>2650</td>
</tr>
<tr>
<td>schedule2</td>
<td>10</td>
<td>9</td>
<td>28</td>
<td>131</td>
<td>307</td>
<td>2710</td>
</tr>
<tr>
<td>print_tokens</td>
<td>7</td>
<td>6</td>
<td>—</td>
<td>—</td>
<td>563</td>
<td>4130</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>44</td>
<td>508</td>
<td>4115</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>23</td>
<td>245</td>
<td>782</td>
<td>406</td>
<td>1052</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>29</td>
<td>34</td>
<td>16</td>
<td>563</td>
<td>5542</td>
</tr>
<tr>
<td>Col</td>
<td>28</td>
<td>28</td>
<td>147</td>
<td>2030</td>
<td>308</td>
<td>156</td>
</tr>
<tr>
<td>Cal</td>
<td>18</td>
<td>18</td>
<td>115</td>
<td>1475</td>
<td>202</td>
<td>162</td>
</tr>
<tr>
<td>Uniq</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>114</td>
<td>143</td>
<td>431</td>
</tr>
<tr>
<td>Spline</td>
<td>13</td>
<td>13</td>
<td>20</td>
<td>174</td>
<td>338</td>
<td>700</td>
</tr>
<tr>
<td>Checkeq</td>
<td>18</td>
<td>18</td>
<td>56</td>
<td>502</td>
<td>102</td>
<td>332</td>
</tr>
<tr>
<td>Tr</td>
<td>11</td>
<td>11</td>
<td>17</td>
<td>90</td>
<td>137</td>
<td>870</td>
</tr>
<tr>
<td>Concordance</td>
<td>13</td>
<td>11</td>
<td>—</td>
<td>—</td>
<td>1492</td>
<td>372</td>
</tr>
<tr>
<td>Space</td>
<td>38</td>
<td>15</td>
<td>5</td>
<td>1</td>
<td>9059</td>
<td>13585</td>
</tr>
</tbody>
</table>
How to measure the performance of locating bugs?

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$a_{np}$</th>
<th>$a_{nf}$</th>
<th>$a_{ep}$</th>
<th>$a_{ef}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Test Result</td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
<td>Pass</td>
<td>Pass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to measure the performance of locating bugs?

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$a_{np}$</th>
<th>$a_{nf}$</th>
<th>$a_{ep}$</th>
<th>$a_{ef}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Test Result</td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
<td>Pass</td>
<td>Pass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Definition of Tarantula metric

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>$\frac{a_{ef}}{a_{ef} + a_{np}}$ + $\frac{a_{ep}}{a_{ep} + a_{np}}$</td>
</tr>
</tbody>
</table>
How to measure the performance of locating bugs?

<table>
<thead>
<tr>
<th></th>
<th>$T_1$</th>
<th>$T_2$</th>
<th>$T_3$</th>
<th>$T_4$</th>
<th>$T_5$</th>
<th>$a_{np}$</th>
<th>$a_{nf}$</th>
<th>$a_{ep}$</th>
<th>$a_{ef}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_2$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$S_3$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_4$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>$S_5$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Test Result</td>
<td>Fail</td>
<td>Fail</td>
<td>Fail</td>
<td>Pass</td>
<td>Pass</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Definition of Tarantula metric

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>$\frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{ep}}{a_{ep} + a_{np}}$</td>
</tr>
</tbody>
</table>

- Let $S_1$ as the bug.
- Tarantula ($S_1=0.5, S_2=0.5, S_3=0.5714, S_4=0.5714, S_5=0.4000$)
- Rank percentages has been used - 10% refers to 10% of the statements in the program must be inspected before the bug is located.
- Ties handled differently - Best case = $\frac{3}{5} \times 100\%$. Worst case = $\frac{4}{5} \times 100\%$. Average case = $\frac{3.5}{5} \times 100\%$. 
Two approaches to distinguish buggy statements

*discrete* and *continuous approach*

**Discrete Approach**

1. Uses three-color system (*Red*, *Green* and *Yellow*)
2. Statements executed by pass test case(s) visualized *Green*
3. Statements executed by fail test case(s) visualized *Red*
4. Statements executed by BOTH pass and fail test case(s) visualized *Yellow*
5. Unable to distinguish bug if buggy statement executed by both pass and fail test cases
Jones et al. Proposed Approach II - Continuous

Definition (Color Visualization using Continuous Approach for a statement \( s \))

\[
\text{color} = \text{low color (red)} + \left( \frac{a_{ep}}{\text{totP}} \times 100\% + \frac{a_{ef}}{\text{totF}} \times 100\% \right) \times \text{color range}
\]

1. Map color for statements
2. *low color* is defined as Red for one of the color spectrum
3. *color range* defined as the other end of spectrum (0 - Red, 100 - Green)
4. Statement that is executed most of the time either by pass/fail test cases assigned a brighter color.
**Jones et al. Proposed Approach III**

**Definition (Brightness for a statement s)**

\[ \text{bright} = \max \left( \frac{a_{ep}}{\text{totP}} \times 100\%, \frac{a_{ef}}{\text{totF}} \times 100\% \right) \]

**Definition (Tarantula metric)**

\[ \frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{ep}}{a_{ep} + a_{np}} \]
Jones et al. Proposed Approach IV

1. Define brightness component in the scale of 0 to 100
2. Statement executed less by test cases assigned darker color and vice versa
3. Gimp [GIMP] is used to define values of colors
4. Color for statement $s$ would be a single number representing color spectrum
More pass test cases execute program statement, color mapped towards Green spectrum
More fail test cases execute program statement, color mapped towards Red spectrum
Same number of pass and fail test cases execute program statement, color mapped towards Yellow spectrum
Color could distinguish buggy from non-buggy statements
Space [Do et al., 2005] is used as benchmark
1. Tarantula could not distinguish the multiple bugs in the program by judging the color spectrum; only single bug programs.

2. Evaluated with other approaches and outperformed others; Intersection model, Union model, Nearest Neighbor model [Renieres et al., 2003] and Cause Transitions [Cleve et al., 2005]
Wong et al. Proposed Approach I [Wong et al., 2007]

1. Proposed several spectra metrics; Wong1, Wong2 and Wong3

2. Less weight given to pass test cases if particular statement executed by more pass test cases

3. $\chi$ Suds [Telecordia, 1998] is used to instrument and perform test executions on the program
Wong et al. Proposed Approach II

1. Wong1 - depends on the number of fail test cases.
2. Wong2 - depends on both the number of pass and fail test cases.
3. Wong3 - used a constant value, $\alpha = 0.001$ to adjust weights for test cases
4. Used Siemens Test Suite as benchmark
5. Proposed another metric, Wong4 [Wong et al., 2009], adjust both pass and fail test cases and $\alpha = 0.0001$

2. Used If-Then-Else (ITE2) model program to understand single bug programs.

3. Model program can be used to generate and simulate different program models (if-then-else, nested conditional expression).
ITE28 Model Program

```c
if (t1())
    s1(); /* S1 */
else
    s2(); /* S2 */
if (t2())
    x = True; /* S3 */
else
    x = t3(); /* S4 - BUG */
```
Naish et al. Proposed Approach II

1. Proposed several metrics used in other fields and compare with $O$ and $O^p$ metrics
2. Observed bug localization performance is optimal using $O$ and $O^p$ on single bug programs
3. Empirically evaluated Siemens Test Suite, Unix and Space datasets
4. Compared on several metrics

Empirical Results
Xie et al. Proposed Approach I
[XiaoYuan Xie et al., 2010]

1. Proposed post-ranking of program statements evaluated with spectra metrics
2. Group statements into two suspicious groups; $G_S$ and $G_U$ according to heuristics
3. $G_S$ - any statements executed by any fail test case
4. $G_U$ - statements not executed by any fail test case
Xie et al. Proposed Approach II

1. Value of 1 is added to statement(s) in $G_S$
2. Small value (minimum metric value) is assigned to statement(s) in $G_U$
3. Found improved bug localization performance using Wong2, Wong3, Scott and M2 metrics
Debroy et al Proposed Approach I

[Vidroha et al., 2010]

1. Proposed post-ranking of program statements
2. Group statements based on the fail test cases which execute respective statements \( (a_{ef}) \)
3. Group statements based on same number of failed test cases \( (G_f) \); \( f \) refers to \( f \) fail test cases
4. Group statements executed with most fail test cases sorted at the top
5. In each group of \( G_f \), statements sorted again based on metric value
Debroy et al [Vidroha et al., 2010] Algorithm

**Input:** statement-based spectra coverage, ranked statements of program with respective spectra metrics

**Output:** ranked statements of program that are likely to be buggy

Group statements (in descending order) based on number of failed test cases

```
foreach statement s do
    if statement s is executed in f fail test cases then
        Group the statement s as part of group G_f;
    end
end
```

Sort group G_f according to f fail test cases (in descending order)

Sort again the statements based on their metric value within the sorted group G_f
Debroy et al. Proposed Approach II

1. Used $\chi$suds [Telecordia, 1998] for program instrumentation and gather coverage information
2. Evaluated on single bug programs of Siemens Test Suite, grep and gzip datasets
3. Using Tarantula metric [Jones et al., 2005] and Radial Basis function [Wong et al., 2008]
4. Found their strategy is more effective than without using grouping strategy
1. Proposed block-based spectra metric and gathered block-based spectra coverage
2. Introduced Ochiai and Jaccard metrics and compared with Tarantula
3. Evaluated on Siemens Test Suite and observed Ochiai outperformed Tarantula metric
Developed Zoltar metric, based on the modification of Jaccard metric

Used this metric to detect dynamic invariants instead of ranking program spectra
Renieres et al. Proposed Approach I

[Renieres et al., 2003]

1. Proposed to locate bugs based on *difference spectra* (differences between test cases) and *distance spectra* (using Nearest Neighbor model)

2. Implemented a tool, WHITHER in their study

3. *difference spectra* consists of two different models, namely *Intersection model* and *Union model*

4. *Intersection model*: $\cap S_p - S_f$

5. *Union model*: $S_f - \cup S_p$
Renieres et al Algorithm [Renieres et al., 2003]

**Input:** block-based spectra coverage of a fail test case and, all pass test cases

**Output:** Set of blocks of program likely to be buggy

**Intersection Model -**
\( \cap S_p \) (blocks program executed by ALL passed test cases) \( - S_f \);

**Union Model -**
\( S_f \) \( - \cup S_p \) (any block program executed by the passed test cases)

**Nearest Neighbor Model -**

foreach pass test case do
  **Coverage Type:** Apply Hamming distance on the binary execution counts of \( S_p \) and \( S_f \)
  **Permutation Type:** Sort blocks of program in \( S_p \) and \( S_f \) based on frequency execution counts & apply Hamming distance on the test cases
end

Choose the pair of pass and fail test case with the least Hamming distance

Set of blocks of program likely to be buggy is returned
Renieres et al. Proposed Approach II

1. *Nearest Neighbor* model – one of the pass test cases that is most similar to the selected fail test cases would be chosen

2. Evaluated 109 single bug programs of Siemens Test Suite

By not examine more than 10% of the program code:

1. *Intersection model* – 1 bug
2. *Union model* – 6 bugs
3. *Nearest-neighbor using Coverage Type* – 5 bugs
4. *Nearest-neighbor using Permutation Type* – 18 bugs
Hsu et al. Proposed Approach I [Hsu et al., 2008]

1. Proposed **Ranking, Analysis, and Pattern Identification for Debugging** (RAPID)

2. Instrumented program branch-level and gathered branch-based spectra coverage

3. Mapped each branch to respective program spectra properties $a_{ef}$, $a_{ep}$, $a_{nf}$ and $a_{np}$
Hsu et al. Proposed Approach II

1. Identify common branches patterns using data mining approach (BI-Directional Extension (BIDE)) [Wang et al., 2004].
2. Longest subsequence of branches is chosen and ranked most buggy
3. If the bug not found, append previous branches to next longest subsequence of branches in $L$
4. Performed a case study on replace program
**Hsu et al. Algorithm [Hsu et al., 2008]**

**Input**: instrumented branches $B$ of program evaluated with Tarantula, branch-based spectra coverage

**Output**: sequence of branches most likely to be buggy

Map branch $MetValue$ to matching program branch in branch-based spectra coverage;

Identify sequences of buggy branches (bug signatures);

**foreach** branch-based spectra coverage of fail test case **do**

  **if** $MetValue$ of all the mapped program branches in the test case $< 0.6$ (threshold value) **then**

  Eliminate the fail test case;

  **else**

  Store the spectra coverage of the fail test case in a new list, $L$;

**end**

**end**

Identify common patterns of spectra coverage in $L$ using BIDE;

Present longest subsequence of branches as most likely buggy;
Evaluation Results on Empirical Benchmarks of Single bug Programs

Average Rank Percentages (%) for Single bug Siemens, Unix, Concordance and Space programs

<table>
<thead>
<tr>
<th>Metric</th>
<th>Siemens</th>
<th>Unix</th>
<th>Concordance</th>
<th>Space</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O, O^p$</td>
<td>15.69</td>
<td>20.47</td>
<td>10.11</td>
<td>1.60</td>
<td>16.55</td>
</tr>
<tr>
<td>Wong3</td>
<td>16.28</td>
<td>20.47</td>
<td>10.15</td>
<td>1.60</td>
<td>16.84</td>
</tr>
<tr>
<td>Zoltar</td>
<td>15.71</td>
<td>21.24</td>
<td>10.11</td>
<td>1.76</td>
<td>16.88</td>
</tr>
<tr>
<td>Wong4</td>
<td>16.27</td>
<td>22.11</td>
<td>11.35</td>
<td>1.60</td>
<td>17.56</td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>16.13</td>
<td>22.58</td>
<td>10.21</td>
<td>2.01</td>
<td>17.65</td>
</tr>
<tr>
<td>JacCube</td>
<td>18.31</td>
<td>22.15</td>
<td>10.43</td>
<td>1.87</td>
<td>18.54</td>
</tr>
<tr>
<td>Ochiai</td>
<td>20.17</td>
<td>23.37</td>
<td>11.19</td>
<td>2.12</td>
<td>20.00</td>
</tr>
<tr>
<td>Jaccard</td>
<td>23.47</td>
<td>23.85</td>
<td>17.68</td>
<td>3.19</td>
<td>22.15</td>
</tr>
<tr>
<td>CBI Log</td>
<td>24.46</td>
<td>29.59</td>
<td>22.63</td>
<td>6.01</td>
<td>25.37</td>
</tr>
<tr>
<td>Tarantula</td>
<td>24.12</td>
<td>30.65</td>
<td>20.03</td>
<td>6.22</td>
<td>25.53</td>
</tr>
<tr>
<td>Russell</td>
<td>28.36</td>
<td>31.99</td>
<td>21.03</td>
<td>17.30</td>
<td>28.85</td>
</tr>
<tr>
<td>Binary</td>
<td>28.36</td>
<td>31.99</td>
<td>21.03</td>
<td>17.30</td>
<td>28.85</td>
</tr>
</tbody>
</table>
Average Rank Percentages (%) vs Spectra Metrics in Respective Datasets

- Siemens
- Unix
- Concordance
- Space
- Combined

Dataset

Average Rank Percentages (%)
## Rank Percentages

Results of Best, Average, Worst, Median, 1st Quartile and 3rd Quartile Rank Percentages (%) for Siemens Test Suite (Single bug Programs)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Median</th>
<th>1st Quartile</th>
<th>3rd Quartile</th>
</tr>
</thead>
<tbody>
<tr>
<td>O, O^p</td>
<td>10.69</td>
<td>15.69</td>
<td>20.68</td>
<td>5.80</td>
<td>2.34</td>
<td>24.57</td>
</tr>
<tr>
<td>Zoltar</td>
<td>10.72</td>
<td>15.71</td>
<td>20.71</td>
<td>6.10</td>
<td>2.34</td>
<td>24.57</td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>11.13</td>
<td>16.13</td>
<td>21.12</td>
<td>6.25</td>
<td>2.34</td>
<td>24.57</td>
</tr>
<tr>
<td>Wong4</td>
<td>11.27</td>
<td>16.27</td>
<td>21.28</td>
<td>6.85</td>
<td>2.34</td>
<td>25.78</td>
</tr>
<tr>
<td>Wong3</td>
<td>11.29</td>
<td>16.28</td>
<td>21.28</td>
<td>5.80</td>
<td>2.34</td>
<td>25.78</td>
</tr>
<tr>
<td>Ochiai</td>
<td>15.17</td>
<td>20.17</td>
<td>25.16</td>
<td>9.95</td>
<td>2.76</td>
<td>27.34</td>
</tr>
<tr>
<td>Jaccard</td>
<td>18.48</td>
<td>23.48</td>
<td>28.47</td>
<td>17.48</td>
<td>2.80</td>
<td>35.89</td>
</tr>
<tr>
<td>Tarantula</td>
<td>19.13</td>
<td>24.12</td>
<td>29.12</td>
<td>18.62</td>
<td>2.80</td>
<td>38.69</td>
</tr>
<tr>
<td>CBI Log</td>
<td>18.79</td>
<td>24.46</td>
<td>30.13</td>
<td>19.09</td>
<td>2.80</td>
<td>40.57</td>
</tr>
<tr>
<td>Russell</td>
<td>0.90</td>
<td>28.36</td>
<td>55.82</td>
<td>29.87</td>
<td>21.77</td>
<td>37.71</td>
</tr>
<tr>
<td>Overlap</td>
<td>0.90</td>
<td>28.39</td>
<td>55.88</td>
<td>29.87</td>
<td>21.77</td>
<td>37.71</td>
</tr>
<tr>
<td>Ample</td>
<td>25.82</td>
<td>31.39</td>
<td>36.96</td>
<td>18.30</td>
<td>3.04</td>
<td>67.97</td>
</tr>
</tbody>
</table>
Breakdown of Average Rank Percentages for Single bug Programs on $O^p$

Average Rank Percentages and Median of $O^p$ (%) vs Single bug Programs

Median
Breakdown of Average Rank Percentages for Single bug Programs (Space) on $O^p$

Average Rank Percentages and Median of $O^p$ (%) vs Single bug Programs (WOBin and 10 Bins of Space)

End of Empirical Results
Santelices et al. Proposed Approach I
[Santelices, R. et al., 2009]

1. Investigated 3 types of spectra; statement-based spectra, branch-based spectra and DU-PAIR spectra
2. Used DUA-FORENSICS tool [Santelices, R. et al., 2007] for program instrumentation
3. Evaluated on Java programs using Ochiai metric for statement ranking
4. Use similar approach [Hsu et al., 2008] for branch and DU-PAIR instrumentation
DU-PAIR spectra showed most improvement in bug localization performance.

Time incurred to instrument DU-PAIR took longer than statement-based and branch-based spectra.

On average (all Java programs), rank percentages:

1. statement-based - 11.49%
2. branch-based - 10.24%
3. DU-PAIR - 9.02%
Hao et al. Proposed Approach Algorithm
[Hao, D. et al., 2008]

Input: statement-based spectra coverage
Output: $Appear_s$ and $Fail_s$

Find the membership grade of test case $t$;

$$
\mu_t = \frac{1}{\sum_{k=1}^{N_s(P)} e_{s,t}} \quad // \quad N_s(P) -$total program statements$

Apply conditional probability [Newmark, J., 1988];

foreach statement $s$ in program do

- Find the set of test cases that execute statement $s$, $Appear_s$;
- Find the set of test cases that execute statement $s$ and fail, $Fail_s$;

end
Hao et al. Proposed Approach Algorithm (continued)

**Input:** $\text{Appear}_s$ and $\text{Fail}_s$

**Output:** ranked program statements likely to be buggy

**foreach statement $s$ in program do**

- **Calculate the maximum of the membership grade for statement $s$ in $\text{Appear}_s$**;
- Determine the maximum membership grade of test $t$ in $\text{Appear}_s$;
- Aggregate the maximum of the membership grade for statement $s$ to $A_s$;
- **Calculating the maximum of the membership grade for statement $s$ in $\text{Fail}_s$**;
- Determine the maximum membership grade of test $t$ in $\text{Fail}_s$;
- Aggregate the maximum of the membership grade for statement $s$ to $F_s$;
- Calculate suspiciousness of statement, $s$ as $P(s) = |F_s|/|A_s|$;

**end**

Rank statements based on suspiciousness $P(s)$ in decreasing order;
Hao et al. Proposed Approach I


2. Evaluated on Siemens Test Suite [Do et al., 2005], Tiny C Compiler (TCC) [Bellard, F., 2010], desk calculator (DC) [Morris, R. et al., 1983] and Counter of directory sizes (CNT) [CNT, 2010] datasets.
Hao et al. Proposed Approach II

1. Compared with other approaches namely Dicing [Agrawal, H. et al., 1995] and Tarantula [Jones et al., 2005]
2. Fuzzy set approach had similar bug localization performance as Dicing approach
3. Only outperformed Nearest Neighbor approach

1. Investigated the accuracy of bug localization performance on human-seeded faults
2. Claimed that manually seeded fault datasets (Siemens Test Suite) is inaccurate for bug localization study
3. For example, Siemens was manually seeded faults by the researchers [SIR, 2010]

1. Proposed using Concordance (consists of 13 program versions)
2. Mutant generator tool *mutgen* [Ali, S. et al., 2009] used to generate small changes of code; treated as potential bugs
3. Compare the mutant generated program code and program code written by students

1. Found similar effectiveness of bug localization performance using Tarantula metric.
2. Examine within 1% of the program code of Concordance program;
3. Concordance programs written by the students - 21% of the bugs
4. Concordance programs with the bugs generated from the mutant generator - 28% of the bugs
Hua et al. Proposed Approach
[HuaJie, L. et al., 2010]

1. Proposed frequency weighting function approach
2. Use more information of the test coverage (frequency counts)
3. Use the adapted sigmoid function $M$ to map the non-zero frequency counts
4. If no execution (frequency count is zero), the function $M$ returns 0.

**Definition (Adapted Sigmoid Function, $M$)**

$$M(k_{st}) = \begin{cases} 
\frac{1}{e^{-\alpha k_{st}}+1} & \text{if } k_{st} > 0 \\
0 & \text{otherwise}
\end{cases}$$

where $k$ is frequency counts of statement $s$ of test $t$, $\alpha$ is a constant value
Hua Jie L. et al. Proposed Approach II

1. Found to improve the bug localization performance compared to using binary coverage information.

2. Single bug programs, the improvement of the bug localization performance using $O^p$ ranges in average rank percentages of 0.02% to 0.86%.

3. Multiple-bug Siemens and subset of Unix programs, the improvement of the bug localization performance using Kulczynski2 ranges in average rank percentages of 1.52% to 4.08%.
THINGS TO DO WHILE WAITING FOR YOUR EXPERIMENT TO FINISH (OR SIMULATION TO RUN, OR CODE TO COMPILE, OR...)

DEVELOP YOUR LATENT PSYCHIC POWERS.

c'mon, data! Support my hypothesis!

CHECK TO SEE IF ANYONE WROTE YOU IN THE LAST 10 SECONDS.
i have no friends.

READ A COUPLE OF RESEARCH PAPERS.

ZZZZZZ...

WATCH LIFE PASS YOU BY.

compile... compile...

www.phdcomics.com

Two types; static slicing and dynamic slicing
Static Slicing

1. Parts of the program (statements or blocks of program) that affect the value of particular program variable
2. Rely on information from control flow graph (CFG).
Korel et al. introduced dynamic slicing using dynamic analysis [Korel, B. et al., 1990]

Determine the parts of the program (a set of statements or blocks of the program) that affect the value of a particular program variable

Relies on the respective test cases to determine the dynamic slice
Dynamic Slicing II

1. Size of a typical dynamic slice always smaller than static slice

2. Consider only parts of program (statements/blocks) that is executed by test cases

3. Better known as an execution slice
Parts of the program that appear in an execution slice but not in another execution slice [Lyle, J.R. et al., 1987]
Chen et al Proposed Approach I
[Chen, TY. et al., 1993]

1. Used dynamic slice to narrow down search of the bug
2. Two possible dynamic slices, successful execution slice, $Slice_{suc}$ and fail execution slice $Slice_{fail}$
3. Used test inputs of dynamic slices to determine the successful and fail execution slice
Chen et al Proposed Approach II

1. Successful slice - output value of its slicing variables correct when applied with all associated set of test input of dynamic slice

2. Fail slice - output value of its slicing variables incorrect when applied with all associated set of test input of dynamic slice
Chen et al Proposed Approach III

1. Proposed several strategies to construct dynamic dice

1. Remove program statements in $Slice_{fail}$ executed in the $Slice_{suc}$.
2. Remove program statements in $Slice_{fail}$ which have been executed at least once in $Slice_{suc}$.
3. Remove program statements in $Slice_{fail}$ which have been executed in all $Slice_{suc}$.
4. Remove program statements which have been executed in ALL $Slice_{fail}$ from $Slice_{suc}$.
5. Remove program statements which have been executed in ALL $Slice_{fail}$ and $Slice_{suc}$.
1. Found first strategy better than second when multiple-bug of the program are in different $Slice_{suc}$

2. Found dynamic dice is more effective to locate bugs than using static dice approach [Lyle, J.R. et al., 1987]
Agrawal et al Proposed Approach

[Agrawal, H. et al., 1995]

1. Extended study of dynamic slicing and dicing, to study bug localization performance

2. Used base-version program code as *oracle* of the program

3. *successful execution slice* - slice of each test execution where output of fault-seeded program code similar to base version program code

4. *fail execution slice* - slice of each test execution where output of fault-seeded program code differs from base version program code
Agrawal et al Algorithm [Agrawal, H. et al., 1995]

Input: successful execution slices, fail execution slices
Output: present the dices to the programmer to debug the program code
Form all the possible dices by subtracting the respective successful execution slices
Compute the number of the statement(s) in the dice, average size
Compute the number of occurrences of the buggy statement in the dice, good dices
Present one of the dices randomly to the programmer to debug the program code
Agrawal et al Proposed Approach II

1. $\chi$-slice [Telecordia, 1998] used to perform slicing and dicing of the program

2. Allows programmer visualize the execution slices and dices of the program

3. **average size** - number of statements (including the bug) for each dice.

4. **good dice** - number of occurrences that the buggy statement appear in the dices
Agrawal et al Proposed Approach III

1. Programmer presented with one of the dices randomly to debug program code
2. Found **good dice** is effective to locate bugs
3. GNU Unix Sort program used in their evaluation
Wong et al. Proposed Approach I  
[Wong, W.E. and Qi, Y., 2004]

1. Proposed an improved dynamic slicing approach – information of data dependencies between blocks of program
2. Agrawal et al. [Agrawal, H. et al., 1995] approach could not locate bugs if executed in both successful and fail execution slices
3. Two iterative approaches; *augmentation approach* and *refining approach*
4. Evaluated Space [Do et al., 2005] programs and performed case studies on several Space program versions
Wong et al. Proposed Approach - Augmentation Approach

1. Used if bug not found in dynamic dice
2. Iteratively include additional block(s) of program in the dynamic dice until bug is found
**Wong et al. Algorithm – Augmentation Approach**

**Input:** successful execution slice, \( E_s \) and fail execution slice, \( E_f \)

**Output:** code segment containing bug(s)

Define Dice \( D: E_f - E_s \);

Set iteration, \( k++ \);

Construct code segment, \( A^k (D \cap E_f - D) \);

if bug is in code segment \( A^k \) then
  Stop;
else
  Repeat if-block;
end

if \( A^k \) is similar to \( A^{k-1} \) then
  No new code can be augmented;
  Reach final augmented code segment;
  No further code segment is needed to be constructed;
  Stop and return \( E_f \) to the programmer to examine the bug;
end
Wong et al. Proposed Approach - Refining Approach

1. Used to remove block(s) of the program which does not contain the bug in the dynamic dice
2. Helped programmer examine lesser program code to locate bugs
3. Debugging Tool Based on Execution Slice and Interblock data Dependency (DESiD) is developed and used
Wong et al. Algorithm – Refining Approach

**Input:** dice $D$, successful and fail execution slice

**Output:** Code segment containing bug(s)

Randomly select $k$ successful execution slices;

Construct new dice $D^k: D \cup (k^{th} \text{ successful execution slices})$;

*if* bug is found in dice $D^k$ *then*

- Stop;

*else*

- Set $k ← −$;
- *if* $k==0$ *then*
  - Stop and examine code in dice $D$;
- *else*
  - Unless bug is found, repeat from the beginning;

*end*
Break?!?

AAGH!

KHAAAAANNNNN!!!

YOU NAMED YOUR THESIS PROJECT “KHAN”?

IT’S EASIER TO VENT. NOW SET DEBUGGERS ON KILL.

WWW.PHDCOMICS.COM
Different statistical methods proposed to locate bugs
1. Proposed to remotely sample predicates from user execution of program code
2. Gathered user execution information in bug report format that feeds to a database
3. Used in software vendors: Mozilla, Microsoft, GNOME and KDE
4. If web browser crashes, a typical bug report is gathered automatically from user and fed into the database
5. Enable programmers to analyse and fix the bug

Liblit et al Proposed Approach [Liblit, B. et al., 2003]
Liblit et al Proposed Approach II

1. Program code instrumented with predicates (branches, return values and scalar pairs)
2. Gather random sampling of predicates of program using geometrically distributed random numbers.
3. Each predicate consists of counter variables that determines next predicate to be sampled
Predicate Remote Sampling Algorithm

**Input:** instrumented program code (predicate-based), predicate countdown variable indicating next predicate to be sampled

**Output:** sampled predicate(s)

**Predicate Sampling:**

```
foreach user execution of the program code do
    if (next sampled predicate countdown > current countdown variable) then
        Fast Path;
        Discard the current countdown variable;
        Get the new next sample predicate countdown;
        Decrement the value of the current countdown variable;
    end

    if (next sampled predicate countdown == current countdown variable) then
        Slow Path;
        if next sampled predicate countdown reached zero then
            Perform sampling on the predicate;
            Reset and retrieve the next sampled predicate countdown;
        end
    end
end
```
Liblit et al Proposed Approach III

1. Evaluated on CCRYPT program using their proposed approach
2. Found the sampled predicates are able to represent the information to help locate bug
Predicates and Its Concepts

S1: if (f == NULL){
S2: x=0;
S3: *f;
S4: }else x=1;

1. Excerpt to show differences of True and reach
2. If predicate on S1 is True, program crashes
3. Predicates on S1 and its negation (S4) is reach if and only if S1 is executed
4. Negation of predicate on S1 is True if and only if S4 is executed
Statistical Bug Isolation Algorithm
[Liblit, B. et al., 2005]

**Input:** instrumented predicates (if-then-else, return values or scalar-pairs scheme), user executions represented as feedback report \( R \)

**Output:** Predicates ranked according to Importance(Pred) based on \( F(Pred) \), Increase (Pred) and CBI Log (Pred)

Perform sparse random sampling approach on instrumented predicates [Liblit, B. et al., 2003] in \( R \);
Gather the predicate-based spectra coverage information for the sampled predicates;
Compute Failure and Context for each instrumented predicate, \( Pred \);
Rank the Importance of all the \( Pred \) based on \( F \), CBI Inc and CBI Log
Extension of Liblit et al Proposed Approach I
[Liblit, B. et al., 2005]

1. Each *Pred* has $F(Pred)$, $S(Pred)$, $F(Pred \text{ observed})$ and $S(Pred \text{ observed})$
2. $F(Pred)$ - number of fail tests that the *Pred* was executed and True
3. $S(Pred)$ - number of pass tests that the *Pred* was executed and True
Extension of Liblit et al Proposed Approach II

1. $F(Pred \text{ observed})$ - number of fail tests where the $Pred$ was observed (reached)
2. $S(Pred \text{ observed})$ - number of pass tests where the $Pred$ was observed (reached)
3. Computed $\text{Failure}(Pred)$ and $\text{Context}(Pred)$ for the program
4. Evaluated on MOSS, CCRYPT, BC, EXIF and RHYTHMBOX datasets [Rhythm, 2010]
5. Found their approach is able to locate bugs in these datasets, but did not compare with Tarantula metric
### Spectra Metrics used in Predicate-based Spectra studies

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure(Pred)</td>
<td>$\frac{F(Pred)}{S(Pred)+F(Pred)}$</td>
</tr>
<tr>
<td>Context(Pred)</td>
<td>$\frac{F(Pred \text{ observed})}{S(Pred \text{ observed})+F(Pred \text{ observed})}$</td>
</tr>
<tr>
<td>CBI Inc (Pred)</td>
<td>Failure(Pred) - Context(Pred)</td>
</tr>
<tr>
<td>CBI Log (Pred)</td>
<td>$2^\left(\frac{1}{CBI \text{ Inc}} + \frac{\log \text{ totF}}{\log F(Pred)}\right)$</td>
</tr>
<tr>
<td>CBI Sqrt (Pred)</td>
<td>$2^\left(\frac{1}{CBI \text{ Inc}} + \frac{\sqrt{\text{totF}}}{\sqrt{F(Pred)}}\right)$</td>
</tr>
<tr>
<td>FPC (Pred)</td>
<td>Failure(Pred) + Context(Pred)</td>
</tr>
<tr>
<td>OFPC</td>
<td>$3 \times F(Pred) + FPC$</td>
</tr>
<tr>
<td>O8FPC</td>
<td>$10 \times F(Pred) + 8 \times \text{Failure(Pred)} + \text{Context}$</td>
</tr>
</tbody>
</table>
Liu et al. Proposed Approach I[Chao Liu et al., 2005]

1. Propose statistical model-based bug localization (better known as SOBER) to locate bugs
2. Use predicate-based spectra coverage; instrumentation is done with respect to branches and return values
3. Use information of execution counts instead of using binary information [Liblit, B. et al., 2005]
4. Capture the number of times a particular predicate, $Pred$ has been executed in pass and fail test cases
Liu et al. Proposed Approach II

1. Evaluated on Siemens and BC1.06 datasets

Within 10% of program code to be examined by the programmer (130 programs in Siemens):

1. SOBER - 68 bugs
2. Liblit et al. - 52 bugs
3. Cleve et al. - 34 bugs
1. SOBER uses more information of predicates than Liblit et al. [Liblit, B. et al., 2005]

2. Considered multiple evaluations of predicates (frequency counts)
Naish et al. Proposed Approach I
[Naish, Lee. et al., 2010]

1. Investigated relationship of statement-based and predicate-based spectra
2. Proposed heuristics to reconstruct predicate-based spectra (if-then-else) based on statement-based spectra
Naish et al. Proposed Approach II

1. Observed improved bug localization performance for predicate-based spectra compared to statement-based spectra.

2. Proposed several metrics based on Failure and Context and outperform the CBI Inc, CBI Log, and CBI Sqrt metrics [Liblit, B. et al., 2005] (Predicate Metrics).

3. Overall 08FPC metric outperforms other metrics in all the empirical benchmarks.
Break?!?

This image is reproduced with the permission from Randall Munroe

xkcd ©
Automated Debugging

1. Debugging – stepping program code (line, blocks, functions) to locate program bugs
2. State-based approaches involves automating debugging; better known as Delta Debugging [Zeller, A., 2000]
3. Isolate test case input that causes program to fail

Zeller proposed to isolate the following:

1. Program input e.g. input of webpage causing the web browser to fail.
2. User interaction e.g. keystrokes by the user causing the program to crash.
3. Changes to the program code e.g. failure inducing code changes in regression testing.
Zeller et al. Isolate Program Input Algorithm
[Zeller, A., 2000]

**Input:** one test case that causes failure to the web browser

**Output:** test case input which last causes web browser failure

**while** test case input can be further simplified **do**

- Program code executed with the simplified test case input;
- **if** the simplified test case input does not cause failure to the web browser **then**
  - Stop and return the previous test case input used as root cause of the web browser failure;
- **else**
  - Simplify the test case input and repeat **while-loop**

**end**

**end**

Return the last test case input as the root cause of the failure of the web browser;
Zeller et al. Isolate Program Input Approach

1. Test case input is simplified one at a time
2. Rerun with the web browser program to check whether the web browser fails (failure still occurs)
3. Returns programmer with the last test case input that causes web browser to fail
4. Evaluated Mozilla and took 21 minutes to isolate failure in web page
5. Similar approach is used to debug failure due to user interaction
Zeller et al. Isolate Failure From Changes to Program Code

1. In regression testing, changes are made in the program code – tend to cause program code to fail

2. Known as *code differences*

Three different options :-

1. **Option 1** - Remove differences of pass test case input from fail test case input.

2. **Option 2** - Add differences of pass test case input from fail test case input to the pass test case input.

3. **Option 3** - Combination of *Option 1* and *Option 2*. 
Zeller et al. Isolate Failure From Changes to Program Code (Option 1 & 2) Algorithm

**Input:** pass test case, fail test case, code differences

**Output:** test case input which last caused (code differences) to fail

while test case input can be further simplified do

   Code differences is executed with the simplified test input;

   if the simplified test case input does not cause any failure to the program then
      Stop;
      Return the previous test case input used as the root cause of program failure;
   end
   else
      Repeat while-loop
   end

end

Test case input could not be further simplified (input is minimum);
Return the last test case input as the root cause of program failure;
Exit;
## Zeller et al. Isolate Failure From Changes to Program Code (Option 3) Algorithm

**Input:** pass test case, fail test case, *code differences*

**Output:** test case input which last caused *(code differences)* to fail

while *test case input can be further simplified* do

- *Code differences* is executed with the pass and fail test case inputs;

  if *both pass and fail test case inputs converge* then

    Return the minimal difference of the last pass and fail test case input as the root cause of program failure;

    Stop;

  end

end

Return the last test case input as the root cause of program failure;

Exit;
Zeller et al. Cause-Effect Chain Approach I
[Zeller, A., 2002]

1. Proposed to narrow down states (variables and values) of the program that causes program failure
Zeller et al. Cause-Effect Chain Algorithm
[Zeller, A., 2002]

**Input:** program code that do not cause program failure $P_1$,
program code version that causes program failure (crash) $P_2$, memory graph of both of these program codes

**Output:** cause-effect chain of failure-induced states that are relevant to the failure

**Isolating relevant failure inducing state:**
Use GDB [GNU GDB, 2010] to extract all the state(s) of the program code $P_1$ and $P_2$;

**Isolate failure-induced states on both of the programs using delta debugging approach** [Zeller, A., 2000];

```plaintext
foreach state do
    Extract the memory graphs of the state of both $P_1$ and $P_2$;
    Compare the vertices and edges from the memory graphs;
    Variables of the state that causes failure as inputs;
    Apply these inputs to $P_1$ using Delta Debugging approach [Zeller, A., 2000];
    Obtain last input (variables of the state) that causes failure to $P_1$;
end
```

Gather all the last input (variables of the states) as part of the cause-effect chain
Zeller et al. Cause-Effect Chain Approach II

1. Evaluated on one of the programs in GNU GCC [GCC GNU, 2010]; fail.c program
2. Does not require prior knowledge of the program code
3. Costly to search for the program code that causes failure
1. Need to navigate each state and compare with the memory graph at the same time

2. Complexity of searching the states is $2^{V \log SD}$ where $V$ – the number of failure inducing variables and $SD$ – the number of state differences

3. Developed framework in a server AskIgor [AskIgor, 2010] and made public
Break?!?

I have found the perfect phrase for condescendingly dismissing anything:

Have you seen the new Ubuntu release?

Nah, I'm not really into Pokémon.
Machine Learning Approaches

1. Several approaches proposed machine learning to improve existing bug localization approaches
2. Information (from test coverage) can be exploited to help locate bugs
Jones et al. Proposed Approach I

[Jones, J.A. et al., 2007]

1. Proposed machine learning technique; clustering to locate bugs in multiple-bug programs
2. Cluster fail test cases that are responsible for particular bug

Two techniques to generate clusters for fail test cases:

1. Clustering based on the profiles and bug localization results, $\text{Cluster}_1$.
2. Clustering based on the bug localization results, $\text{Cluster}_2$. 
Jones et al. Cluster$_1$ Algorithm

**Input:** a set of instrumented statement-based spectra coverage of the fail test cases

**Output:** cluster(s) of fail test cases responsible for the respective bug(s)

Gather sequences of branch profiles (consist of statements) executed in fail test cases in discrete-time Markov chains (DTMCs);

Apply hierarchical clustering algorithm on the DTMC for fail test cases;

```plaintext
foreach level of the dendogram do
    Choose the smallest absolute differences of the branch profiles in each pair of DTMC;
    Treat these fail test cases as one cluster;
    if only one cluster formed for the level then
        Stop;
    end
end
```
1. Discrete-time Markov chains (DTMC) [Romanovskii, V.I., 1970] is used; better known as *behaviour models*

2. From the algorithm, the output is dendogram [Kononenko, Igor and Matjaz Kukar, 2007]
Once cluster(s) formed, bug localization results are used to refine clusters of fail test cases

Evaluate each cluster (consists of respective set of pass and fail test cases)

Ranking of program statements for the cluster is computed using Tarantula metric
Example of Dendogram

This image is reproduced with the permission from Virginia University
[Chemokines Dendrogram, 2013]
Jones et al Cluster\textsubscript{2} Algorithm [Jones, J.A. et al., 2007]

**Input:** fail test cases, *specialised test suite*

**Output:** cluster(s) of fail test cases responsible for respective bug(s)

Evaluate and rank statements for each *specialised test suite* on Tarantula;
Perform pairwise similarity using Jaccard on the ranking statements between the specialised test suites;

1. Each *specialised test suite* contains the statement-based spectra coverage of a fail test case and all the pass test cases.
2. Evaluate proposed approaches on Space program [Do et al., 2005] using rank percentages
3. The average rank percentages evaluated using Tarantula metric on $Cluster_1$ - 31.50\% and $Cluster_2$ - 26.43\%
Briand et al. Proposed Approach I
[Briand, L.C. et al., 2007]

1. Propose category partition method to transform test cases from the test specifications [Ostrand, T.J. and Balcer, M.J., 1988]

2. Example of transformed test case: size of an array; whether array is empty

3. Use C4.5 algorithm [Quinlan, J.R., 1993] to generate rules (forming decision trees) or better known as RUle-BAshed statement Ranking (RUBAR)
Each rule – a set of statements of program; classified as Pass/Fail

A rule is **Fail** - majority of Fail tests execute the set of statements of the rule

A rule is **Pass** - majority of Pass tests execute the set of statements of the rule

Positive weights - **Pass** rules; Negative weights - **Fail** rules

Tarantula - 15% of bugs; proposed **RUBAR** approach - 25% of the bugs in Space [Do et al., 2005]
Briand et al. [Briand, L.C. et al., 2007] Algorithm

- **Input:** test specifications, pass and fail test cases, program code
- **Output:** program statements with respective weights assigned

Use Category-Partition method to generate transformed test cases from the test specifications;

**Classification;**
Generate rules from the transformed test cases using C4.5 algorithm;
Classify rules to Pass and Fail rules;

**Compute weights for Pass and Fail rules;**

```plaintext
foreach rule do
    if Pass rule then
        Statements related to the rule assigned positive weight;
    end
    else
        Statements related to the rule assigned negative weight;
    end
end
```

Weights aggregated for statement and presented to the programmer;
Dickinson et al. Proposed Approach I
[Dickinson, W. et al., 2001]

1. Propose clustering test cases based on similarity of test coverage using the hierarchical clustering algorithm [Murtagh, F., 1983]

2. Apply several similarity measures to form test case clusters; e.g: Euclidean distance

3. Minimal dissimilarity threshold is set to limit number of clusters
Dickinson et al. Algorithm [Dickinson, W. et al., 2001]

**Input**: Instrumented function caller/callee of the fault-seeded program code version, test cases

**Output**: clustered test cases

Execute test cases on the fault-seeded program code to gather function caller and callee of the execution coverage;

**Using hierarchical clustering algorithm** [Murtagh, F., 1983];

foreach *iteration or level of dendogram* do

- Apply similarity measures on the function caller and callee execution coverage of test cases;
- Similar execution coverage of test cases fulfills minimal dissimilarity threshold forms a cluster;

end
Sampling strategies are used to find failures; *one-per-cluster sampling*, *adaptive sampling*, and *random sampling*

1. *one-per-cluster sampling* - a test case is selected for each cluster

2. *adaptive sampling* - similar to *one-per-cluster sampling*, in addition to the selected test case is used to determine if causes program failure
Dickinson et al. Proposed Approach III

1. Percentages of failure in the clusters is evaluated where percentage of fail test cases found in each cluster

2. Evaluate on several Java programs and GNU GCC version 2.95.2 and observe bugs are not distributed in random fashion

In the smallest cluster for Java and GCC programs,

1. *adaptive sampling* – 40.26% and 5.55% of the failures

2. *one-per-cluster sampling* – 4.30% and 1.12% of the failures
Fatta el al. Proposed Approach I
[Di Fatta, G. et al., 2006]


2. Instrument program to gather function-based spectra coverage

3. Pass and fail test cases are represented in function call tree; set of functions (represented by vertices and edges) that are executed in the test cases.
**Fatta et al [Di Fatta, G. et al., 2006] Algorithm**

**Input:** function-based spectra coverage represented in function call trees

**Output:** functions ranked according to how likely they are buggy

**Abstraction phase;**
Function call trees are analysed using *zero-one-many* abstraction;

**Filtering phase;**
Define neighbourhood size $N$;
Use Frequent Pattern Mining algorithm to extract discriminative patterns according to neighbourhood size $N$;
Identify functions in the subtree that are executed more frequently in fail test cases than pass test cases;

**Analysis phase;**
Apply ranking function, $f$ of the neighbourhood size $N$ using

$$P(f) = \frac{\text{support of } f \text{ in all fail test cases}}{\text{support of } f \text{ in all test cases}}$$
Fatta el al. Proposed Approach II

1. *zero-one-many* abstraction proposed to reduce the number of function call trees (reduce performance and memory overhead)

2. Frequent pattern mining algorithm used to identify subtrees that are frequently executed in the pass and fail test cases – discriminative patterns
Evaluate on Siemens Test Suite and outperform other approaches such as the nearest neighbour [Renieres et al., 2003], cause-transition [Zeller, A., 2002], and SOBER [Chao Liu et al., 2005] approaches.

Able to locate 22% of the buggy functions in the Siemens Test Suite with the frequent patterns of neighbourhood size 2.
Jiang et al. Proposed Approach I
[Jiang, L. and Su, Z., 2005]

2. Use linear and radial basis functions [Walczak, B. and et al., 1996] to determine hyperplanes of SVM
3. Combine with the cause-effect chain approach [Zeller, A., 2002] to isolate predicates of a program that cause the bug
4. CBI system [Liblit, B. et al., 2005] is used for instrumenting predicate of the program
Jiang et al. Algorithm I [Jiang, L. and Su, Z., 2005]

**Input**: predicate-based instrumented program, predicate-based spectra coverage of the program  
**Output**: predicates in clusters  
**Classification**;
Apply predicate-based spectra coverage as the input for SVM algorithm [Steinwart, I. and Christmann, A., 2008];
Assign score to the predicates of the program using random forest [Breiman, L., 2001];
Perform feature selection to choose top predicates that are most likely the bug;
**Clustering**;
foreach pair of predicates chosen in feature selection do  
  Gather the differences of the distribution of pairs of predicates in all test execution coverage [Chao Liu et al., 2005];
  if differences of distribution of the predicates in the test execution coverage \( < \epsilon \) then  
    Predicates belong to the similar cluster;
end  
end
Jiang et al. Algorithm II

**Input:** predicates in clusters

**Output:** chain of predicates that are likely to be buggy

**Cause-effect chain:**

```plaintext
foreach cluster do
    if any predicates in the cluster related to the bug then
        Use Control Flow graph (CFG) to find paths that are related to the predicate;
        Form a chain of predicates;
    end
end

Present the chain of predicates for all the clusters to the programmer;
```
Jiang et al. Proposed Approach II

1. Siemens Test Suite - 74 bugs within examining 10% of the program code
2. Rhythmbox v0.6.4 dataset - 5 bugs after examining 1000 lines out of 56 484 lines of code
3. Causes performance overhead on larger size programs such as the Rhythmbox dataset
Proposed Approach of Zheng et al. I
[Zheng, A.X. et al., 2006]

1. Proposed classification and feature selection approaches to narrow down predicates that are buggy [Zheng, A.X. et al., 2003]

2. In this study, proposed to use clustering approach – a bi-clustering scheme to cluster predicates of program responsible for a bug

3. Voting is proposed to determine particular predicate $Pred$ to be the bug in each cluster using $Q_{Pred}$ (quality of predicate)
Proposed Approach of Zheng et al. II

1. Vote of predicate, $Pred$ in each cluster is aggregated across all test cases.
2. Predicate $Pred$ with highest total votes in each cluster identified as one of the bugs of program.
3. Evaluate proposed approach on Siemens Test Suite using PDG performance measure.
   [Renieries et al., 2003, Cleve et al., 2005]
4. Examine more than 7% of the program code – found additional 70 bugs using SOBER; 65 bugs using CBI.
   [Liblit, B. et al., 2003]
Zheng et al. Algorithm [Zheng, A.X. et al., 2006]

**Input**: predicate-based spectra coverage (frequency counts) of sampled predicates

**Output**: predicate with the highest votes for each cluster

\[ Q_{\text{Pred}} \] as quality of each predicate, \( \text{Pred} \)

**Inferring Truth Probabilities**;
Sampled predicates are inferred with truth probabilities using graphical model;
Generate clusters of predicates using spectral clustering algorithm [Ng, A. et al., 2001];

**Collective Voting approach using bi-clustering scheme**;

foreach cluster of predicates formed do
  foreach predicate, \( \text{Pred} \), in the cluster do
    foreach test case do
      Compute the \( Q_{\text{Pred}} \);
      Contribution of predicate, \( \text{Pred} \) to respective test case is computed;
      Aggregate vote assigned by the test case for predicate, \( \text{Pred} \);
    end
  end
Predicate, \( \text{Pred} \) with the highest number of total votes is chosen as the most likely bug in the cluster;
end
Example of Practical Bug Localization Tools

1. CBI system [Liblit, B. et al., 2005]
2. HOLMES [Chilimbi et al., 2009]
Liblit et al. Proposed Public Deployment of CBI Approach [Liblit et al., 2004]

1. Distributed binaries for several large open source projects; e.g. Evolution, Gaim, Gimp, Gnumeric, Nautilus and Rhythmbox
2. Gathered information of program when program crashes
3. Reports sent back to CBI server for further analysis of the root cause of the program failures
4. Observed most crashes occurred in Rhythmbox application
HOLMES system [Chilimbi et al., 2009]

1. Chilimbi et al. proposed using path-based spectra coverage and two debugging approaches
   - 1. Non-adaptive debugging
   - 2. Adaptive debugging

2. Used several real-world applications (e.g. GCC, Apache and EDG compiler)

3. Used Microsoft Phoenix compiler framework for instrumentation of program branches and paths
Chilimbi et al. Proposed Approach – Non-Adaptive debugging

1. Uses similar framework as CBI system [Liblit, B. et al., 2005]
2. Gather user execution information (program paths), in the form of reports $R$, using sparse random sampling
3. Adapt the *Failure* and *Increase* metrics [Liblit, B. et al., 2005] to rank paths instead of predicates
4. Causes instrumentation overhead
5. Similar to Liblit et al. [Liblit et al., 2005] where require to maintain path counters and paths sampling
Chilimbi et al. Non-Adaptive Debugging Algorithm

**Input:** instrumented program code (path-based) *Path*, path-based spectra coverage

**Output:** paths ranked in *Importance* of *Path* (likely to be buggy) *Failure* and *Context* are computed for each instrumented path of the program code, *Path*;

Rank the *Importance* based on *Failure, CBI Inc, and CBI Log* of all the *Path*;
Chilimbi et al. Adaptive Debugging Algorithm

**Input:** program code, user executions information of program

**Output:** Paths likely to be bug returned to the programmer

Monitor each program execution for failures and gather reports;
Perform static analysis based on the reports;
Compute set of functions appear in fail stack traces;
Instrumented program code especially on the selected set of functions;
Monitor several program execution on the set of functions for failures;
Use static analysis component model on the buggy functions;
Narrow down the functions using *Importance*;

while all failures not explainable to the programmer do
    Invoke static analysis component to identify other fragments of code;
end

Bug is found and stop;
Chilimbi et al. Proposed Approach I – Adaptive debugging

1. The approach did not compromise instrumentation overhead
2. Evaluate on 6 out of 7 programs of the Siemens Test Suite [Do et al., 2005] using the Microsoft Phoenix compiler [Hall, M. et al., 2009]
3. Evaluate other programs such as GCC, Translate, and Edg
Chilimbi et al. Proposed Approach II – Adaptive debugging

1. Evaluate approach on different spectra such as predicate, path and branch of the programs. By examining 10% of the program nodes in the Siemens Test Suite,
   - path-based spectra - 24 bugs
   - predicate-based spectra - 14 bugs
   - branch-based spectra - 0 bug

2. Concluded path-based spectra coverage performed best in locating bugs
Recent Bug Localization Research Trend I

1. Bug localization approaches involves fiddling on metrics and no theoretical support
   [XiaoYuan Xie et al., 2010, Wong, W. Eric et al., 2012]

2. More research studies combining existing (discussed) approaches
   1. Slicing and statistical bug localization approaches
      [Lei, Yan, et al., 2012, Lei, Yan, et al., 2012]
   2. Statistical bug localization and mining source code history
      [Servant, Francisco, and James A Jones., 2012]
Recent study investigated the relationship of different bugs (fault classes) with bug localization effectiveness [Bandyopadhyay, Aritra, and Sudipto Ghosh, 2011]

Attempt to use more information from test coverage; using frequency counts [HuaJie, L. et al., 2010]

Give weights to test cases based on the proximity among the test cases [Bandyopadhyay, Aritra, 2011]
Using Test Coverage Information Approach

1. Test coverage information is used to help narrow down the search of bugs
2. Uses mainly binary counts of test coverage information
3. Several studies proposed different spectra metrics; shown improved bug localization performance
Take Home Message II

Using Slicing Approach

1. Different dimensions of using test coverage information
2. Study relationships between test cases (pass and fail)
3. An advantage of providing more information of bugs in the program e.g related blocks or statements of program which could be executed by a pass test case but not by a fail test case
Take Home Message III

Using Statistical Approach

1. Test coverage information is combined with statistical approaches e.g. sampling predicates of program

2. Statistical methods such as random sampling, standard deviation, mean and probability model were used
Take Home Message IV

Using State-based Approach

1. Automate the traditional debugging approach to isolate the last circumstance or trails of inputs that lead to causing program failure
2. Improve efficiency of debugging large program code rather having programmer to step-in the program code
Using Machine Learning Approach

1. Exploit extra information from test coverage information
2. Learn these information and predict likelihood of bugs in the program
3. More computation required but shown improved bug localization performance overall compared to other discussed approaches
Challenges

- Explore other coverage types than statement-based spectra [Naish et al., 2009a], block-based spectra [Abreu et al., 2006], predicate-based spectra [Liblit et al., 2005] or path-based spectra [Chilimbi et al., 2009]
- Apply optimal metrics ($O$ and $O^p$) and other spectra metrics into existing debugging system such as CBI [Liblit et al., 2005] and HOLMES [Chilimbi et al., 2009].
- Establish theoretical approach for optimal metrics in multiple-bug programs
- Gather more comprehensive datasets; help serve as standard de facto benchmark to compare different bug localization studies; e.g Siemens Test Suite
Question and Answer
Some Insights ...

The Lab/Office Whiteboard

- Do not erase! Left here by previous generation of grad students.
- Not really white anymore.
- List of supplies to buy: why buy it yourself when you can just write it here and hope it magically appears.
- To buy:
  - Pens
  - Tools
  - Rubber gloves
  - Whiteboard markers

- Whiteboard productivity:
  - "A beautiful mind"
  - Quid pro quo
  - Hey, looks like a party
  - Chaos!

- People in front of whiteboard

- Marker roulette:
  - Only marker that works
  - Have wrong color cap on
  - Actually a permanent marker #&*!!!

- In case of emergency
  - Magnets + menus from late night takeout places

Jorge Cham © 2008

www.phdcomics.com
Thank You for your attention.
## Appendix of Spectra Metrics I

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard</td>
<td>( \frac{a_{ef}}{totF + a_{ep}} )</td>
<td>Ochiai</td>
<td>( \frac{a_{ef}}{\sqrt{totF(a_{ef} + a_{ep})}} )</td>
</tr>
<tr>
<td>Tarantula</td>
<td>( \frac{a_{ef}}{totF} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \frac{a_{ef}}{totF + a_{ep}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoltar</td>
<td>( \frac{a_{ef}}{totF + a_{ep} + \frac{10000 a_{nf} a_{ep}}{a_{ef}}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ample</td>
<td>( \frac{a_{ef}}{totF} - \frac{a_{ep}}{totP} )</td>
<td>Ample2</td>
<td>( \frac{a_{ef}}{totF} - \frac{a_{ep}}{totP} )</td>
</tr>
<tr>
<td>Ample2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ample3</td>
<td>( a_{ef} - h, \text{ where } h = \begin{cases} a_{ep} &amp; \text{if } a_{ep} \leq 2 \ 2 + 0.1(a_{ep} - 2) &amp; \text{if } 2 &lt; a_{ep} \leq 10 \ 2.8 + 0.001(a_{ep} - 10) &amp; \text{if } a_{ep} &gt; 10 \end{cases} )</td>
<td>Ample2</td>
<td>( a_{ef} - a_{ep} )</td>
</tr>
<tr>
<td>Ample3'</td>
<td>( a_{ef} - h, \text{ where } h = \begin{cases} -1000 &amp; \text{if } a_{ep} + a_{ef} = 0 \ Wong3 &amp; \text{otherwise} \end{cases} )</td>
<td>Ample2</td>
<td></td>
</tr>
<tr>
<td>CBI Inc</td>
<td>( \frac{a_{ef}}{a_{ef} + a_{ep}} - \frac{totF}{T} )</td>
<td>CBI Log</td>
<td>( \frac{1}{CBI \text{ Inc} + \log totF} + \frac{2}{\log a_{ef}} )</td>
</tr>
<tr>
<td>CBI Sqrt</td>
<td>( \frac{2}{CBI \text{ Inc} + \frac{\sqrt{totF}}{\sqrt{a_{ef}}}} )</td>
<td>M2</td>
<td>( \frac{a_{ef}}{a_{ef} + a_{np} + 2(a_{nf} + a_{ep})} )</td>
</tr>
</tbody>
</table>
Wong4 metric

Definition (Ranking metric Wong4)

Wong4 \[ (1.0) \times n_{F,1} + (0.1) \times n_{F,2} + (0.01) \times n_{F,3} \] - \[ (1.0) \times n_{S,1} + (0.1) \times n_{S,2} + 0.0001 \times \frac{(a_{ef} + a_{nf})}{(a_{ep} + a_{np})} \times n_{S,3} \]

where

\[ n_{F,1} = \begin{cases} 0, & \text{for } a_{ef} = 0 \\ 1, & \text{for } a_{ef} = 1 \\ 2, & \text{for } a_{ef} \geq 2 \end{cases} \]

\[ n_{F,2} = \begin{cases} 0, & \text{for } a_{ef} \leq 2 \\ a_{ef} - 2, & \text{for } 3 \leq a_{ef} \leq 6 \\ 4, & \text{for } a_{ef} > 6 \end{cases} \]

\[ n_{F,3} = \begin{cases} 0, & \text{for } a_{ef} \leq 6 \\ a_{ef} - 6, & \text{for } a_{ef} > 6 \end{cases} \]

\[ n_{S,1} = \begin{cases} 0, & \text{for } n_{F,1} = 0,1 \\ 1, & \text{for } n_{F,1} = 2 \text{ and } a_{ep} \geq 1 \\ 0, & \text{for } a_{ep} \leq n_{S,1} \end{cases} \]

\[ n_{S,2} = \begin{cases} a_{ep} - n_{S,1}, & \text{for } n_{S,1} < a_{ep} < n_{F,2} + n_{S,1} \\ n_{F,2}, & \text{for } a_{ep} \geq n_{F,2} + n_{S,1} \end{cases} \]

\[ n_{S,3} = \begin{cases} 0, & \text{for } a_{ep} < n_{S,1} + n_{S,2} \\ a_{ep} - n_{S,1} - n_{S,2}, & \text{for } a_{ep} \geq n_{S,1} + n_{S,2} \end{cases} \]
### Appendix of Spectra Metrics II

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>( a_{ef} + a_{np} )</td>
<td>M3</td>
<td>( \frac{2 \times (a_{ef} + a_{np})}{\text{totF}} )</td>
</tr>
<tr>
<td>Sørensen-Dice</td>
<td>( \frac{2a_{ef}}{2a_{ef} + a_{nf} + a_{ep}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kulczynski1</td>
<td>( \frac{a_{ef}}{a_{nf} + a_{ep}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>( \frac{1}{2} \left( \frac{a_{ef}}{\text{totF}} + \frac{a_{ef}}{a_{ef} + a_{ep}} \right) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Russell and Rao</td>
<td>( \frac{a_{ef}}{a_{ef} + a_{np}} )</td>
<td>Hamann</td>
<td>( \frac{a_{ef} + a_{np} - a_{nf} - a_{ep}}{\text{totF}} )</td>
</tr>
<tr>
<td>Simple Matching</td>
<td>( \frac{a_{ef} + a_{np}}{\text{totF}} )</td>
<td>Lee</td>
<td>( a_{ef} + a_{np} )</td>
</tr>
<tr>
<td>Rogers &amp; Tanimoto</td>
<td>( \frac{a_{ef} + a_{np}}{a_{ef} + a_{np} + 2(a_{nf} + a_{ep})} )</td>
<td>Goodman</td>
<td>( \frac{2a_{ef} - a_{nf} - a_{ep}}{2a_{ef} + a_{nf} + a_{ep}} )</td>
</tr>
<tr>
<td>Hamming</td>
<td>( a_{ef} + a_{np} )</td>
<td>Euclid</td>
<td>( \sqrt{a_{ef} + a_{np}} )</td>
</tr>
<tr>
<td>Ochiai2</td>
<td>( \frac{a_{ef}a_{np}}{\sqrt{(a_{ef} + a_{ep})(a_{np} + a_{nf})(\text{totF})(\text{totP})}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ochiai3</td>
<td>( \frac{a_{ef}^2}{(a_{ef} + a_{nf})(a_{ef} + a_{ep})} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Platetsky-Shapiro</td>
<td>( a_{ef} + a_{np}^2 + \text{totF} - a_{ep}a_{ef} - a_{ep}a_{nf} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collective Strength</td>
<td>( 1 - \frac{a_{ef} + a_{np}}{(a_{ef} + a_{ep})(\text{totF}) + (a_{nf} + a_{np})(\text{totP})} ) * ( \frac{1 - (a_{ef} + a_{np})(\text{totF}) - (a_{nf} + a_{np})(\text{totP})}{1 - a_{ef} - a_{np}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometric Mean</td>
<td>( \frac{a_{ef}a_{np} - a_{nf}a_{ep}}{\sqrt{(a_{ef} + a_{np})(a_{np} + a_{nf})(\text{totF})(\text{totP})}} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Harmonic Mean</td>
<td>( \frac{(a_{ef}a_{np} - a_{nf}a_{ep})(a_{ef} + a_{ep})(a_{np} + a_{nf}) + (\text{totF})(\text{totP})}{(a_{ef} + a_{np})(a_{np} + a_{nf})(\text{totF})(\text{totP})} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Arithmetic Mean</td>
<td>( \frac{2a_{ef}a_{np} - 2a_{nf}a_{ep}}{(a_{ef} + a_{np})(a_{np} + a_{nf}) + (\text{totF})(\text{totP})} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Appendix of Spectra Metrics III

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cohen</td>
<td>$\frac{2a_{ef}a_{np}-2a_{nf}a_{ep}}{(a_{ef}+a_{ep})(\text{totP})+(\text{totF})(a_{nf}+a_{np})}$</td>
</tr>
<tr>
<td>Scott</td>
<td>$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})(2a_{np}+a_{nf}+a_{ep})}$</td>
</tr>
<tr>
<td>Fleiss</td>
<td>$\frac{4a_{ef}a_{np}-4a_{nf}a_{ep}-(a_{nf}-a_{ep})^2}{(2a_{ef}+a_{nf}+a_{ep})+(2a_{np}+a_{nf}+a_{ep})}$</td>
</tr>
<tr>
<td>Rogot1</td>
<td>$\frac{1}{2} \left( \frac{a_{ef}}{2a_{ef}+a_{nf}+a_{ep}} + \frac{a_{np}}{2a_{np}+a_{nf}+a_{ep}} \right)$</td>
</tr>
<tr>
<td>Rogot2</td>
<td>$\frac{1}{4} \left( \frac{a_{ef}}{a_{ef}+a_{ep}} + \frac{a_{ef}}{\text{totF}} + \frac{a_{np}}{\text{totP}} + \frac{a_{np}}{a_{np}+a_{nf}} \right)$</td>
</tr>
<tr>
<td>Binary</td>
<td>$0 \text{ if } a_{nf} &gt; 0, \text{ otherwise } 1$</td>
</tr>
<tr>
<td>Gower1</td>
<td>$\frac{a_{ef}-(a_{nf}+a_{ep})+a_{np}}{t}$</td>
</tr>
<tr>
<td>Gower2</td>
<td>$\frac{a_{ef}+a_{np}}{a_{ef}+a_{np}+0.5 \ast (a_{nf}+a_{ep})}$</td>
</tr>
<tr>
<td>Anderberg</td>
<td>$\frac{a_{ef}}{a_{ef}+2(a_{nf}+a_{ep})}$</td>
</tr>
<tr>
<td>Added Value</td>
<td>$\frac{a_{ef}}{\max(a_{ef}+a_{np}, \text{totF})}$</td>
</tr>
<tr>
<td>Confidence</td>
<td>$\max\left(\frac{a_{ef}}{a_{ef}+a_{np}}, \frac{a_{ef}}{\text{totF}}\right)$</td>
</tr>
</tbody>
</table>
## Appendix of Spectra Metrics IV

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certainty</td>
<td>[ \max \left( \frac{a_{ef}}{a_{ef} + a_{ep}} - (a_{ef} + a_{ep}), 1 - (a_{ef} + a_{ep}) \right) ]</td>
<td>Sneath &amp; Sokal1</td>
<td>[ \frac{a_{ef}}{2(a_{ef} + a_{np})} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sneath &amp; Sokal2</td>
<td>[ \frac{a_{ef}}{a_{ef} + 2(a_{nf} + a_{ep})} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phi</td>
<td>[ \frac{a_{ef} a_{np} - a_{nf} a_{ep}}{\sqrt{(totF)(a_{ef} + a_{np})(a_{nf} + a_{np}) (totP)}} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kappa</td>
<td>[ 1 - \frac{a_{ef} + a_{np} - (a_{ef} + a_{ep})(totF) - (a_{nf} + a_{np})(totP)}{1 - (a_{ef} + a_{ep})(totF) - (a_{nf} + a_{np})(totP)} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conviction</td>
<td>[ \max \left( \frac{(a_{ef} + a_{np})(totP)}{totF}, \frac{(a_{nf} + a_{np})}{totF} \right) ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mountford</td>
<td>[ \frac{2a_{ef}}{2(a_{ef} + a_{np})(totF) - (2a_{ef} + a_{np} + a_{nf})a_{ef}} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Klosgen</td>
<td>[ \sqrt{a_{ef} \times \max \left( \frac{a_{ef}}{a_{ef} + a_{ep}} - \frac{totF}{a_{ef} + a_{ep}}, \frac{a_{ef}}{totF} - (a_{ep} + a_{ef}) \right)} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YuleQ</td>
<td>[ \sqrt{a_{ef} a_{np} - a_{ep} a_{nf}} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YuleY</td>
<td>[ \sqrt{a_{ep} a_{np} - \sqrt{a_{ep} a_{np}} + a_{ep} a_{nf}} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>YuleV</td>
<td>[ \sqrt{a_{ef} a_{np} - (a_{ef} + a_{ep})(totF)} ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>J-Measure</td>
<td>[ a_{ef} \times \log\left( \frac{a_{ef}}{a_{ef} + a_{ep}} \right) + \max \left( a_{ep} \times \log\left( \frac{a_{ep}}{a_{ef} + a_{ep}} \right), a_{nf} \times \log\left( \frac{a_{nf}}{totF * (a_{nf} + a_{np})} \right) \right) ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Correlation</td>
<td>[ \frac{</td>
</tr>
</tbody>
</table>
## Appendix of Spectra Metrics V

<table>
<thead>
<tr>
<th>Name</th>
<th>Formula</th>
<th>Name</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manhattan</td>
<td>$1 - \frac{a_{nf} + a_{ep}}{T}$</td>
<td>Braun</td>
<td>$\frac{a_{ef}}{a_{ef} + a_{ep}}$</td>
</tr>
<tr>
<td>Baroni</td>
<td>$\frac{\sqrt{a_{ef}a_{np} + a_{ef}}}{\sqrt{a_{ef}a_{np} + toF + a_{ep}}}$</td>
<td>Coef</td>
<td>$\frac{a_{ef}}{a_{ef} + a_{ep}}$</td>
</tr>
<tr>
<td>Levandowsky</td>
<td>$1 - \frac{(a_{nf} + a_{ep})}{toF + a_{ep}}$</td>
<td>Watson</td>
<td>$1 - \frac{(a_{nf} + a_{ep})}{2a_{ef} + a_{nf} + a_{ep}}$</td>
</tr>
<tr>
<td>Jaccube</td>
<td>$\frac{a_{ef}}{\sqrt{3 \times toF + a_{ep}}}$</td>
<td>NFD</td>
<td>$a_{ef} + a_{np}$</td>
</tr>
<tr>
<td>SokalDist</td>
<td>$\sqrt{\frac{a_{ef} + a_{np}}{T}}$</td>
<td>Overlap</td>
<td>$\frac{a_{ef}}{\min(a_{ef}, a_{nf}, a_{ep})}$</td>
</tr>
<tr>
<td>CorRatio</td>
<td>$\frac{a_{ef}^2}{toF(a_{ef} + a_{ep})}$</td>
<td>Forbes</td>
<td>$\frac{(T + a_{ef})}{(a_{ef} + a_{ep})(toF)}$</td>
</tr>
<tr>
<td>Fager</td>
<td>$\frac{a_{ef} \sqrt{2T}}{\sqrt{toF(a_{ef} + a_{ep})}} - \frac{1}{2*\sqrt{a_{ef} + a_{ep}}}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simpson</td>
<td>$\frac{a_{ef}}{toF}$</td>
<td>McCon</td>
<td>$\frac{a_{ef}^2 - a_{nf}a_{ep}}{(toF)(a_{ef} + a_{ep})}$</td>
</tr>
<tr>
<td>AssocDice</td>
<td>$\frac{a_{ef}}{\min((a_{ef} + a_{ep})(toF))}$</td>
<td>Dice</td>
<td>$\frac{2a_{ef}}{toF + a_{ep}}$</td>
</tr>
<tr>
<td>Fossum</td>
<td>$\frac{T(a_{ef} - 0.5)^2}{(toF)(a_{ef} + a_{ep})}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pearson</td>
<td>$\frac{a_{ef}a_{np} - a_{nf}a_{ep}}{\sqrt{(toF)(a_{ef} + a_{ep})(a_{nf} + a_{np})(toP)}}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dennis</td>
<td>$\frac{a_{ef}a_{np} - (a_{nf}a_{ep})}{\sqrt{(T)(toF)(a_{ef} + a_{ep})}}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Surface of Respective Spectra Metrics

Figure: Surface for $O^P$ metric with respect to MetValue, $a_{ef}$ and $a_{ep}$
Surface for Rogers Metric

**Figure:** Surface for Rogers metric
Surface for Wong3 Metric

Surface for Wong3 metric with respect to MetValue, $a_{ef}$ and $a_{ep}$

Figure: Surface for Wong3 metric
Surface for Wong4 Metric

Figure: Surface for Wong4 metric
Surface for Zoltar Metric

Figure: Surface for Zoltar metric
Surface for Tarantula Metric

Surface for Tarantula metric with respect to MetValue, $a_{ef}$ and $a_{ep}$

**Figure:** Surface for Tarantula metric
Surface for Jaccard Metric

Figure: Surface for Jaccard metric
Surface for Ochiai Metric

Surface for Ochiai metric with respect to MetValue, $a_{ef}$ and $a_{ep}$

Figure: Surface for Ochiai metric
### Evaluation Results on Multiple-bug Programs (Siemens (STS), Unix and Space test suite)

Average Rank Percentages (%) for Multiple-bug Siemens, Unix and Space programs

<table>
<thead>
<tr>
<th>Metric</th>
<th>STSUnix(2 bug)</th>
<th>STSUnix(3 bug)</th>
<th>Space</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kulczynski2</td>
<td>19.53</td>
<td>21.94</td>
<td>2.61</td>
<td>21.44</td>
</tr>
<tr>
<td>Ochiai</td>
<td>20.18</td>
<td>22.60</td>
<td>2.65</td>
<td>22.09</td>
</tr>
<tr>
<td>Wong4</td>
<td>20.78</td>
<td>22.60</td>
<td>2.50</td>
<td>22.21</td>
</tr>
<tr>
<td>Jaccard</td>
<td>21.10</td>
<td>23.12</td>
<td>3.25</td>
<td>22.69</td>
</tr>
<tr>
<td>Zoltar</td>
<td>20.52</td>
<td>23.49</td>
<td>2.32</td>
<td>22.87</td>
</tr>
<tr>
<td>CBI Log</td>
<td>22.26</td>
<td>23.37</td>
<td>4.41</td>
<td>23.12</td>
</tr>
<tr>
<td>Ample</td>
<td>24.49</td>
<td>23.88</td>
<td>8.13</td>
<td>23.97</td>
</tr>
<tr>
<td>Wong3</td>
<td>22.54</td>
<td>25.24</td>
<td>2.50</td>
<td>24.67</td>
</tr>
<tr>
<td>O$^p$</td>
<td>22.84</td>
<td>25.33</td>
<td>2.50</td>
<td>24.80</td>
</tr>
<tr>
<td>Tarantula</td>
<td>23.13</td>
<td>27.23</td>
<td>4.51</td>
<td>26.39</td>
</tr>
<tr>
<td>O</td>
<td>24.95</td>
<td>29.29</td>
<td>2.50</td>
<td>28.40</td>
</tr>
<tr>
<td>Russell</td>
<td>32.10</td>
<td>28.67</td>
<td>17.50</td>
<td>29.31</td>
</tr>
<tr>
<td>Binary</td>
<td>34.02</td>
<td>32.43</td>
<td>17.50</td>
<td>32.71</td>
</tr>
</tbody>
</table>
Breakdown of Average Rank Percentages for Two-bug Programs

Figure: Breakdown of Average Rank Percentages for Two-bug Programs on Kulczynski2
Breakdown of Average Rank Percentages for Three-bug Programs

Figure: Breakdown of Average Rank Percentages for Three-bug Programs on Kulczynski2
## Appendix of Incremental Approach on other Multiple-bug programs

Experimental Results: Results For Three-Bug Programs Siemens and Unix(Average rank percentages)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unweighted</th>
<th>Weighted</th>
<th>10% Inc.</th>
<th>20% Inc.</th>
<th>Incre.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kulczynski2</td>
<td>21.94</td>
<td>21.61</td>
<td>17.59</td>
<td>17.57</td>
<td>17.54</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Ochiai</td>
<td>22.60</td>
<td>22.25</td>
<td>18.22</td>
<td>18.20</td>
<td>18.19</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Wong4</td>
<td>22.60</td>
<td>21.62</td>
<td>18.43</td>
<td>18.17</td>
<td>17.97</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Jaccard</td>
<td>23.12</td>
<td>22.69</td>
<td>18.63</td>
<td>18.54</td>
<td>18.62</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Zoltar</td>
<td>23.49</td>
<td>23.44</td>
<td>19.32</td>
<td>19.40</td>
<td>19.35</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Ample</td>
<td>23.88</td>
<td>23.54</td>
<td>19.60</td>
<td>19.60</td>
<td>19.52</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Wong3</td>
<td>25.24</td>
<td>25.18</td>
<td>21.04</td>
<td>20.96</td>
<td>21.05</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>O$^p$</td>
<td>25.33</td>
<td>25.25</td>
<td>21.05</td>
<td>21.02</td>
<td>21.08</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Tarantula</td>
<td>27.23</td>
<td>26.26</td>
<td>20.58</td>
<td>20.48</td>
<td>20.44</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>O</td>
<td>29.29</td>
<td>28.90</td>
<td>22.84</td>
<td>22.74</td>
<td>22.66</td>
<td>&lt;0.05</td>
</tr>
</tbody>
</table>

End of Appendix of Incremental Approach on other Multiple-bug programs
Appendix of Incremental Approach on other Multiple-bug programs

Experimental Results: Results For Multiple-bug Space Programs (Average rank percentages)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unweighted</th>
<th>Weighted</th>
<th>10% Inc.</th>
<th>20% Inc.</th>
<th>Incre.</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoltar</td>
<td>2.42</td>
<td>2.42</td>
<td>2.04</td>
<td>2.05</td>
<td>2.05</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>O</td>
<td>2.55</td>
<td>2.55</td>
<td>2.16</td>
<td>2.16</td>
<td>2.17</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>$O^p$</td>
<td>2.55</td>
<td>2.55</td>
<td>2.17</td>
<td>2.18</td>
<td>2.18</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Wong3</td>
<td>2.55</td>
<td>2.55</td>
<td>2.18</td>
<td>2.17</td>
<td>2.17</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Kulczynski2</td>
<td>2.63</td>
<td>2.61</td>
<td>2.24</td>
<td>2.25</td>
<td>2.25</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Ochiai</td>
<td>2.77</td>
<td>2.76</td>
<td>2.39</td>
<td>2.39</td>
<td>2.39</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Wong4</td>
<td>2.80</td>
<td>2.75</td>
<td>2.36</td>
<td>2.38</td>
<td>2.37</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Jaccard</td>
<td>3.25</td>
<td>3.23</td>
<td>2.83</td>
<td>2.85</td>
<td>2.85</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Tarantula</td>
<td>4.36</td>
<td>4.31</td>
<td>3.94</td>
<td>3.94</td>
<td>3.95</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Ample</td>
<td>8.39</td>
<td>4.48</td>
<td>4.09</td>
<td>4.09</td>
<td>4.09</td>
<td>&lt;0.05</td>
</tr>
<tr>
<td>Russell</td>
<td>17.85</td>
<td>17.85</td>
<td>10.52</td>
<td>10.45</td>
<td>10.50</td>
<td>&lt;0.05</td>
</tr>
</tbody>
</table>

End of Appendix of Incremental Approach on other Multiple-bug programs
Appendix of Evaluation of Frequency Weighting Function of Other Multiple-bug Programs

Table: Average Rank Percentages Result on Three-Bug Programs using Traditional (Binary) and different $\alpha$ values for Siemens Test Suite and Unix datasets

<table>
<thead>
<tr>
<th>Metric</th>
<th>$\alpha$ = 0.1</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kulczynski2</td>
<td>17.86</td>
<td>16.34</td>
<td><strong>16.12</strong></td>
<td>16.89</td>
<td>17.26</td>
<td>18.02</td>
<td>18.09</td>
<td>19.04</td>
<td>22.18</td>
</tr>
</tbody>
</table>

Average Rank Percentages vs Alpha values for Three-Bug Programs (Siemens and subset of Unix)

Figure: Average Rank Percentages for Different $\alpha$ values (Three-Bug Programs)
Appendix of Evaluation of Frequency Weighting Function of Other Multiple-bug Programs

Table: Average Rank Percentages Result on Multiple-bug Space Programs using Traditional (Binary) and different $\alpha$ values

<table>
<thead>
<tr>
<th>Metric</th>
<th>$\alpha$ =</th>
<th>0.1</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>20</th>
<th>Bin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zoltar</td>
<td></td>
<td>4.01</td>
<td>4.46</td>
<td>4.28</td>
<td>4.15</td>
<td>3.33</td>
<td>2.70</td>
<td><strong>2.34</strong></td>
<td>2.38</td>
<td>2.42</td>
</tr>
</tbody>
</table>

Figure: Average Rank Percentages for Different $\alpha$ values (Multiple-Bug Programs for Space)
References I

Hailpern, B.; Santhanam, P.

Abreu, R., Zoeteweij, P., van Gemund, A.:
An Evaluation of Similarity Coefficients for Software Fault Localization.

R. Abreu, P. Zoeteweij, and A. van Gemund.
On the Accuracy of Spectrum-based Fault Localization.

W. Wong, J. Horgan, S. London, and A. Mathur.
Effect of Test Set Minimization on the Fault Detection Effectiveness of the All-Uses Criterion.
References II

M. Hollander and D. Wolfe  
Nonparametric statistical methods  

Wong, W.E. and Qi, Y. and Zhao, L. and Cai, K.Y.  
Effective Fault Localization using Code Coverage.  
Proceedings of the 31st COMPSAC 449–456

Eric Wong, W. and Debroy, V. and Choi, B.  
A family of code coverage-based heuristics for effective fault localization.  
Journal of Systems and Software

A Model for Ranking-based Software Diagnosis.  
References III

Spectral Debugging with Weights and Incremental Ranking.

Proceedings of the 2009 16th APSEC, 2009. 168–175

J. Heaton
Introduction to Neural Networks for Java, 2008.

Yu, Y. and Jones, J.A. and Harrold, M.J.
An empirical study of the effects of test-suite reduction on fault localization.

Jones, J.A. and Harrold, M.J. and Stasko, J.
Visualization of test information to assist fault localization.

References V

- Wong, W.E. and Horgan, J.R. and Mathur, A.P. and Pasquini, A.
  Test set size minimization and fault detection effectiveness: A case study in a space application. *JSS*, 1999. 79–89

- Dan Hao, Lu Zhang, Hao Zhong, Hong Mei and Jiasu Sun.
  Eliminating harmful redundancy for testing-based fault localization using test suite reduction: An experimental study.

References VII


References VIII

Renieres, M. and Reiss, SP.
Fault localization with nearest neighbor queries

Cleve, H. and Zeller, A.
Locating causes of program failures
In Proceedings of the 27th International Conference on Software Engineering, page 342-351, ACM.

Telecordia Technologies, Inc.
Telecordia software visualization and analysis toolsuite (χSuds)
In User Manual.
References IX

Wong, W.E. and Qi, Y. and Zhao, L. and Cai, K.Y. Effective Fault Localization using Code Coverage

Hsu, H.Y. and Jones, J.A. and Orso, A. RAPID: Identifying bug signatures to support debugging activities
Public Deployment of Cooperative Bug Isolation

Wong, W. Eric, Vidroha Debroy, Yihao Li, and Ruizhi Gao
Software Fault Localization Using DStar (D*)
References XI

Lei, Yan, Xiaoguang Mao, Ziying Dai and Chengsong Wang
Effective Statistical Fault Localization Using Program Slices
In *Proceedings of the 36th International Computer Software and Applications Conference (COMPSAC), 2012*, page 1-10, IEEE.

Wen, Wanzhi
Software Fault Localization based on Program Slicing Spectrum
Servant, Francisco, and James A. Jones
WhoseFault: Automatic developer-to-fault assignment through fault localization

Bandyopadhyay, Aritra, and Sudipto Ghosh
On the Effectiveness of the Tarantula Fault Localization Technique for Different Fault Classes
References XIII

- Bandyopadhyay, Aritra
  Improving spectrum-based fault localization using proximity-based weighting of test cases

- XiaoYuan Xie and Tsong Yueh Chen and BaoWen Xu
  Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques
Vidroha Debroy and Wong, W.E and Xiaofeng Xu and ByoungJu Choi
A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques
In *Proceedings of the 10th International Conference on Quality Software (QSIC), 2010*, page 13-22, IEEE.

Jones, J.A. and Harrold, M.J.
Empirical evaluation of the tarantula automatic fault-localization technique
References XV

Wong, W.E. and Shi, Y. and Qi, Y. and Golden, R.
Using an RBF neural network to Locate Program Bugs

A. Gonzalez
Automatic Error Detection Techniques based on Dynamic Invariants
References XVI

Wang, J and Han, J
BIDE: Efficient mining of frequent closed sequences

Santelices, R. and Jones, J.A. and Yu, Y. and Harrold, M.J.
Lightweight fault-localization using multiple coverage types

Zadeh, Lotfi A.
Fuzzy Sets
In *Information and Control Vol. 8 No. 3*, page 338-353, 1965
References XVII

Santelices, R. and Harrold, M.J.
Efficiently monitoring data-flow test coverage

Hao, D. and Zhang, L. and Pan, Y. and Mei, H. and Sun, J.
On similarity-awareness in testing-based fault localization

Newmark, J.
Statistics and probability in modern life, Saunders College Pub.
References XVIII

Bellard, F.
Tiny C Compiler
Retrieved http://bellard.org/tcc/

CNT
CNT
Retrieved http://sourceforge.net

Morris, R. and Cherry, L.
DC- An Interactive Desk Calculator
In UNIX time-sharing system: UNIX programmer’s manual, 1983
References XIX

Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests

Evaluating the accuracy of fault localization techniques

Software-artifact Infrastructure Repository
SIR
Andrews, JH and Briand, LC and Labiche, Y. 
Is mutation an appropriate tool for testing experiments? 

HuaJie Lee and Lee Naish and Kotagiri Ramamohanarao 
Effective Software Bug Localization Using Spectral Frequency Weighting Function 
Lyle, JR
Evaluating variations on program slicing for debugging
In *Dissertation Abstracts International Part B: Science and Engineering*[DISS. ABST. INT. PT. B- SCI. & ENG.], 1985,

Weiser, M. and Lyle, J.
Experiments on slicing-based debugging aids

Agrawal, H. and Horgan, J.R.
Dynamic program slicing
References XXII

**Weiser, M.D.**
Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method
*In Journal of Ann Arbor, MI, 1979*

**Weiser, M.**
Program slicing

**Weiser, M.**
Programmers use slices when debugging
*In Journal of Communications of the ACM Vol. 25 No. 7, 1982, ACM*
References XXIII

Korel, B. and Laski, J.
Dynamic slicing of computer programs

Chen, TY and Cheung, YY
Dynamic program dicing
In *Proceedings of Conference on Software Maintenance*, page 378-385, 1993

Lyle, J.R. and Weiser, M.
Automatic program bug location by program slicing
Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests
In *Proceedings of 6th International Conference on Software Reliability Engineering*, page 143-151, 1995

Agrawal, H. and Horgan, JR and London, S. and Wong, WE and Bellcore, M.
Fault localization using execution slices and dataflow tests
In *Proceedings of 6th International Conference on Software Reliability Engineering*, page 143-151, 1995
References XXV

Wong, W.E. and Qi, Y.
An execution slice and inter-block data dependency-based approach for fault localization

Bug isolation via remote program sampling

Scalable statistical bug isolation


Rhythm

Chao Liu and Xifeng Yan and Long Fei and Jiawei Han and Samuel P. Midkiff
SOBER: Statistical Model-based Bug Localization

Di Fatta, G. and Leue, S. and Stegantova, E.
Discriminative pattern mining in software fault detection
References XXVIII

Jiang, L. and Su, Z.
Automatic Isolation of Cause-effect Chains with Machine Learning
In Technical Report CSE-2005-32, University of California, Davis, 2005

Steinwart, I. and Christmann, A.
Support Vector Machines, Springer Verlag

Goethals, B.
Survey on Frequent Pattern Mining, Univ. of Helsinki

Breiman, L.
Random forests Vol. 45 No. 1, page 5-32, 2001, Springer Verlag
References XXIX

Statistical debugging: simultaneous identification of multiple bugs

Ng, A. and Jordan, M. and Weiss, Y.
On spectral clustering: Analysis and an algorithm
Hall, M. and Padua, D. and Pingali, K.
Compiler research: the next 50 years

Effect of test set minimization on fault detection effectiveness

Zeller, A.
From Automated Testing to Automated Debugging
In *Journal of Uni Passau*, Feb, 2000
Zeller, A.
Isolating cause-effect chains from computer programs

GNU GCC
GCC, the GNU Compiler Collection
AskIgor - Automated Debugging Service
Retrieved
http://www.st.cs.uni-saarland.de/askigor/

Romanovskiï, V.I.
Discrete Markov Chains, Wolters-Noordhoff

Jones, J.A. and Bowring, J.F. and Harrold, M.J.
Debugging in parallel
In Proceedings of the International Symposium on Software Testing and Analysis, 2007,
Ostrand, T.J. and Balcer, M.J.
The category-partition method for specifying and generating functional tests
In *Communications of the ACM* Vol. 31 No. 6, page 676-686, 1988, ACM

Quinlan, J.R.
C4. 5: programs for machine learning, Morgan Kaufmann

Kononenko, Igor and Matjaz Kukar
Machine Learning and Data Mining, Woodhead Publishing, 2007
Dickinson, W. and Leon, D. and Podgurski, A.
Finding failures by cluster analysis of execution profiles

Murtagh, F.
A survey of recent advances in hierarchical clustering algorithms

GCC, GNU
GDB: The GNU Project Debugger
Retrieved [http://www.gnu.org/software/gdb/]
References XXXV

- The University of Virginia, Biomedical Engineering Chemokines Dendrogram

- Briand, L.C. and Labiche, Y. and Liu, X.
  Using machine learning to support debugging with Tarantula

- Lee Naish and HuaJie Lee and Kotagiri Ramamohanarao
  Statements versus predicates in spectral bug localization
References XXXVI

Walczak, B, and Massart, DL
The radial basis functions-partial least squares approach as a flexible non-linear regression technique
In Journal of Analytica Chimica Acta Vol. 331 No. 3, 1996, Elsevier

Statistical debugging of sampled programs