Simplifying and Isolating Failure-Inducing Input

Andreas Zeller, Ralf Hildebrandt

Presented By: Anvika
20.05.2016

Motivation:
- Given some test cases, a program fails
- Decompose bug reports into simple test cases
- Minimal test cases that still cause the program to fail – Simplifying
- Every part of the input would be significant in reproducing the failure
- It helps in localising the fault, thus reducing time and effort in solving it
- A minimal test case gives valuable problem insights and has a general context
- It subsumes several other bug reports that differ in irrelevant details

Algorithm Overview:
- Dmin : the minimising delta debugging algorithm
- INPUT: Takes a failing test case
- Simplifies it by successive testing
- Stops when a minimal test case is reached
- OUTPUT: relevant part of failing test case i.e. removing any single entity would cause the failure to disappear

Input:
Simplification of the input by dadmin:

Grey characters indicate the ones removed in subsequent test cases

Formalisation:
- $R$, a set of possible configurations of circumstances
- Circumstances that cause a different test outcome
- $r \in R$ is a specific program run which can be either a failing or passing one
- Difference between the passing and failing run is the change that causes failure
- Smaller the change, the better

Decomposing Changes:
- A change can stand for:
  - insertion of a single character
  - adding tags
  - deletion of a character or characters
- The set of changes are $C = R^k$
- The function $r_{test} : R \rightarrow \{✓, X, ?\}$ determines for a program run $r \in R$
  - $✓$: Failure occurs
  - $X$: Failure doesn't occur
  - $?$: Unresolved test case
- A test case is a subset of changes
- Minimal test Case: No smaller subset of $c_X$ that causes test to fail

Minimising test cases:
- Test input subsets with removed characters
- A given test case:
  - Fails: if the program still crashes [$C_X$]
  - Passes: otherwise [$C_Y$]
- Testing for a change, in particular changeable circumstances
  - Program code
  - Data from storage or I/P devices
  - Environment
- Circumstances that cause a different program behaviour
Minimal test Cases:

- Achieving the global minimum is always good but practically impossible due to exponential complexity
- Local minimum comes to the rescue
- A test case is minimal if none of its subsets cause a failure
- Still need $2^{\|c\|} - 2$ tests
- Approximation: 1- minimality - removing any single change causes the failure to disappear

Minimising Algorithm:

- If $c_\times$ contains only one change, it is minimal
- Else, partition it into two subsets of similar size $\Delta_1$ and $\Delta_2$
- Three possible outcomes:
  - if test of $\Delta_1$ fails – it is a smaller test case
  - if test of $\Delta_2$ fails - it is a smaller test case
  - else none of them are possible simplifications

<table>
<thead>
<tr>
<th>Step</th>
<th>Test case</th>
<th>Test</th>
<th>Isol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\Delta_1$</td>
<td>$\checkmark$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\Delta_2$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\checkmark$</td>
<td>$\times$</td>
<td></td>
</tr>
</tbody>
</table>

Observations:

- Testing larger subsets increases the chances of test failing
- Testing smaller subsets leads to faster progression but a lesser chance of test failing
- Strategy:
  - $c_\times$ into larger subsets
  - Test each of the small subsets and their respective larger complements
  - Complement $\nabla_i = c_\times - \Delta_i$

Outcomes:

- If any of the $\Delta_i$ fails, then it is the smaller test case
- Else, its complement is the smaller test case and has to be reduced with $n-1$ subsets
- Increase granularity and test $2^n$ subsets, when no test fails
- Repeat until granularity cannot be increased further, $n > |c_\times|$ 
- Resulting change set is minimal
Example:

<table>
<thead>
<tr>
<th>Map</th>
<th>Test case</th>
<th>Ref.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \Delta_1 \cap V )</td>
<td><img src="image1.png" alt="Image" /></td>
</tr>
<tr>
<td>2</td>
<td>( \Delta_2 \cap V )</td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>3</td>
<td>( \Delta_3 \cap V )</td>
<td><img src="image3.png" alt="Image" /></td>
</tr>
<tr>
<td>4</td>
<td>( \Delta_4 \cap V )</td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
<tr>
<td>5</td>
<td>( \Delta_5 \cap V )</td>
<td><img src="image5.png" alt="Image" /></td>
</tr>
<tr>
<td>6</td>
<td>( \Delta_6 \cap V )</td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
<tr>
<td>7</td>
<td>( \Delta_7 \cap V )</td>
<td><img src="image7.png" alt="Image" /></td>
</tr>
<tr>
<td>8</td>
<td>( \Delta_8 \cap V )</td>
<td><img src="image8.png" alt="Image" /></td>
</tr>
<tr>
<td>9</td>
<td>( \Delta_9 \cap V )</td>
<td><img src="image9.png" alt="Image" /></td>
</tr>
<tr>
<td>10</td>
<td>( \Delta_{10} \cap V )</td>
<td><img src="image10.png" alt="Image" /></td>
</tr>
<tr>
<td>11</td>
<td>( \Delta_{11} \cap V )</td>
<td><img src="image11.png" alt="Image" /></td>
</tr>
<tr>
<td>12</td>
<td>( \Delta_{12} \cap V )</td>
<td><img src="image12.png" alt="Image" /></td>
</tr>
<tr>
<td>13</td>
<td>( \Delta_{13} \cap V )</td>
<td><img src="image13.png" alt="Image" /></td>
</tr>
<tr>
<td>14</td>
<td>( \Delta_{14} \cap V )</td>
<td><img src="image14.png" alt="Image" /></td>
</tr>
<tr>
<td>15</td>
<td>( \Delta_{15} \cap V )</td>
<td><img src="image15.png" alt="Image" /></td>
</tr>
<tr>
<td>16</td>
<td>( \Delta_{16} \cap V )</td>
<td><img src="image16.png" alt="Image" /></td>
</tr>
<tr>
<td>17</td>
<td>( \Delta_{17} \cap V )</td>
<td><img src="image17.png" alt="Image" /></td>
</tr>
<tr>
<td>18</td>
<td>( \Delta_{18} \cap V )</td>
<td><img src="image18.png" alt="Image" /></td>
</tr>
<tr>
<td>19</td>
<td>( \Delta_{19} \cap V )</td>
<td><img src="image19.png" alt="Image" /></td>
</tr>
<tr>
<td>20</td>
<td>( \Delta_{20} \cap V )</td>
<td><img src="image20.png" alt="Image" /></td>
</tr>
<tr>
<td>21</td>
<td>( \Delta_{21} \cap V )</td>
<td><img src="image21.png" alt="Image" /></td>
</tr>
<tr>
<td>22</td>
<td>( \Delta_{22} \cap V )</td>
<td><img src="image22.png" alt="Image" /></td>
</tr>
<tr>
<td>23</td>
<td>( \Delta_{23} \cap V )</td>
<td><img src="image23.png" alt="Image" /></td>
</tr>
<tr>
<td>24</td>
<td>( \Delta_{24} \cap V )</td>
<td><img src="image24.png" alt="Image" /></td>
</tr>
<tr>
<td>25</td>
<td>( \Delta_{25} \cap V )</td>
<td><img src="image25.png" alt="Image" /></td>
</tr>
<tr>
<td>26</td>
<td>( \Delta_{26} \cap V )</td>
<td><img src="image26.png" alt="Image" /></td>
</tr>
<tr>
<td>27</td>
<td>( \Delta_{27} \cap V )</td>
<td><img src="image27.png" alt="Image" /></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td><img src="image28.png" alt="Image" /></td>
</tr>
</tbody>
</table>

After 857 tests, the input is reduced to 77 characters
It is 1-minimal test case
Since, we know it is an optimisation fault, minimisation can be performed on a list of 31 optimisations
Response to GCC maintainers:
- The minimal test case
- Failure occurs due to optimisation
- \( \text{ffast-math} \) and \( \text{fforce-addr} \) prevent the failure

Continued...

```c
typedef int z;  
int t(double z[], int n, int i, int j); for( ; ; ) { 
    i = i + j + 1;  
    z[i] = (z[i-1] + z[i+1]);  
    return z[n];  
}
```

Isolating Failure-Inducing Differences:

- Number of tests required is proportional to the size of the input
- If the individual tests are fast then it can be handled
- What happens when tests are complicated or the size of inputs is large?
- Combine the earlier approach and narrow the set of differences, whenever a test either fails or passes.
Example:

| 2 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✗ |
| 4 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✗ |
| 7 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✓ |
| 6 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✓ |
| 5 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✓ |
| 3 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✓ |
| 1 | SELECT_NAME="priority",MULTIPLE_SIZE=? | ✓ |

Isolation:

- Initially give an empty passing case
- As a test case fails:
  - Smaller test case is a failing case
- When a test case passes:
  - Larger test case is used as a new passing test case
- This minimises the difference between the pass and fail test

Isolation vs Simplification:

- Simplification: Make each part of simplified test case relevant
- Isolation: Find one relevant part of test case
- Relevant: Removal of part or parts that make the failure go away
- Isolating differences will pinpoint failure cause much faster than minimising test case
- The programmer has to keep track of both the test cases

DD: extending ddmin

- Extend ddmin to work on two sets at a time
- Goal is to find a two sets, \( c' \) and \( c' \)
  - \( c' \): needs to be minimised
  - \( c' \): needs to be maximised
- The minimality is now applied to the difference \( \Delta_i \) and not the test case
- Find the 1- minimal difference between the failing and the passing test case
- Test \( c' \) \( \cup \) \( \Delta_i \) instead of just \( \Delta_i \)
Continued:

- Two new rules:
  - Increase to complement:
    - If \( c' \in A \) passes, which is the fail subset then reduce the difference between fail-subset and fail
  - Increase to subset:
    - If \( c' \notin A \) passes then that is the larger passing test

Performance Comparison:

- 'Ddmin' took 731 tests but 'dd' took only 59
- No of test runs is way less for dd than ddmin
- Dd is a predecessor of dd
- Dd doesn't guarantee 1- minimal subset
- dd can be a generalised replacement for Dd

Future Work:

- Domain- specific simplification methods
- Optimisation
- Undoing changes
- Program Analysis
- Other failure-inducing circumstances

Summary:

- Delta debugging algorithms simplify and isolate failure inducing input
- It can be applied to program invocations, program input and also to a sequence of user interactions
- The testing procedure is automated
- Easy to implement and straight-forward