Delta Debugging

CS169
Lecture 13

Announcements

• Requirements due March 8
• Examples now on class page

Debugging

• Debugging is invoked when testing finds a bug
  - Hard task, but good tools can help a lot

• Hard problems in debugging
  - Find a minimal test case that reproduces the bug
    • Minimal = Each element is relevant
  - Locate the fault in the source code

A Generic Algorithm

• How do people solve these problems?

• Binary search
  - Cut the test case in half
  - Iterate

• Brilliant idea: Why not automate this?

Delta Debugging Usage Scenarios

• Scenario 1: program used to work, but after a number of changes it fails
  - Find the change that is responsible for failure

• Scenario 2: program works on an input, but after a number of changes to the input, it fails
  - Find the change that is responsible for failure
  - Special case, works on empty input, fails on input I, find the smallest failing subset of I

Delta Debugging Setup

• Program P works on input I

• Failure after a set of changes C to P(I)
  - To either the program P
  - Or the input I

• Find the smallest set of changes that still fail
  - For simplicity we consider changes to program only
Version I

• Assume
  - There is a set of changes $C$
  - There is a single change that causes failure
    • And still causes failure when mixed with other changes
  - Every subset of changes makes sense
    • Any subset of changes produces a test case that either
      passes $\sqrt{\text{ }}$ or fails $X$

Algorithm for Version I

```c
/* invariant: P succeeds and P with changes $c_1$, $c_2$ fails */
DD(P,\{c_1,\ldots,c_n\}) =
  \text{if } n = 1 \text{ return } \{c_1\}
  \text{let } P_1 = P \setminus \{c_1, \ldots, c_{n/2}\}
  \text{let } P_2 = P \setminus \{c_{n/2+1}, \ldots, c_n\}
  \text{if } P_1 = \sqrt{\text{ }} \text{ then } DD(P,\{c_{n/2+1}, \ldots, c_n\})
  \text{else } DD(P,\{c_1, \ldots, c_{n/2}\})
```

Version I: Example

• Assume $C = \{1, 2, 3, 4, 5, 6, 7, 8\}$
  - The bug is in 7

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>$\sqrt{\text{ }}$</td>
</tr>
<tr>
<td>5 6 7 8</td>
<td>$X$</td>
</tr>
<tr>
<td>5 6</td>
<td>$\sqrt{\text{ }}$</td>
</tr>
<tr>
<td>7 8</td>
<td>$X$</td>
</tr>
<tr>
<td>7</td>
<td>$X$</td>
</tr>
</tbody>
</table>

Version I: Comments

• This is just binary search!

• A very sensible algorithm to try first
  - By hand, or automated

• Assumptions hold for most bugs
  - If $P \oplus (C_1 \cup C_2) = X$ then
    • Either $P \oplus C_1 = X$ and $P \oplus C_2 = q$ or
    • Or $P \oplus C_2 = X$ and $P \oplus C_1 = q$
  - It becomes interesting when this is not true…

Extensions

• Let’s get fancy. Assume:
  - Any subset of changes may cause the bug
    - But no undetermined (?) tests, yet

And the world is:

- Monotonic (failing changes are not annihilated):
  - $P \oplus C = X \Rightarrow P \oplus (C \setminus C) \neq q$
- Unambiguous (unique smallest set of failing changes):
  - $P \oplus C = X \Rightarrow \exists C : (P \oplus C) \neq q$
- Consistent (all changes make sense, for now)
  - $P \oplus C = \sqrt{\text{ }}$

Scenarios

• Try binary search:
  - Partition changes $C$ into $C_1$ and $C_2$
  - If $P \oplus C_1 = X$, recurse with $C_1$
  - If $P \oplus C_2 = X$, recurse with $C_2$

• Notes:
  - By consistency, only other possibilities are
    - $P \oplus C_1 = X$ and $P \oplus C_2 = q$
    - $P \oplus C_1 = q$ and $P \oplus C_2 = X$
  - What happens in these cases?
Multiple Failing Changes

• If $P \oplus C_1 = X$ and $P \oplus C_2 = X$
  - (This would be ruled out by unambiguity)

• There exist a subset of $C_1$ that fails
  - And another subset inside $C_2$

• We can simply continue to search inside $C_1$
  - And then inside $C_2$

• And we'll find two separate subsets that reproduce the failure
  - Choose the smallest one of the two

Interference

By monotonicity, if $P \oplus C_1 = q \quad P \oplus C_2 = q$
then no subset of $C_1$ or $C_2$ causes failure

So the failure must be a combination of elements from $C_1$ and $C_2$

This is called interference

Handling Interference

• The cute trick:
  - Consider $P \oplus C_1$
  - Find minimal $D_2 \subseteq C_2$ s.t. $P \oplus (C_1 \cup D_2) = X$
  - Consider $P \oplus C_2$
  - Find minimal $D_1 \subseteq C_1$ s.t. $P \oplus (C_2 \cup D_1) = X$
  - $P \oplus ((C_1 \setminus D_2) \cup (C_2 \setminus D_1)) = P \oplus (D_1 \setminus D_2)$
  - Then by unambiguity $P \oplus (D_1 \setminus D_2)$ fails
  - This is also minimal

Interference Example

Consider 8 changes, of which 3, 5 and 7 cause the failure, but only when applied together

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>q, interference</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>q</td>
</tr>
<tr>
<td>3 5 6 7 8 X</td>
<td>q</td>
</tr>
<tr>
<td>3 5 6 7 8 X</td>
<td>X</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>q, interference</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>X</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>X</td>
</tr>
</tbody>
</table>

Algorithm

/* invariant: $P$ succeeds, $P$ with changes $c_1, \ldots, c_n$ fails
find the smallest subset that still fails */

$$DD(P, \{c_1, \ldots, c_n\}) =$$
if $n = 1$ return $(c_1)$
$P_1 \leftarrow P \oplus (c_1 \cup \ldots \cup c_{\lfloor n/2 \rfloor})$
$P_2 \leftarrow P \oplus (c_{\lfloor n/2 \rfloor + 1} \cup \ldots \cup c_n)$
if $P_1 = X$ then $DD(P, \{c_1, \ldots, c_{\lfloor n/2 \rfloor}\})$
else if $P_2 = X$ then $DD(P, \{c_{\lfloor n/2 \rfloor + 1} \cup \ldots \cup c_n\})$
else $DD(P_2, \{c_1, \ldots, c_{\lfloor n/2 \rfloor}\}) \setminus DD(P_1, \{c_{\lfloor n/2 \rfloor + 1} \cup \ldots \cup c_n\})$

Complexity

• If a single change induces the failure, then logarithmic
  - Why?

• Otherwise, linear
  - Assumes constant time per invocation
  - Is this realistic?
  - What is a more realistic complexity?
Revisit the Assumptions

- All three assumptions are suspect
- But consistency is egregious
  - In practice, many inconsistent sets of changes
  - E.g., because some changes must be made together
  - Otherwise: build errors, execution errors

Handling Inconsistency

- Idea
  - Change sets closer to ∅ or all changes are more likely to be consistent
  - Get information from a subset C
  - And its complement : C
  - If we do this with smaller C, we are more likely to stay consistent

Handling Inconsistency: Cases

For each C \( \in \{C_1, \ldots, C_n\} \) (partition of changes)
1. If \( P \oplus C = X \), recurse on C
   - As before
2. If \( P \oplus C = \sqrt{} \) and \( P \oplus : C = \sqrt{} \), interference
   - As before
3. If \( P \oplus C = ? \) and \( P \oplus : C = \sqrt{} \), preference
   - C has a failure-inducing subset
   - Possibly in interference with : C
4. Otherwise, try again
   - Repeat with twice as many subsets

Interference Example

- Consider 8 changes, of which #8 causes failure, and 2, 3 and 7 must be together

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test {1,2,3,4}</td>
<td>1 2 3 4 ?</td>
</tr>
<tr>
<td>Test complement</td>
<td>5 6 7 8 ? Inconsistent</td>
</tr>
<tr>
<td>Test {1,2}</td>
<td>1 2 ?</td>
</tr>
<tr>
<td>Test complement</td>
<td>3 4 5 6 7 8 ?</td>
</tr>
<tr>
<td>Test {3,4}</td>
<td>3 4 ?</td>
</tr>
<tr>
<td>Test complement</td>
<td>1 2 5 6 7 8 ?</td>
</tr>
<tr>
<td>Test {5,6}</td>
<td>5 6 ?</td>
</tr>
<tr>
<td>Test complement</td>
<td>1 2 3 4 7 8 X We dig down here</td>
</tr>
</tbody>
</table>

... improvements

- Consider 8 changes, of which #8 causes failure, and 2, 3 and 7 must be together

Results

- This really works!
- Isolates problematic change in gdb
  - After lots of work (by machine)
    - 178,000 lines changed, grouped into 8700 groups
    - Can do 230 builds/tests in 24 hours
    - Would take 37 days to try 8700 groups individually
    - The algorithm did this in 470 tests (48 hours)
    - We can do better
  - Doing this by hand would be a nightmare
Opinions

- How to address the assumptions
  - Unambiguity
    - Just look for one set of failure-inducing changes
  - Consistency
    - We dealt with it already
  - Monotonicity
    - Deal with it next

Delta Debugging ++

- Drop all of the assumptions
- What can we do?
- Different problem formulation
  *Find a set of changes that cause the failure, but removing any change causes the failure to go away*
  - This is 1-minimality

Model

- Once again, a test either
  - Passes \(\checkmark\)
  - Fails \(\times\)
  - Is unresolved ?

Naïve Algorithm

- To find a 1-minimal subset of \(C\), simply
- Remove one element \(c\) from \(C\)
- If \(C - \{c\} = X\), recurse with smaller set
- If \(C - \{c\} \neq X\), \(C\) is 1-minimal

Analysis

- In the worst case,
  - We remove one element from the set per iteration
  - After trying every other element
- Work is potentially
  \[N + (N-1) + (N-2) + \ldots\]
- This is \(O(N^2)\)

Work Smarter, Not Harder

- We can often do better
- Silly to start out removing 1 element at a time
  - Try dividing change set in 2 initially
  - Increase # of subsets if we can't make progress
  - If we get lucky, search will converge quickly
**Algorithm**

\[
\text{DD}(P, \{C_1, \ldots, C_n\}) = \\
\text{if } P \oplus C_i = X \text{ then } \text{DD}(P, \{C_{i1}, C_{i2}\}) \\
\text{elseif } P \oplus \neg C_i = X \text{ then } \text{DD}(P, \{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n\}) \\
\text{elseif } \text{sizeof}(C_i) = 1 \text{ then } C_1 \cup \ldots \cup C_n \\
\text{else } \text{DD}(P, \{C_{i1}, C_{i2}, \ldots, C_{n1}, C_{n2}\})
\]

where \(C_{i1}\) and \(C_{i2}\) are the two halves of \(C_i\)

---

**Analysis**

- Worst case is still quadratic
- Subdivide until each set is of size 1
  - Reduced to the naive algorithm
- Good news
  - For single, monotone failure, converges in \(\log N\)
  - Binary search again

---

**A Distinction**

- **Simplification**
  - Removing any piece of the test removes the failure; every piece of the test is relevant
- **Isolation**
  - Find at least one relevant piece of the test; removing this piece makes the failure go away

---

**Simplification vs. Isolation**

- So far, DD does simplification
- Performance is inherently limited
  - Must remove every piece of test separately to verify that it is fully simplified
  - Performance limited by size of output
- Isolation, however, can be more efficient
  - Just need to find a change that makes working test case fail

---

**Formalization**

- Consider two test cases
  - \(P \oplus C = q\)
  - \(P \oplus D = X\)
  - \(C \supseteq D\)
- Then \(D - C\) is 1- minimal difference if
  - For each \(c \subseteq (D - C)\)
    - \(P \oplus (C \setminus \{c\}) = q\)
    - \(P \oplus (D - \{c\}) = X\)

---

**1-Minimality**

- There is always a 1 minimal pair
- **Proof**
  - Initially
    - original program works \(C < \)
    - modified program fails \(D \geq \) (all changes)
  - DD produces \(D'\) that is minimal
  - Now grow \(C\) with elements from \(D'\)
Algorithm

\[ DD(P, \ C, \ D, \ \{e_1, \ldots, e_n\}) = \]
\[ \text{if } P \oplus (C \cup \{e_i\}) = X \text{ then} \]
\[ \text{if } P \oplus (D \cup \{e_{i_1}, e_{i_2}\}) = \text{q} \text{ then} \]
\[ DD(P, D \cup \{e_i\}, \ D, \ \{e_{i_1}, e_{i_2}\}) \]
\[ \text{else if } P \oplus (D - e_i) = X \text{ then} \]
\[ DD(P, C \cup \{e_i\}, \ D, \ \{e_{1}, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\}) \]
\[ \text{else if } P \oplus (C \cup \{e_i\}) = \text{q} \text{ then} \]
\[ DD(P, C \cup \{e_i\}, \ D, \ \{e_{1}, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n\}) \]
\[ \text{else } DD(P, C, \ D, \ \{e_{11}, \ldots, e_{n1}, e_{n2}\}) \]

Analysis

\[ \text{- Worst case is the same} \]
\[ \text{- Worst case example is the same} \]
\[ \text{- Quadratic} \]
\[ \text{- But best case has improved significantly} \]
\[ \text{- If all tests either pass or fail, runs in } \log N \]

Case Studies

\[ \text{- Famous paper showed 40% Unix utilities failed on random inputs} \]
\[ \text{- Repeated that experiment with DD} \]
\[ \text{And found the same results, 10 years later!} \]
\[ \text{- Conclusion: Nobody cares} \]
\[ \text{- Applied delta debugging to minimize test cases} \]
\[ \text{Revealed buffer overrun, parsing problems} \]

The Importance of Changes

\[ \text{- Basic to delta debugging is a change} \]
\[ \text{- We must be able to express the difference between the good and bad examples as a set of changes} \]
\[ \text{- But notion of change is semantic} \]
\[ \text{- Not easy to capture in a general way in a tool} \]
\[ \text{- And notion of change is algorithmic} \]
\[ \text{Poor notion of change \rightarrow many unresolved tests} \]
\[ \text{Performance goes from linear (or sub-linear) to quadratic} \]

Notion of Change

\[ \text{- We can see this in the experiments} \]
\[ \text{Some gdb experiments took 48 hours} \]
\[ \text{Improvements came from improving notion of changes} \]
\[ \text{- Also important to exploit correlations between changes} \]
\[ \text{Some subsets of changes require other changes} \]
\[ \text{Again, can affect asymptotic performance} \]

Opinion

\[ \text{- Delta Debugging is a technique, not a tool} \]
\[ \text{- Bad News:} \]
\[ \text{Probably must be reimplemented for each significant system} \]
\[ \text{To exploit knowledge of changes} \]
\[ \text{- Good News:} \]
\[ \text{Relatively simple algorithm, significant payoff} \]
\[ \text{It's worth reimplementing} \]