Debugging without Debuggers

Lecture 6
CS195

Debugging sans Debuggers

- Debugging is more than debuggers
- In fact, debuggers are often the last resort
- Two other common problems:
  - Figuring out which program change caused a bug
  - Reducing a test case to a minimal example

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

- Find set of changes that cause a program to fail a test case
- Want to find a minimal set of changes that cause failure

Version I

- Assume
  - There is a set of changes \( C \)
  - There is a single change that caused failure
  - Every set of changes is possible
    - Any subset produces a test case that either passes \( \top \) or fails \( \bot \)

Algorithm for Version I

/* invariant: \( P \) with changes \( c_1, \ldots, c_n \) fails */

\[ DD(P, c_1, \ldots, c_n) = \]
if \( n = 1 \) return \( c_1 \)
let \( P_1 = P \oplus (c_1 \ldots c_{n/2}) \)
let \( P_2 = P \oplus (c_{n/2+1} \ldots c_n) \)
if \( P_1 = \top \)
  then \( DD(P, c_{n/2+1} \ldots c_n) \)
else \( DD(P, c_1 \ldots c_{n/2}) \)

This is just binary search...
Extensions

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

- And the world is
  - Monotonic:
    \[ P \circ C = x \Rightarrow P \circ (C \cup C) = x \]
  - Unambiguous:
    \[ P \circ C = x \land P \circ C = y \Rightarrow P \circ (C \cap C) = x \land y \]
  - Consistent
    \[ P \circ C = ? \]

Scenarios

Try binary search:
- Divide changes \( C \) into \( C_1 \) and \( C_2 \)
- If \( P \circ C = x \), recurse with \( C_1 \)
- If \( P \circ C = x \), recurse with \( C_2 \)

Notes:
- At most one case can apply, by ambiguity
- By consistency, only other possibility is
  \[ P \circ C_1 = ? \land P \circ C_2 = ? \]
- What happens in this case?

Interference

By monotonicity, if \( P \circ C_1 = \bot \land P \circ C_2 = \bot \) 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 \( P \circ C_1 \)
    - Find minimal \( D_1 \subset C_1 \) s.t. \( P \circ C_1 \cup D_1 \neq x \)
  - Consider \( P \circ C_2 \)
    - Find minimal \( D_2 \subset C_2 \) s.t. \( P \circ C_2 \cup D_1 \neq x \)

- Then by unambiguity
  \[ P \circ (C_1 \cup D_2) \cap (C_2 \cup D_1) = P \circ (D_1 \cup D_2) \]
- This is also minimal

Example: 3 & 6 (of 8) Cause Failure

```
<table>
<thead>
<tr>
<th>1 2 3 4 5 6 7 8</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>√</td>
</tr>
<tr>
<td>5 6 7 8</td>
<td>√</td>
</tr>
<tr>
<td>3 4 5 6 7 8</td>
<td>X</td>
</tr>
</tbody>
</table>

interference

<table>
<thead>
<tr>
<th>1 2 3 4 5 6</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4 5 6 7 8 X</td>
<td></td>
</tr>
<tr>
<td>2 3 4 5</td>
<td></td>
</tr>
<tr>
<td>1 2 3 4 6 X</td>
<td></td>
</tr>
</tbody>
</table>
```

Algorithm

/* invariant: \( P \) with changes \( c_0, \ldots, c_n \) fails */

\[
\text{DD}(P, \{c_0, \ldots, c_n\}) = \\
\text{if } n = 1 \text{ return } \{c_0\} \\
P_1 \leftarrow P \circ \{c_0, \ldots, c_{n-1}\} \\
P_2 \leftarrow P \circ \{c_{n-1}, \ldots, c_n\} \\
\text{if } P_1 = x \text{ then DD}(P, \{c_0, \ldots, c_{n-1}\}) \\
\text{else } P_2 = x \text{ then DD}(P, \{c_{n-1}, \ldots, c_n\}) \\
\text{else } \text{DD}(P, \{c_0, \ldots, c_{n-1}\}) \cup \text{DD}(P, \{c_{n-1}, \ldots, c_n\})
\]
**Complexity**

- If a single change induces the failure, then logarithmic
  - Why?
- Otherwise, linear
  - Assumes constant time per invocation
  - Is this realistic?

**Handling Inconsistency**

- Idea
  - Get information from a subset \( C \)
  - And its complement \( \neg C \)
- We may also work with more than 2 subsets at a time

**Complexity**

- Linear
- Two test for \( C \) and \( \neg C \)
- At most double tests at each level
  - For at most \( \log N \) levels

**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
  - Or in order, etc.

**Handling Inconsistency: Cases**

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

**Improvement**

- If \( P \oplus R = \checkmark \), then no subset of \( R \) causes failure
  - By monotonicity
- Accumulate some such \( R \) and apply at every opportunity
  - If \( P \oplus C \oplus R = \checkmark \) and \( P \oplus \neg C \oplus R = \checkmark \), interference
- Why? To promote consistency
  - Closer to original, failing program
  - More likely to be consistent
  - See Section 5 of the paper
Results

- This really works!
- Isolates problematic change in gdb
  - After lots of work
  - But finding it by hand would be a nightmare

Opinions

- The assumptions aren’t realistic
  - Monotonicity
  - Unambiguity
  - Consistency
- Apparently one author thinks so, too
  - Second paper

Delta Debugging **

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

Model

- Once again, a test either
  - Passes ✓
  - Fails ×
  - Is unresolved ?

Naïve Algorithm

- To find a 1-minimal subset of C, simply
- Remove one element c from C
- If C - {c} = \( \chi \), recurse with smaller set
- If C - {c} \( \neq \chi \), 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
  \( N \cdot (N-1) \cdot (N-2) \cdot \ldots \)
- 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

\[ DD(P, \{C_1, \ldots, C_n\}) = \]
- if \( P \cup C = \emptyset \) then \( DD(P, \{C_1, C_2\}) \)
- if \( P \cup C = \emptyset \) then \( DD(P, \{C_1, \ldots, C_{i-1}, C_{i+1}, \ldots, C_n\}) \)
- otherwise \( DD(P, \{C_1, C_2, \ldots, C_n\}) \)

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 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
  - \( P @ C = \checkmark \)
  - \( P @ D = \times \)
  - \( C \subseteq D \)
- Then \( D - C \) is 1-minimal if
  - For each \( c \in (D - C) \)
    - \( P @ (C \cup (c)) = \checkmark \)
    - \( P @ (D - (c)) = \times \)
1. Minimality

- There is always a 1-minimal pair

- Proof
  - Initially
    - original program works \( C = \emptyset \)
    - modified program fails \( D = \{ \text{all changes} \} \)
  - DD produces \( D' \) that is minimal
  - Now add elements of \( D' \) to \( C \) until failure

Algorithm

\[
\text{DD}(P, C \cup D(e_1, \ldots, e_n)) =
\begin{cases}
  \text{if } P \circ (C \cup e_i) = X & \text{then} \\
  \text{DD}(P.C \cup D(e_1, e_2, \ldots, e_n)) & \text{if } P \circ (D - e_i) = X \\
  \text{DD}(P.D - e_i.D(e_1, e_2, \ldots, e_n)) & \text{if } P \circ (C \cup e_i) = X \\
\end{cases}
\]

Analysis

- Worst case is the same
  - Worst case example is the same
  - Quadratic

- But best case has improved significantly
  - If all tests either pass or fail, runs in \( \log N \)

Algorithm

\[
\text{DD}(P, C \cup D(e_1, \ldots, e_n)) =
\begin{cases}
  \text{if } P \circ (C \cup e_i) = X & \text{then} \\
  \text{DD}(P.C \cup D(e_1, e_2, \ldots, e_n)) & \text{if } P \circ (D - e_i) = X \\
  \text{DD}(P.D - e_i.D(e_1, e_2, \ldots, e_n)) & \text{if } P \circ (C \cup e_i) = X \\
\end{cases}
\]

Case Studies

- Many in the papers
  - And convincing, too

- Isolating failure in modified \texttt{gdb}
  - 178,000 modified source lines
  - Symptom was that program simply crashed
  - What was the bug? Changing
    - "Set arguments to give..."
    - "Set argument list to give..."

A Depressing Example

- Famous paper showed 40% Unix utilities failed on random inputs

- Repeated that experiment
  - And found the same results, 10 years later!
  - Conclusion: Nobody cares

- Applied delta debugging to minimize test cases
  - Revealed buffer overrun, parsing problems
The Importance of Changes

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

Opinion

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

Nation of Change

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