

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 q or fails X

Algorithm for Version I

/* invariant: P succeeds and P with changes c_1, \dots, c_n fails */

```
DD(P, {c1, ..., cn}) =
  if n = 1 return {c1}
  let P1 = P / {c1 ... cn/2}
  let P2 = P / {cn/2+1 ... cn}
  if P1 = q
    then DD(P, {cn/2+1 ... cn})
    else DD(P, {c1 ... cn/2})
```

Version I: Example

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

Configuration	Result
1 2 3 4	q
5 6 7 8	X
5 6	q
7 8	X
(7)	X

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 $P \circ (C_1 \setminus C_2) = X$ then
 - Either $P \circ C_1 = X$ and $P \circ C_2 = q$ or
 - Or $P \circ C_2 = X$ and $P \circ C_1 = q$
 - 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):
 $P \circ C = X \wedge P \circ (C \setminus C) \neq q$
 - Unambiguous (unique smallest set of failing changes):
 $P \circ C = X \wedge P \circ C = X \wedge P \circ (C \setminus C) \neq q$
 - Consistent (all changes make sense, for now)
 $P \circ C \neq ?$

Scenarios

- Try binary search:
 - Partition changes C into C_1 and C_2
 - If $P \circ C_1 = X$, recurse with C_1
 - If $P \circ C_2 = X$, recurse with C_2
- Notes:
 - By consistency, only other possibilities are
 $P \circ C_1 = X$ and $P \circ C_2 = X$
 $P \circ C_1 = q \wedge P \circ C_2 = q$
 - What happens in these cases?

Multiple Failing Changes

- If $P_c C_1 = X$ and $P_c C_2 = X$
 - (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

Prof. Brewer CS 169 Lecture 13

13

Interference

By monotonicity, if $P_c C_1 = \alpha$ - $P_c C_2 = \alpha$
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

Prof. Brewer CS 169 Lecture 13

14

Handling Interference

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

Prof. Brewer CS 169 Lecture 13

15

Interference Example

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

Configuration	Result
1 2 3 4	α
5 6 7 8	α <i>interference</i>
1 2 5 6 7 8	α
3 4 5 6 7 8	X
5 6 7 8	X
1 2 3 4 5 6	α
1 2 3 4 7 8	α <i>interference</i>
1 2 3 4 5 7 8	X
1 2 3 4 5 6 7	X

Prof. Brewer CS 169 Lecture 13

16

Algorithm

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

```

DD(P, {c1, ..., cn}) =
  if n = 1 return {c1}
  P1  $\stackrel{!}{=} P_c \{c_1 \dots c_{n/2}\}$ 
  P2  $\stackrel{!}{=} P_c \{c_{n/2+1} \dots c_n\}$ 
  if P1 = X then DD(P, {c1 ... cn/2})
  elseif P2 = X then DD(P, {cn/2+1 ... cn})
  else DD(P2, {c1 ... cn/2}) \ DD(P1, {cn/2+1 ... cn},)
    
```

Prof. Brewer CS 169 Lecture 13

17

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?

Prof. Brewer CS 169 Lecture 13

18

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 \emptyset or all changes are more likely to be consistent
 - Get information from a subset C
 - And its complement \bar{C}
 - If we do this with smaller C , we are more likely to stay consistent

Handling Inconsistency: Cases

For each $C \in \{C_1, \dots, C_n\}$ (partition of changes)

1. If $P_C = X$, recurse on C
 - As before
2. If $P_C = \alpha$ and $P_{\bar{C}} = \alpha$, interference
 - As before
3. If $P_C = ?$ and $P_{\bar{C}} = \alpha$, preference
 - C has a failure-inducing subset
 - Possibly in interference with \bar{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

	Configuration				Result			
Test {1,2,3,4}	1	2	3	4				?
Test complement					5	6	7	8
Test {1,2}	1	2						?
Test complement			3	4	5	6	7	8
Test {3,4}			3	4				?
Test complement	1	2			5	6	7	8
Test {5,6}					5	6		?
Test complement	1	2	3	4			7	8
							X	We dig down here

...

Improvement

- If $P_C = \alpha$, then no subset of R causes failure
 - By monotonicity
- Accumulate some such R and apply at every opportunity
 - If $P_C = \alpha$ and $P_{\bar{C}} = \alpha$, 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 (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 \square
 - Fails \times
 - Is unresolved $?$

Naïve Algorithm

- To find a **1** minimal subset of C , simply
- Remove one element c from C
- If $C - \{c\} = \times$, recurse with smaller set
- If $C - \{c\} \neq \times$, 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
$$N + (N-1) + (N-2) + \dots$$
- 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, {C1, ..., Cn}) =  
  if P ⊆ Ci = X then DD(P, {Ci1, Ci2})  
  elseif P ⊆ Ci ; Ci = X then DD(P, {C1, ..., Ci-1, Ci+1, ..., Cn})  
  elseif sizeof(Ci) = 1 then C1 \ ... \ Cn  
  else DD(P, {Ci1, Ci2, ..., Cn1, Cn2})
```

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 naïve 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 ⊆ C = q
- P ⊆ D = X
- C ⊆ D

• Then D - C is 1- minimal difference if

- For each $c \in (D - C)$
- P ⊆ (C \ {c}) ≠ q
 - P ⊆ (D - {c}) ≠ X

1-Minimality

• There is always a 1- minimal pair

• Proof

- Initially
 - original program works C = q
 - modified program fails D = { all changes }
- DD produces D' that is minimal
- Now grow C with elements from D'

Algorithm

```
DD(P, C, D, {e1, ..., en}) =  
  if P ⊆ (C \ ei) = X then  
    DD(P, C, C \ ei, {e1, e2})  
  elseif P ⊆ (D - ei) = α then  
    DD(P, D - ei, D, {e1, e2})  
  elseif P ⊆ (D - ei) = X then  
    DD(P, C, D - ei, {e1, ..., ei-1, ei+1, ..., en})  
  elseif P ⊆ (C \ ei) = α then  
    DD(P, C \ ei, D, {e1, ..., ei-1, ei+1, ..., en})  
  else DD(P, C, D, {e11, e12, ..., en1, en2})
```

Profs. Brewer CS 169 Lecture 13

37

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 $\log N$

Profs. Brewer CS 169 Lecture 13

38

Case Studies

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

Profs. Brewer CS 169 Lecture 13

39

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 as 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

Profs. Brewer CS 169 Lecture 13

40

Notion 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

Profs. Brewer CS 169 Lecture 13

41

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

Profs. Brewer CS 169 Lecture 13

42