

Delta-Debugging

Wei Le

March 13, 2020

Real-World Scenario

In July 1999, Bugzilla listed more than 370 open bug reports for Mozilla's web browser

- These were not even simplified
- Mozilla engineers were overwhelmed with the work
- They created the Mozilla BugAthon: a call for volunteers to simplify bug reports

When you've cut away as much HTML, CSS, and JavaScript as you can, and cutting away any more causes the bug to disappear, you're done.

— Mozilla BugAthon call

Delta-debugging (dd) is an idea

The idea derives a set of tools and algorithms, that have different application settings:

- ▶ we have seen a number of test generation techniques that all in some form produce inputs in order to trigger failures. If they are successful – that is, the program actually fails – we must find out why the failure occurred and how to fix it.
- ▶ *Yesterday, my program worked, Today it does not, Why?*
- ▶ Simplifying and Isolating Failure-Inducing Input
- ▶ Text books *Why programs fail*
- ▶ Examples *Reducer for fuzzers*
- ▶ More *here*

The Problem

- ▶ GDB: GNU debugger for C
- ▶ DDD: graphical front-end of GDB
- ▶ Upgrade GDB from 4.16 to 4.17
- ▶ the integration of GDB and DDD no longer work

Goal

Determine the minimal set of failure inducing changes

Existing Work

regression containment (used at Cray research for compiler development):
apply (the ordered) changes one a time until regression tests fail

- ▶ logical change can be large
- ▶ totally ordered changes are considered, no problems of interference, inconsistencies, granularity

Challenges

Challenges

- ▶ *Interference*: single change does not cause the problem, but multiple changes together produce the failure, e.g., merging the products of parallel development
- ▶ *Inconsistency*: the combinations of changes that do not result in a testable program, **you cannot just apply any changes and expect the program to run smoothly**
 1. Integration failure: a change cannot be applied, it may require earlier changes that are not included in the configuration, it may also be in conflict with another change, but the third conflict-resolving change is missing
 2. Construction failure: syntactic and semantic errors after applying the changes
 3. Execution failure: cannot run correctly, e.g., missing "create file" statement, so you cannot open a file
- ▶ *Granularity*: logical change can contain many lines
 - ▶ logical change: the developer commit a change
 - ▶ textual changes: you run a diff, you obtain a set of chunks

Formally define the problem

Background definitions: (sometimes, a paper defines a set of terms for easily presenting their work, these terms may not be applicable beyond the paper)

- ▶ configurations
- ▶ baseline
- ▶ test
- ▶ failure inducing, failure inducing change set, minimal failure-inducing change set

Formally define the problem

Assumptions: the configuration is

- ▶ monotony
- ▶ unambiguity
- ▶ consistency

Basic idea of dd: binary search

1. we partition c into two subsets c_1 and c_2 and test each of them.
2. Found in c_1 . The test of c_1 fails— c_1 contains a failure-inducing change.
3. Found in c_2 . The test of c_2 fails— c_2 contains a failure-inducing change.
4. Interference. Both tests pass. Since we know that testing $c = c_1 \cup c_2$ fails, the failure must be induced by combination of some change sets in c_1 and some change sets in c_2

Basic idea of dd: binary search

Step	c_i	Configuration	$test$
1	c_1	1 2 3 4	✓
2	c_2 5 6 7 8	✗
3	c_1 5 6 . .	✓
4	c_2 7 8	✗
5	c_1 7 .	✗
7 is found			
Result	 7 .	

Basic idea of dd: binary search

Interference: search both halves

Step	c_i	Configuration	$test$	
1	c_1	1 2 3 4	✓	
2	c_2 5 6 7 8	✓	
3	c_1	1 2 . . 5 6 7 8	✓	
4	c_2	. . 3 4 5 6 7 8	✗	
5	c_1	. . 3 . 5 6 7 8	✗	3 is found
6	c_1	1 2 3 4 5 6 . .	✗	
7	c_1	1 2 3 4 5 . . .	✓	6 is found
Result		. . 3 . . 6 . .		

DD basic algorithm

$dd(c) = dd_2(c, \emptyset)$ where

$dd_2(c, r) = \text{let } c_1, c_2 \subseteq c \text{ with } c_1 \cup c_2 = c, c_1 \cap c_2 = \emptyset, |c_1| \approx |c_2| \approx |c|/2$

$$\text{in } \begin{cases} c & \text{if } |c| = 1 \text{ ("found")} \\ dd_2(c_1, r) & \text{else if } test(c_1 \cup r) = \mathbf{x} \text{ ("in } c_1\text{")} \\ dd_2(c_2, r) & \text{else if } test(c_2 \cup r) = \mathbf{x} \text{ ("in } c_2\text{")} \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise ("interference")} \end{cases}$$

Return: the change set that contains the bug

Complexity

Step	c_i	Configuration	test	
1	c_1	1 2 3 4	✓	
2	c_2 5 6 7 8	✓	
3	c_1	1 2 . . 5 6 7 8	✓	
4	c_2	. . 3 4 5 6 7 8	✓	
5	c_1	1 . 3 4 5 6 7 8	✓	2 is found
6	c_2	. 2 3 4 5 6 7 8	✓	1 is found
7	c_1	1 2 3 . 5 6 7 8	✓	4 is found
8	c_2	1 2 . 4 5 6 7 8	✓	3 is found
9	c_1	1 2 3 4 5 6 . .	✓	
10	c_2	1 2 3 4 . . 7 8	✓	
11	c_1	1 2 3 4 5 . 7 8	✓	6 is found
12	c_2	1 2 3 4 . 6 7 8	✓	5 is found
13	c_1	1 2 3 4 5 6 7 .	✓	8 is found
14	c_2	1 2 3 4 5 6 . 8	✓	7 is found
Result		1 2 3 4 5 6 7 8		

Complexity

Worst case: all changes are failure inducing, still linear in terms of the

Step	c_i	Configuration								test
1	c_1	1	2	3	4	✓
2	c_2	5	6	7	8	✓
3	c_1	1	2	.	.	5	6	7	8	✓
4	c_2	.	.	3	4	5	6	7	8	✓
5	c_1	1	.	3	4	5	6	7	8	✓ 2 is found
6	c_2	.	2	3	4	5	6	7	8	✓ 1 is found
7	c_1	1	2	3	.	5	6	7	8	✓ 4 is found
8	c_2	1	2	.	4	5	6	7	8	✓ 3 is found
9	c_1	1	2	3	4	5	6	.	.	✓
10	c_2	1	2	3	4	.	.	7	8	✓
11	c_1	1	2	3	4	5	.	7	8	✓ 6 is found
12	c_2	1	2	3	4	.	6	7	8	✓ 5 is found
13	c_1	1	2	3	4	5	6	7	.	✓ 8 is found
14	c_2	1	2	3	4	5	6	.	8	✓ 7 is found
Result		1	2	3	4	5	6	7	8	

number of changes

Handling inconsistencies

Review: reasons of inconsistencies:

1. Integration failure: a change cannot be applied, it may require earlier changes that are not included in the configuration, it may also be in conflict with another change, but the third conflict-resolving change is missing
2. Construction failure: syntactic and semantic errors after applying the changes
3. Execution failure: cannot run correctly, e.g., missing "create file" statement, so you cannot open a file

Testing output in presence of inconsistencies

- found: If testing any c_i fails, then c_i contains a failure-inducing subset.
- interference: If testing any c_i passes and its complement $\overline{c_i}$ passes as well, then the change sets c_i and $\overline{c_i}$ form an interference
- preference: If testing any c_i is unresolved, and testing $\overline{c_i}$ passes, then c_i contains a failure-inducing subset and is preferred. In the following test cases, $\overline{c_i}$ must remain applied to promote consistency.

Step	c_i	Configuration	test	
1	c_1	1 2 3 4	?	Testing c_1, c_2
2	c_2 5 6 7 8	✓	⇒ Prefer c_1
3	c_1	1 2 . . 5 6 7 8	...	

Handling inconsistencies

Try again: we repeat the process with $2n$ subsets, where n is the number of groups, to increase the chance of consistency

Step	c_i	Configuration	<i>test</i>	
1	$c_1 = \bar{c}_2$	1 2 3 4	?	Testing c_1, c_2
2	$c_2 = \bar{c}_1$ 5 6 7 8	?	\Rightarrow Try again
3	c_1	1 2	?	Testing c_1, \dots, c_4
4	c_2	. . 3 4	?	
5	c_3 5 6 . .	✓	
6	c_4 7 8	?	
7	\bar{c}_1	. . 3 4 5 6 7 8	?	Testing complements
8	\bar{c}_2	1 2 . . 5 6 7 8	?	
9	\bar{c}_3	1 2 3 4 . . 7 8	✗	
10	\bar{c}_4	1 2 3 4 5 6 . .	?	\Rightarrow Try again

Handling inconsistencies

should we just return nodes 1, 2, 3, 4, 7 and 8 as failure inducing? no,

we should try to minimize the change set some more: try again – re-partition!

- ▶ if ci pass, it is not failure inducing,=search for failure inducing sets from 6 changes,
- ▶ why changes 5 and 6 remain applied?

Handling inconsistencies

Step	c_i	Configuration	test	
11	c_1	1 . . . 5 6 . .	✓	Testing c_1, \dots, c_6
12	c_2	. 2 . . 5 6 . .	?	
13	c_3	. . 3 . 5 6 . .	?	
14	c_4	. . . 4 5 6 . .	✓	
15	c_5 5 6 7 .	?	
16	c_6 5 6 . 8	✗	8 is found
Result	 8		

- ▶ nodes 1, 4, 5, 6 are not failure inducing
- ▶ 8 is the cause
- ▶ changes 2, 3, 7 should always be applied together?
- ▶ if step 16 passes, then we know nodes 1, 4 and 8 are also not failure inducing, only nodes 2, 3 and 7 left, we just need to exclude all other changes?

dd+ algorithm

Output:

- ▶ find a minimal set of failure-inducing changes, and they are safe (remember we run them!)
- ▶ at least, exclude all changes that are safe (because we run them then we know they are not failure inducing) and not failure inducing

dd+ algorithm

$dd^+(c) = dd_3(c, \emptyset, 2)$ where

$dd_3(c, r, n) =$

let $c_1, \dots, c_n \subseteq c$ such that $\bigcup c_i = c$, all c_i are pairwise disjoint,
and $\forall c_i (|c_i| \approx |c|/n)$;

let $\bar{c}_i = c - (c_i \cup r)$, $t_i = test(c_i \cup r)$, $\bar{t}_i = test(\bar{c}_i \cup r)$,

$c' = c \cap \bigcap \{\bar{c}_i \mid \bar{t}_i = \mathbf{X}\}$, $r' = r \cup \bigcup \{c_i \mid t_i = \mathbf{\checkmark}\}$, $n' = \min(|c'|, 2n)$,

$d_i = dd_3(c_i, \bar{c}_i \cup r, 2)$, and $\bar{d}_i = dd_3(\bar{c}_i, c_i \cup r, 2)$

in $\begin{cases} c & \text{if } |c| = 1 \text{ ("found")} \\ dd_3(c_i, r, 2) & \text{else if } t_i = \mathbf{X} \text{ for some } i \text{ ("found in } c_i\text{")} \\ d_i \cup \bar{d}_i & \text{else if } t_i = \mathbf{\checkmark} \wedge \bar{t}_i = \mathbf{\checkmark} \text{ for some } i \text{ ("interference")} \\ d_i & \text{else if } t_i = \mathbf{?} \wedge \bar{t}_i = \mathbf{\checkmark} \text{ for some } i \text{ ("preference")} \\ dd_3(c', r', n') & \text{else if } n < |c| \text{ ("try again")} \\ c' & \text{otherwise ("nothing left")} \end{cases}$

Avoid inconsistencies

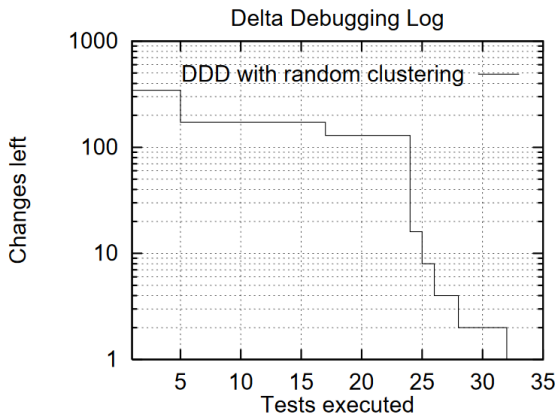
- ▶ group changes with additional information (location, lexical, syntactic, semantic, process)
- ▶ predicting test outcomes without running them (especially which change sets will lead to inconsistencies): only run ordered change sets

Study 1

Failure info:

- ▶ invoking a name of non-existing file, DDD 3.1.2 dumped core, DDD 3.1.1 prints an error message
- ▶ 116 logical changes, 344 textual changes

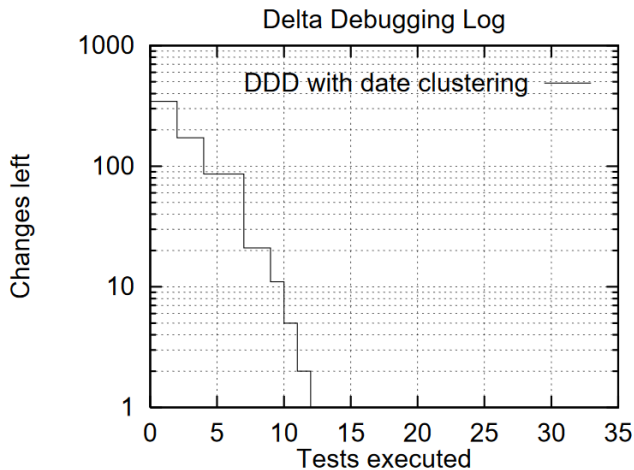
Study 1



(a) with random clustering

4: 172 changes, #31 find the failure inducing changes

Study 1



(b) with date clustering

Study 1

12 test runs, 58 minutes

```
diff -r1.30 -r1.30.4.1 ddd/gdbinit.C
295,296c296
<   string classpath =
<       getenv("CLASSPATH") != 0 ? getenv("CLASSPATH") : "
---
>   string classpath = source_view->class_path();
```

source_view is an uninitialized pointer, lead to core dump

Study 2

Failure info:

- ▶ the integration of GDB and DDD no longer work when GDB upgraded from 4.16 to 4.17
- ▶ 178 k changed lines, 8721 textual changes
- ▶ 370 seconds a change, apply individual changes take about 37 days

Study 2

Random clustering:

- ▶ increase the number of subsets, reduce the changesets in each subset
- ▶ most of first 457 tests unresolved
- ▶ at test 458, find that one of subsets that contains 36 changes is failure inducing
- ▶ use the rest 12 tests to determine a single failure-inducing change
- ▶ run a total of 470 tests, took 48 hours

Study 2

An optimized approach: considering dependencies between changes

- ▶ group changes based on the directory they are located
- ▶ group changes based on the common files
- ▶ within a file, changes are grouped according to the common usage of identifiers (keep changes together if they operated on common variables or functions)
- ▶ scan error messages of unresolved tests, find all changes that reference the identifies reported in the error messages, try again (to find a good construction)

Study 2

- ▶ ran 9 tests with various directory combinations, find failure inducing directory
- ▶ test 289, 20 hours, find the single line error inducing changes
- ▶ broken down 178 k lines to a single line change

Study 2

GDB: show args

GDB 4.16

Arguments to give program being debugged when it is started is "a b c"

GDB 4.17

Argument list to give program being debugged when it is started is "a b c"

DDD failed to parse the string in the new version

change in GDB source code:

```
diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```

Thought Provoking Work

What are your comments and thoughts?

Conclusions and future work

- ▶ automatic approaches to isolate regression causes
- ▶ How to group changes? Using dependency information
- ▶ Using code coverage tools to select changes that have not executed

Further Reading

- ▶ a short tutorial for delta debugging
- ▶ more delta debugging