CS1632, Lecture 18: Pairwise and Combinatorial Testing

Wonsun Ahn

Let’s Test A Word Processor

- Let’s say there are ten possible font effects
    - Italic
    - Bold
    - Underline
    - Strikethrough
    - Superscript
    - Shadow
    - Embossed
    - 3-D
    - Outline
    - Inverse

These can be combined

- Plain text
- Superscript
- **Bold**
- ~~*Italic and strikethrough*~~
- **<u>Bold and underlined</u>**
- ~~***Bold italic strikethrough shadowed superscript***~~

How many tests would you need to test all the possible font combinations?

Exhaustive Testing: 1024 Tests

- 2 choices per each variable (true or false)
- 10 variables
- All possible combinations = 2^{10}

That’s quite a few tests…

b	i	u	s	h	e	d	o	n
T	T	T	T	T	T	T	T	T
T	T	T	T	T	T	T	T	F
T	T	T	T	T	T	T	F	T
T	T	T	T	T	T	T	F	F
T	T	T	T	T	T	F	T	T
T	T	T	T	T	T	F	T	F
T	T	T	T	T	T	F	F	T
T	T	T	T	T	T	F	F	F
T	T	T	T	T	F	T	T	T
T	T	T	T	T	F	T	T	F
T	T	T	T	T	F	T	F	T
T	T	T	T	T	F	T	F	F
T	T	T	T	T	F	F	T	T
T	T	T	T	T	F	F	T	F
T	T	T	T	T	F	F	F	T
T	T	T	T	T	F	F	F	F
T	T	T	T	F	T	T	T	T
T	T	T	T	F	T	T	T	F
T	T	T	T	F	T	T	F	T
T	T	T	T	F	T	T	F	F
T	T	T	T	F	T	F	T	T
T	T	T	T	F	T	F	T	F
T	T	T	T	F	T	F	F	T
T	T	T	T	F	T	F	F	F
T	T	T	T	F	F	T	T	T
T	T	T	T	F	F	T	T	F
T	T	T	T	F	F	T	F	T
T	T	T	T	F	F	T	F	F
T	T	T	T	F	F	F	T	T
T	T	T	T	F	F	F	T	F
T	T	T	T	F	F	F	F	T
T	T	T	T	F	F	F	F	F
T	T	T	F	T	T	T	T	T
T	T	T	F	T	T	T	T	F
T	T	T	F	T	T	T	F	T
T	T	T	F	T	T	T	F	F
T	T	T	F	T	T	F	T	T
T	T	T	F	T	T	F	T	F
T	T	T	F	T	T	F	F	T
T	T	T	F	T	T	F	F	F
T	T	T	F	T	F	T	T	T
T	T	T	F	T	F	T	T	F
T	T	T	F	T	F	T	F	T
T	T	T	F	T	F	T	F	F
T	T	T	F	T	F	F	T	T
T	T	T	F	T	F	F	T	F
T	T	T	F	T	F	F	F	T

But it’s necessary!

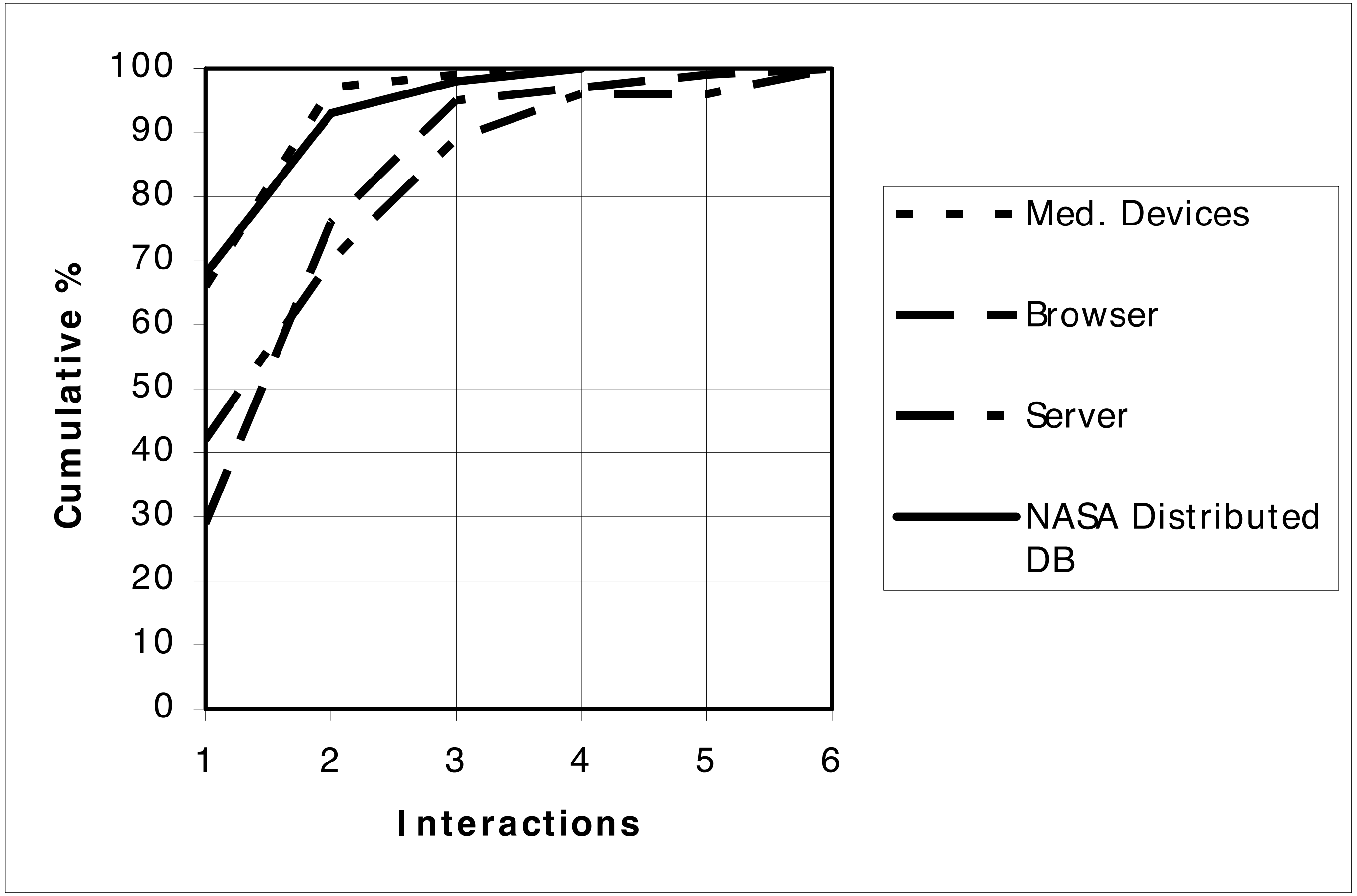
- What if a problem only occurs with shadowed, bold, italic text?
  - Doesn’t occur with just shadowed text
  - Doesn’t occur with just bold text
  - Doesn’t occur with just italic text
  - The 3 in combination interact with each other to cause defect
- Can’t test all interactions unless you test all combinations
- But then how do you deal with the exponentially increasing tests?

Turns Out Other People Have Thought About This!

- The National Institute of Standards and Technology (NIST) did a survey
  - See “Practical Combinatorial Testing”: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf
  - Study of dozens of applications in 6 domains:
    Medical devices, Web Browser, Web Server, Database, Network Security, TCAS

- Q: Do defects really occur as a result of interactions between variables?
  - If not, we can just test each of the 10 font effects individually!
- Q: If so, how many variables are typically involved in a defect?
  - If less than 10, we don’t have to test all combinations of 10 font effects.
  - If 2, we can just test interactions between all pairs of font effects.

Number of Defects for Interactions 1 to 6



Same data, but with more Domains

Vars	Medical Devices	Browser	Server	NASA GSFC	Network Security	TCAS
1	66	29	42	68	17	*
2	97	76	70	93	62	53
3	99	95	89	98	87	74
4	100	97	96	100	98	89
5		99	96		100	100
6		100	100			

Table 1. Number of variables involved in triggering software failures

Takeaways from the Survey

- Defects do occur as a result of interactions between variables
  - Defects covered by just a single variable: 17 – 68%
- At *max*, just *SIX* variables are involved in a defect
  - For all domains, 100% defects are covered by up to 6 interactions
- *Majority* of defects are found just by testing all possible *pairs*
  - Defects covered by up to 2 interacting variables: 53 – 97%

Pairwise Testing

- Testing all possible pairs of interactions (a.k.a “all-pairs” testing)
  - Bold / Italic,
  - Subscript / Bold
  - Underline / Strikethrough
  - Every possible pairing of two variables
- Testing all possible interactions within a pair. E.g.:
  - Not-Bold / Not-Italic
  - Bold / Not-Italic
  - Not-Bold / Italic
  - Bold / Italic

Naïve Pairwise Testing: 180 Tests

- For our font-effects: it was 1,024 (2 ^ 10) tests to test exhaustively.
- How many tests would be required to test only pairs of interactions?

- All possible pairs of interactions: $\binom{10}{2} = \frac{10 * 9}{2} = 45$
- All possible combinations within a pair: 2 * 2 = 4
- So 45 * 4 = 180 tests.

- Already pretty good, but we can do much better!

Pairwise Testing w/ Covering Array: 8 Tests

No.	BOLD	ITALIC	STRIKETHROUGH	UNDERLINE	THREAD	SHADOW	SUPERSCRIPT	SUBSCRIPT	EMBOSSED	ENGRAVED
1	FALSE	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE	TRUE	FALSE
2	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
3	TRUE	FALSE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE
4	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
5	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE
6	FALSE	FALSE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
7	-	-	-	-	-	-	-	FALSE	-	FALSE
8	-	-	-	-	-	-	-	TRUE	-	TRUE

- Wow, how did we reduce it to 8 tests? (from 180 tests, no less)
- Key: a single test case tests 10 font-effects at once (not just a pair)
  - Many pairs are tested at once in a single test (45 pairs to be exact)
  - Test 1: tests BOLD/ITALIC = FALSE/FALSE, ITALIC/STRIKETHROUGH = FALSE/TRUE, …
- Above is called a *covering array* (will tell you how to make this soon)

What if Pairwise Testing is not Enough?

- We need to “dial up” the number of possible interactions
  - To check for any *t* number of interactions
- For example, check every three-way interaction ($t = 3$):
  - Bold / Italic / Underline
- Or four-way ($t = 4$)
  - Bold / Italic / Underline / Superscript
- All the way up to six-way ($t = 6$)
  - According to NIST survey, no need to go beyond this point

Combinatorial Testing

- This generalized testing for any *t* is known as "combinatorial testing"

- Combinatorial (*math*):
  - Relating to the selection of a given number of elements from a larger number

- Combinatorial (*testing*):
  - Relating to testing interactions between *t* variables from entire set of variables

- Pairwise testing is an instance of combinatorial testing where *t* = 2

Combinatorial Testing Example

- Let’s test with t = 6 (the max required according to NIST)

- Recall that:
  - # tests required for exhaustive testing was 1,024
  - # tests required for pairwise testing (with covering array) was 8

- How many to test all six-way interactions?
  - And the answer is: 165 (with covering array)

Interesting!

- Pairwise testing (8 tests): catches 53 – 97% of defects
- Six-way testing (165 tests): catches ~100% of defects
- Exhaustive testing (1024 tests): catches 100% of defects

- Only when using a good covering array!
- Good covering arrays for each situation is given in: https://math.nist.gov/coveringarrays/ipof/ipof-results.html
  - These are not optimal (creating an optimal covering array is NP-Hard)
  - But they are pretty close

Cost of Testing

- Number of tests (cost of testing) for each *t*:

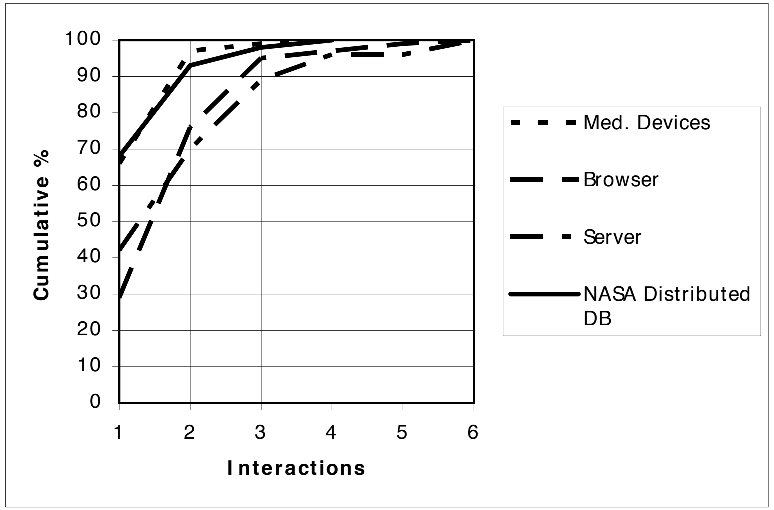
t-way	2	3	4	5	6
No. of Tests	8	18	41	87	165

→ Exponential increase!

- Testing Cost = $O(v^{t} * \log k)$
  - ***t***: number of interactions
  - *k*: number of variables
  - *v*: number of values a variable can take
- Let's look at each factor *t*, *k*, *v* in more depth

Law of Diminishing Returns on *t*

- *$O(v^{t} * \log k)$*: Cost increases exponentially on *t*
- Benefit saturates quickly as we increase *t*



- Cost/benefit analysis says: have a minimal *t* with decent coverage
  - Typically *t = 2* or *t = 3*

But Lots of Variables: Not a Problem!

- *$O(v^t$ * **log k**$)$*: Cost increases logarithmically on k
- If we had 20-variable system (with 4 values per variable)
  - Exhaustive testing: 1 trillion tests (approx.)
  - All 2-way interactions: 37 tests
  - All 6-way interactions: 224 K tests (approx.)
- If we had 30-variable system (with 4 values per variable)
  - Exhaustive testing: 10^{18} tests (approx.)
  - All 2-way interactions: 41 tests
  - All 6-way interactions: 424 K tests (approx.)
- As long as you limit *t*, life is not that difficult!

How about Values per Variable?

- $O(\mathbf{v}^t * \log k)$: values per variable can have a significant impact
- Depending on type of variable, *v* can be pretty big …
  - For boolean variables (like out font-effects) $v = 2$
  - For integer variables, $v = 2^{32} = 4$ *gigs*!
- But we learned about equivalence classes and not testing every input
  - *v* = # of boundary and interior values chosen using equivalence classes

Creating Covering Arrays

Covering Arrays

- *Covering array*: set of test cases covering all *t*-way combinations
- At below is a covering array where t = 3

Tests

A	B	C	D	E	F	G	H	I	J
0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	0	0	0	0	1
1	0	1	1	0	1	0	1	0	0
1	0	0	0	1	1	1	0	0	0
0	1	1	0	0	1	0	0	1	0
0	0	1	0	1	0	1	1	1	0
1	1	0	1	0	0	1	0	1	0
0	0	0	1	1	1	0	0	1	1
0	0	1	1	0	0	1	0	0	1
0	1	0	1	1	0	0	1	0	0
1	0	0	0	0	0	0	1	1	1
0	1	0	0	0	1	1	1	0	1

Steps To Make Covering Array

- Example we will use
  - Variables: Bold, Italic, Underline
  - *t = 2* (Pairwise covering array)

1. Enumerate the set of all possible tests

No.	BOLD	ITALIC	UNDERLINE
1	FALSE	FALSE	FALSE
2	FALSE	FALSE	TRUE
3	FALSE	TRUE	FALSE
4	FALSE	TRUE	TRUE
5	TRUE	FALSE	FALSE
6	TRUE	FALSE	TRUE
7	TRUE	TRUE	FALSE
8	TRUE	TRUE	TRUE

Steps To Make Covering Array

2. Make a list of all *t*-way interactions for desired *t*
   - Bold / Italic
   - Bold / Underline
   - Italic / Underline
3. Complete "mini truth table" for each *t*-way interaction

No.	BOLD	ITALIC
1	FALSE	FALSE
2	FALSE	TRUE
3	TRUE	FALSE
4	TRUE	TRUE

No.	BOLD	UNDERLINE
1	FALSE	FALSE
2	FALSE	TRUE
3	TRUE	FALSE
4	TRUE	TRUE

No.	ITALIC	UNDERLINE
1	FALSE	FALSE
2	FALSE	TRUE
3	TRUE	FALSE
4	TRUE	TRUE

→ Each mini truth table must be covered by final choice of tests

Steps To Make Covering Array

4. Cover each *t*-way interaction mini truth table:
   - For each entry in mini truth table, add test that fulfills it
     E.g.

No.	BOLD	UNDERLINE
3	TRUE	FALSE

     can be fulfilled by:

No.	BOLD	ITALIC	UNDERLINE
5	TRUE	FALSE	FALSE

   - If an already added test case fulfills the entry, nothing to do!
     E.g. if above test already added and

No.	BOLD	ITALIC
3	TRUE	FALSE

     needs fulfilling,

No.	BOLD	ITALIC	UNDERLINE
5	TRUE	FALSE	FALSE

     has you covered already!

5. Continue until all mini truth tables are covered

Sounds easy enough. Why is it NP-Hard?

- Note there are *multiple* candidates to choose from.

  E.g. Bold / Underline = TRUE / FALSE can be fulfilled by:

No.	BOLD	ITALIC	UNDERLINE
5	TRUE	FALSE	FALSE

But also:

No.	BOLD	ITALIC	UNDERLINE
7	TRUE	TRUE	FALSE

- There are only a handful of optimal choices.
  - Here is where the NP-Hardness creeps in.
  - We'll just choose randomly. It affects quality but not correctness of solution.

Covering Array Example

Test	Bold	Italic	Underline		
1	F	F	F		Bold / Italic
2	F	F	T		Bold / Underline
3	F	T	F		Italic / Underline
4	F	T	T		
5	T	F	F		
6	T	F	T		
7	T	T	F		
8	T	T	T		

Covering Array Example – Bold / Italic

Test	Bold	Italic	Underline		
1	F	F	F		Bold / Italic
2	F	F	T		Bold / Underline
3	F	T	F		Italic / Underline
4	F	T	T		
5	T	F	F		
6	T	F	T		
7	T	T	F		
8	T	T	T		

Covering Array Example – Bold / Underline

Test	Bold	Italic	Underline		
1	F	F	F		Bold / Italic
2	F	F	T		Bold / Underline
3	F	T	F		Italic / Underline
4	F	T	T		
5	T	F	F		
6	T	F	T		
7	T	T	F		
8	T	T	T		

Covering Array Example – Italic / Underline

Test	Bold	Italic	Underline		
1	F	F	F		Bold / Italic
2	F	F	T		Bold / Underline
3	F	T	F		Italic / Underline
4	F	T	T		
5	T	F	F		
6	T	F	T		
7	T	T	F		
8	T	T	T		

Run a Subset of Tests

Test	Bold	Italic	Underline		
1	F	F	F		Bold / Italic
2	F	F	T		Bold / Underline
3	F	T	F		Italic / Underline
4	F	T	T		
5	T	F	F		Necessary Tests
6	T	F	T		Unnecessary Tests
7	T	T	F		
8	T	T	T		

Can Minimize Further Using Better Algorithms

Test	Bold	Italic	Underline			
1	F	F	F		Bold / Italic	
2	F	F	T		Bold / Underline	
3	F	T	F		Italic / Underline	
4	F	T	T			
5	T	F	F		Necessary Tests	
6	T	F	T		Unnecessary Tests	
7	T	T	F			
8	T	T	T			

What is a Better Algorithm?

- Determining the optimal covering array is an NP-Hard problem.
- But there are some good algorithms out there that approximate it.
- “IPOG: A General Strategy for T-Way Software Testing” (ECBS ’07): https://www.nist.gov/publications/ipog-general-strategy-t-way-software-testing

Do I have to Learn the Algorithm?

- No, you don’t have to learn the algorithm and apply it yourself. ☺
- You can use the pre-generated covering array for your situation:
  - https://math.nist.gov/coveringarrays/ipof/ipof-results.html
- If your situation is not covered in the above, use NIST ACTS:
  - https://csrc.nist.gov/Projects/automated-combinatorial-testing-for-software/downloadable-tools
  - An implementation of the IPOG algorithm

How about the Outputs?

Test Oracle Problem

- Covering arrays limit number of tests you have to do

- But they may still run into the thousands for a large program
  - Sheer number of tests means we may need to autogenerate them
  - That means we need to autogenerate expected outputs along with the inputs

- How do we autogenerate expected output?
  - Need an oracle – same situation we faced with stochastic testing
  - May consider testing properties (invariants) applicable to all outputs

Now Please Read Textbook Chapter 17