Spec-Based Testing

Combinatorial Strategies for Forming Test Cases

"What if defects are caused by interactions among many blocks from many characteristics,
that is when certain combinations of triggers come together"

What's an interaction?

- **One-Way:**

```
if (acctBalance <= 0) status = "OVER";
```

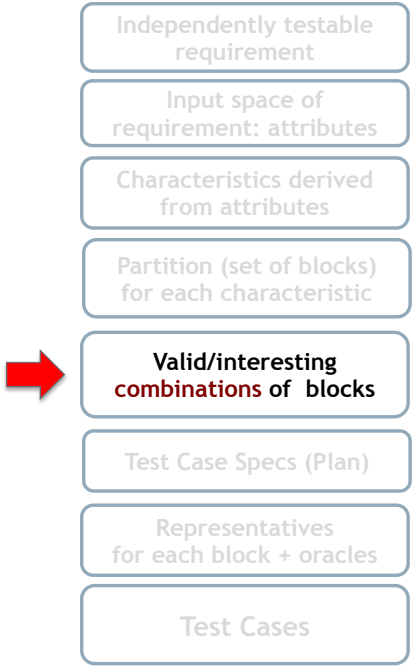
- **Two-Way:**

```
if (acctBalance <= 0 && !overdraftProtection)
   status = "OVER";
```

- **Three-Way:**

```
if (acctBalance <= 0 && !overdraftProtection
   && chequeAmount >= DEPOSIT_LIMIT) status = "ERROR"
```

Combinatorial Strategies



Criteria for choosing combinations

- Once characteristics and partitions are defined, the next step is to *choose combinations of blocks to test*
- We can use different input space model ***coverage criteria*** to choose *effective* subsets of combinations
- Each coverage criterion applies an optimization strategy to reduce the number of combinations
- Meaning of *"effective"*: *"good enough"*, depends on the *context*

We know that some combinations are mathematically or conceptually impossible, but for now, let's assume all combinations are feasible

The most obvious criterion is to choose all combinations

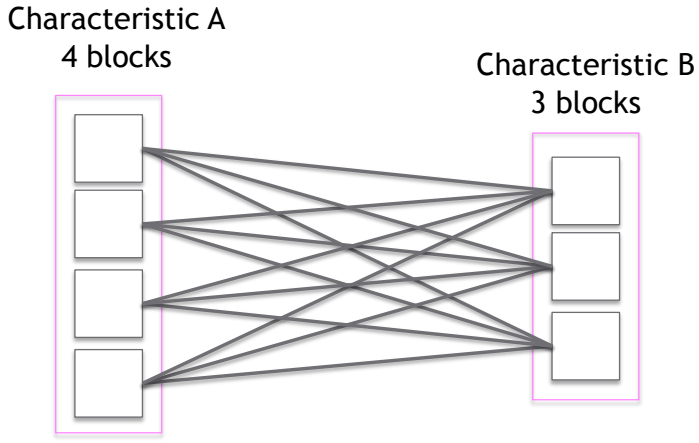
All-Choice (AC) : All combinations of blocks from all characteristics must be used.

- Number of test cases is the product of the number of blocks, $|B_q|$, in each characteristic q:

$$\prod_q |B_q|$$

- This could easily lead to combinatorial explosion!

All-Choice: *all combinations*



4 x 3 = 12 test cases

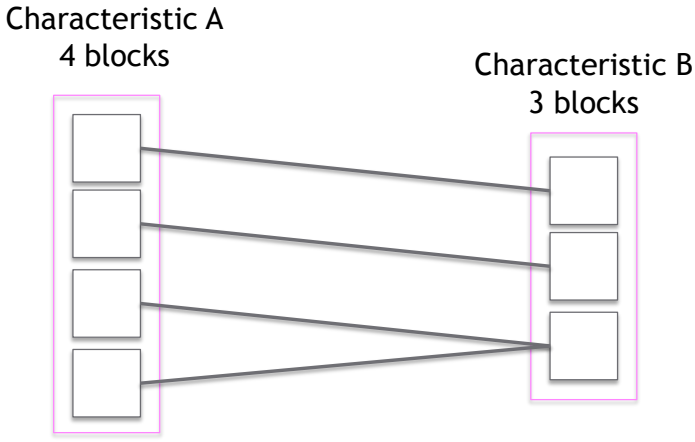
The least we can do is to cover each block with a minimum number of test cases

If all AC yields too many test cases…

- One extreme optimization is to try *at least one* value from each block

Each-Choice (EC) : Each block for each characteristic must be used in at least one test case

Each-Choice



Cover every block once in a test case: 4 + 3 = 7 blocks to cover

4 test cases instead of 12 AC

A nice compromise is to look at characteristics two at a time and cover all pairs of blocks between them

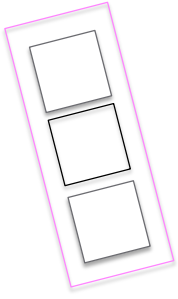
EC yields few tests – *cheap* but could be *ineffective*

- Another approach asks a block to be combined with each of the other blocks

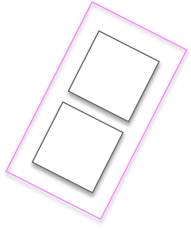
All-Pairs (aka Pair-Wise) (AP): Each block of each characteristic must be combined with every block from every other characteristic in at least one test case

EC AC

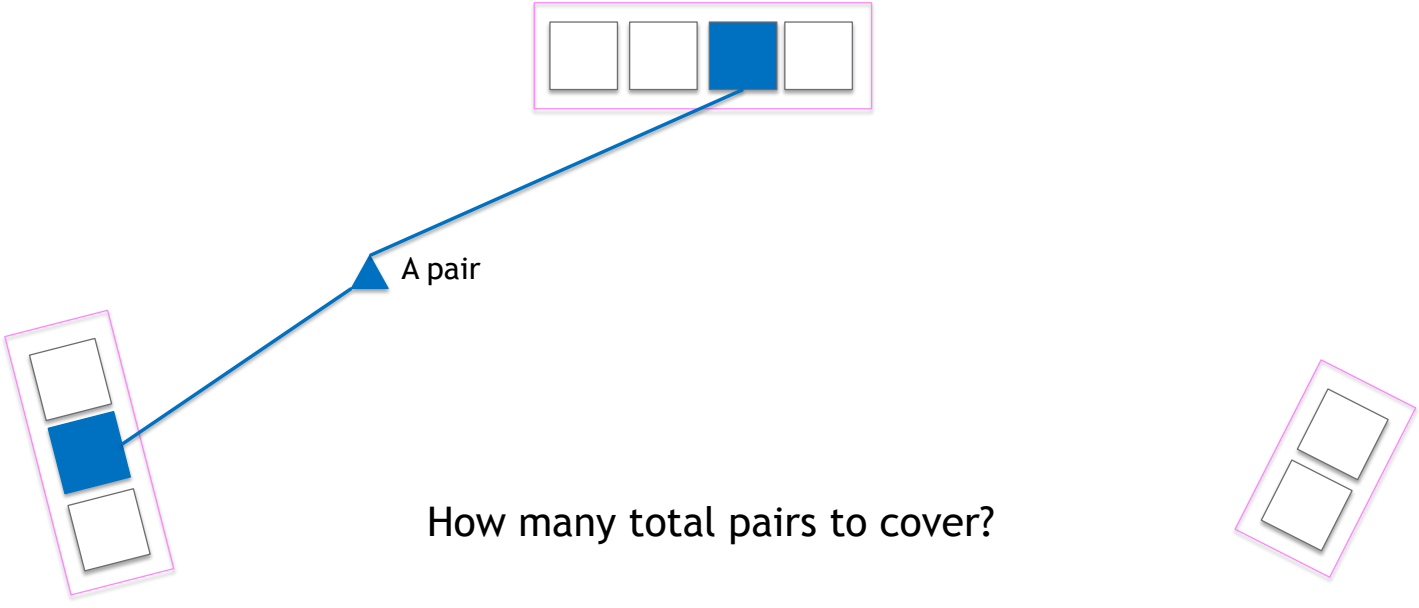
If we were to apply All-Choice



How many test cases with AC?



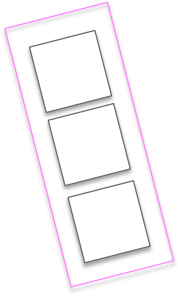
If we were to apply All-Pairs



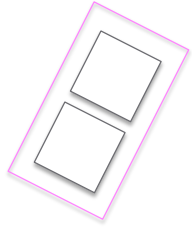
How many total pairs to cover?

All-Choice vs. All-Pairs

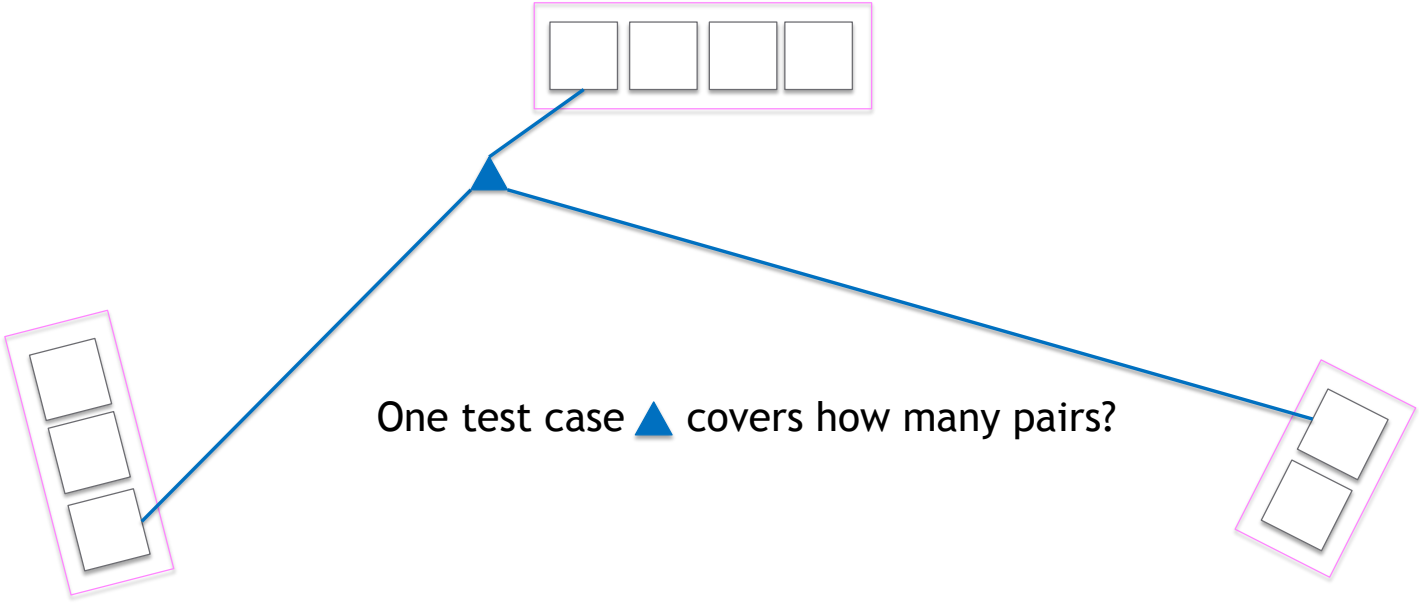
4 x 3 x 2 = 24 test cases with AC
For AP must cover 4 x 3 + 4 x 2 + 3 x 2 = 26 pairs



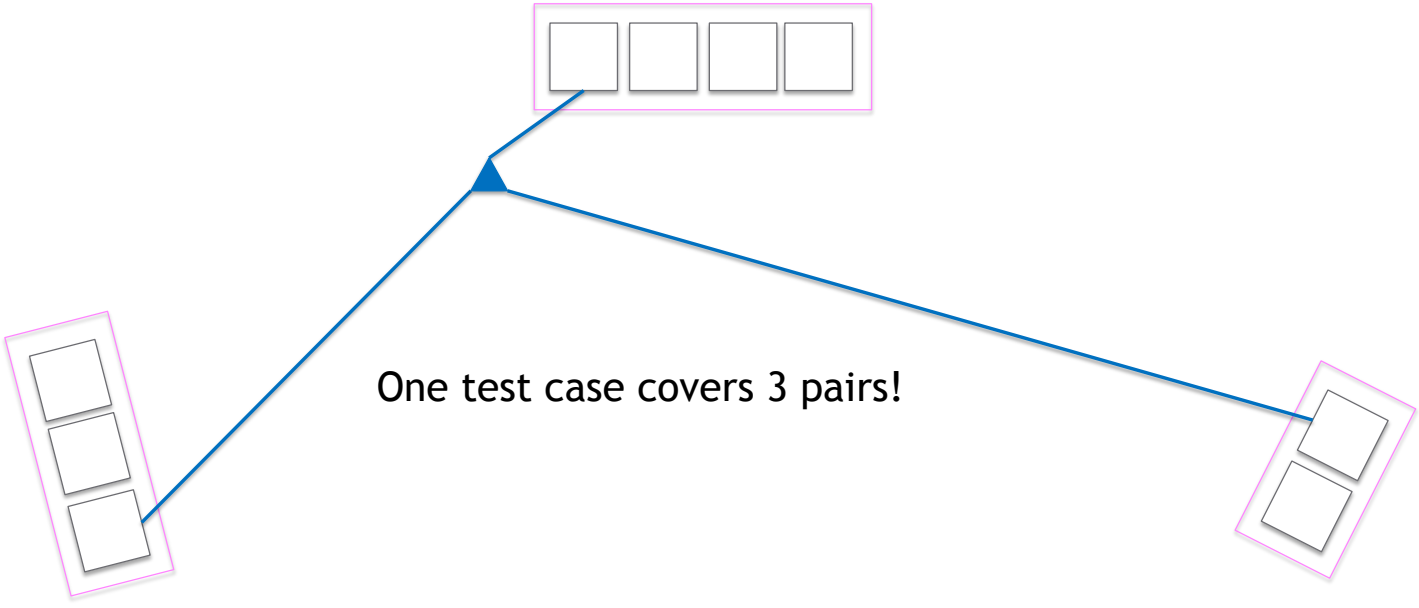
But do we need 26 test cases for AP?



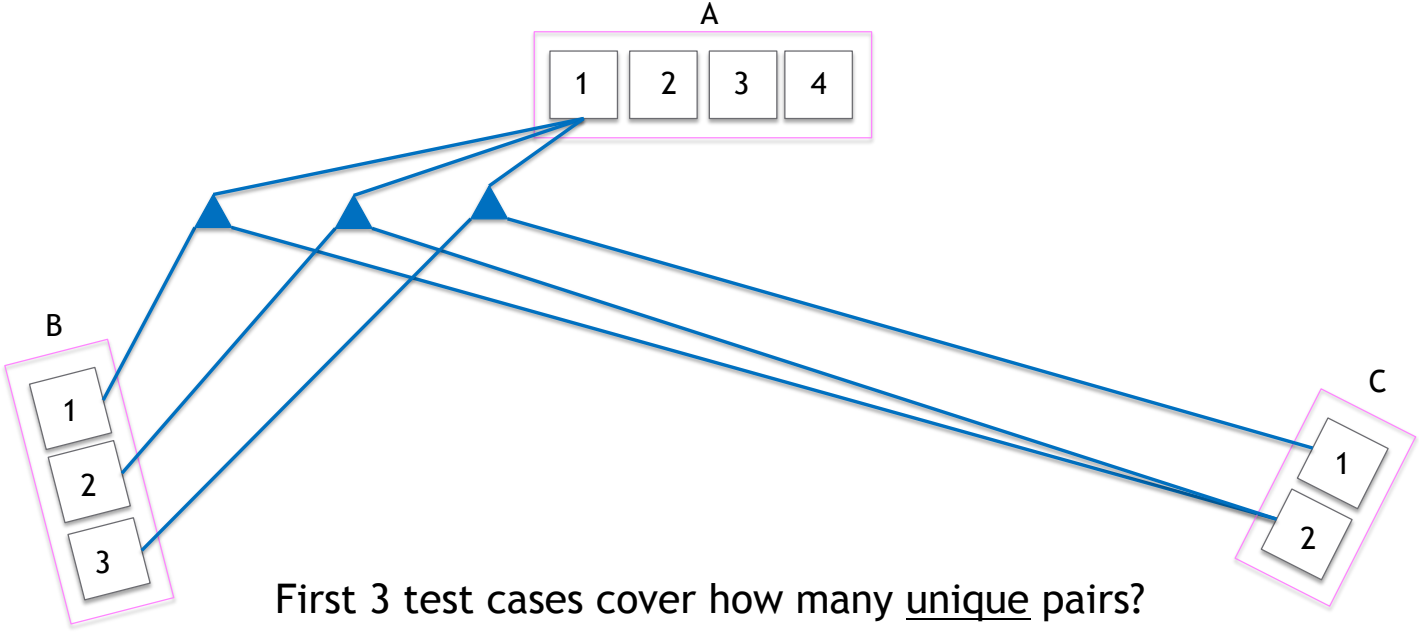
All-Pairs: how can we optimize?



All-Pairs: how can we optimize?

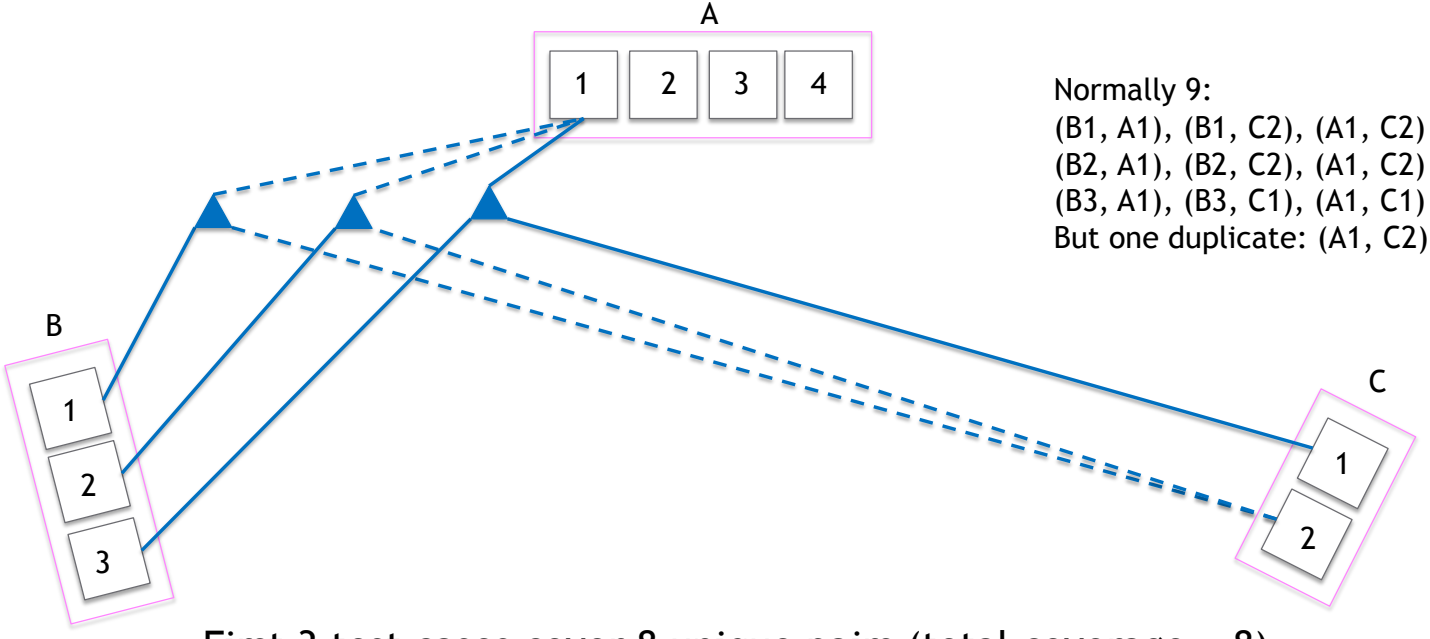


All-Pairs: 3 test cases



First 3 test cases cover how many unique pairs?

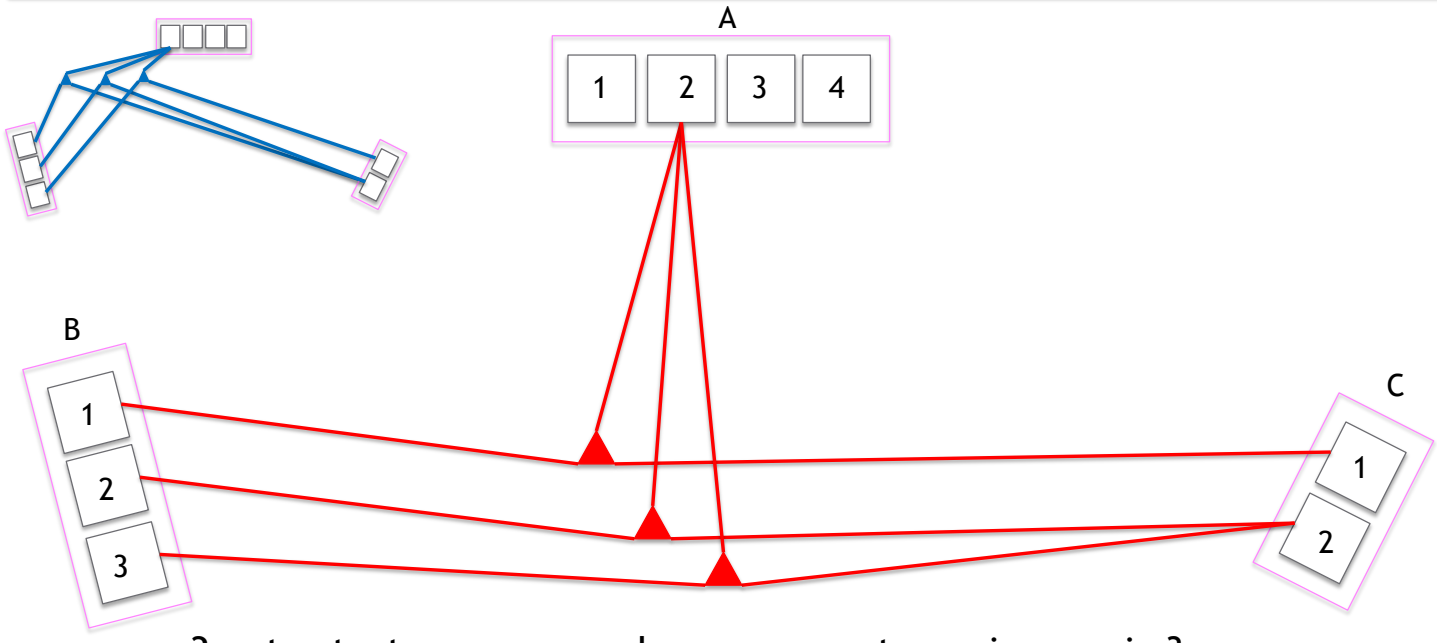
All-Pairs: 3 test cases



Normally 9:
(B1, A1), (B1, C2), (A1, C2)
(B2, A1), (B2, C2), (A1, C2)
(B3, A1), (B3, C1), (A1, C1)
But one duplicate: (A1, C2)

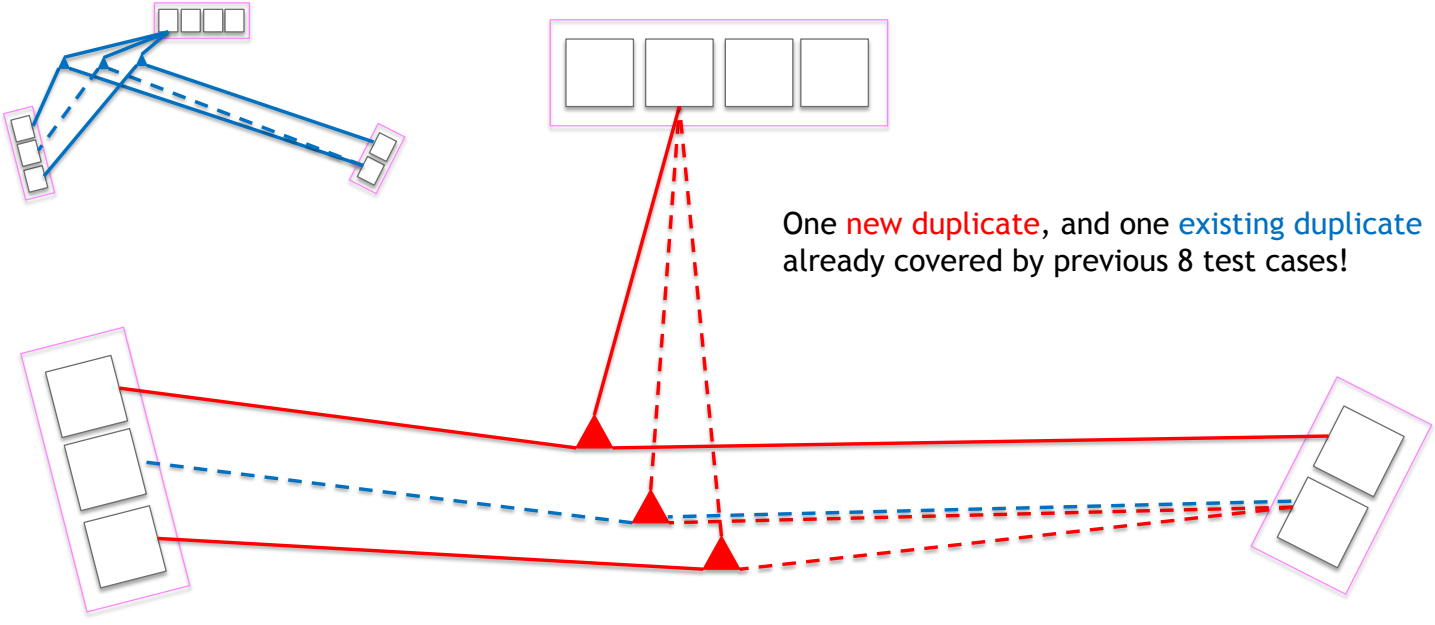
First 3 test cases cover 8 unique pairs (total coverage = 8)

All-Pairs: 6 test cases



3 extra test cases cover how many extra unique pairs?

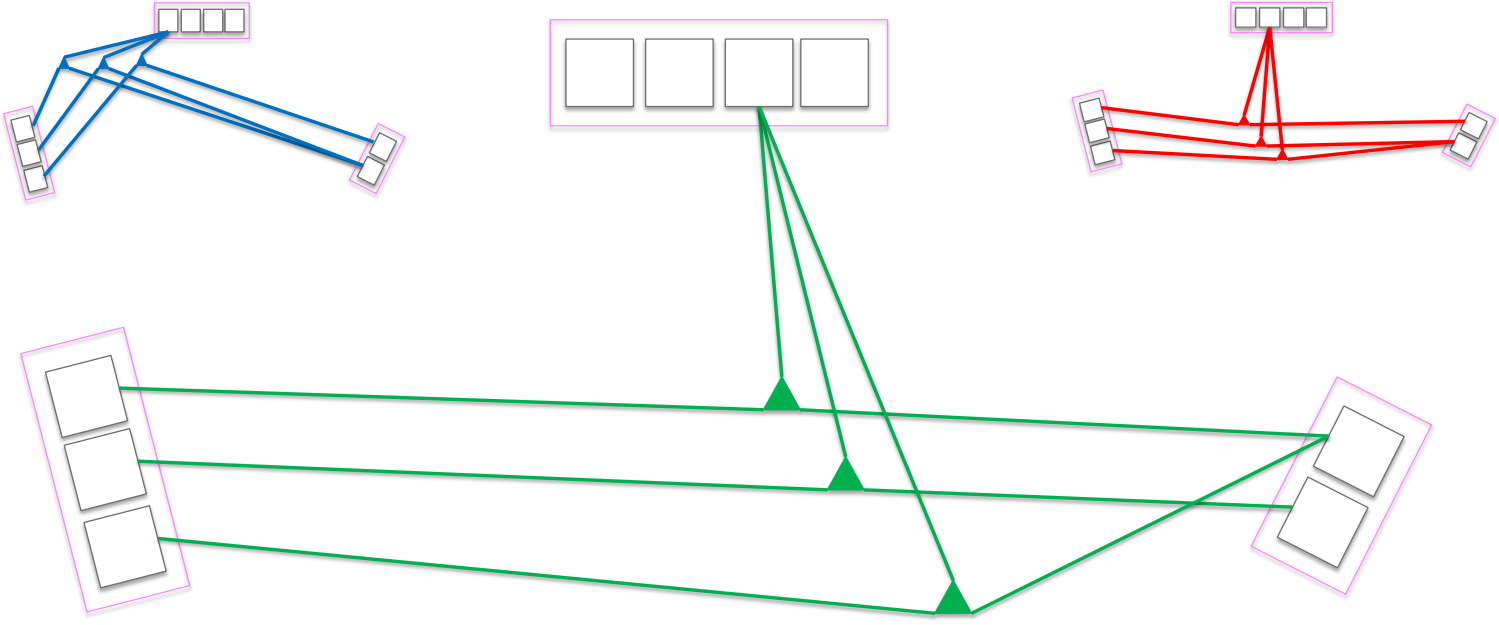
All-Pairs: 6 test cases



One new duplicate, and one existing duplicate already covered by previous 8 test cases!

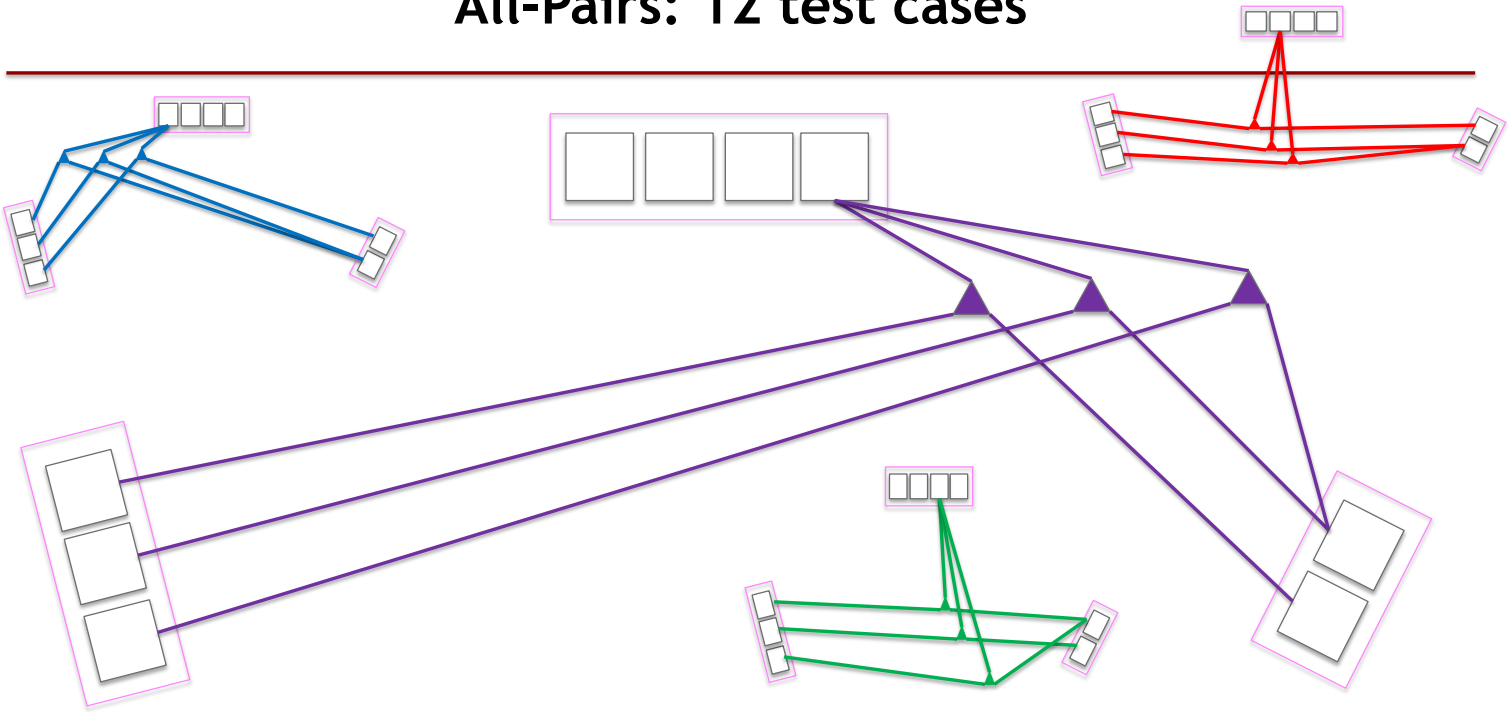
3 extra test cases cover 7 extra unique pairs (total coverage 15)

All-Pairs: 9 test cases



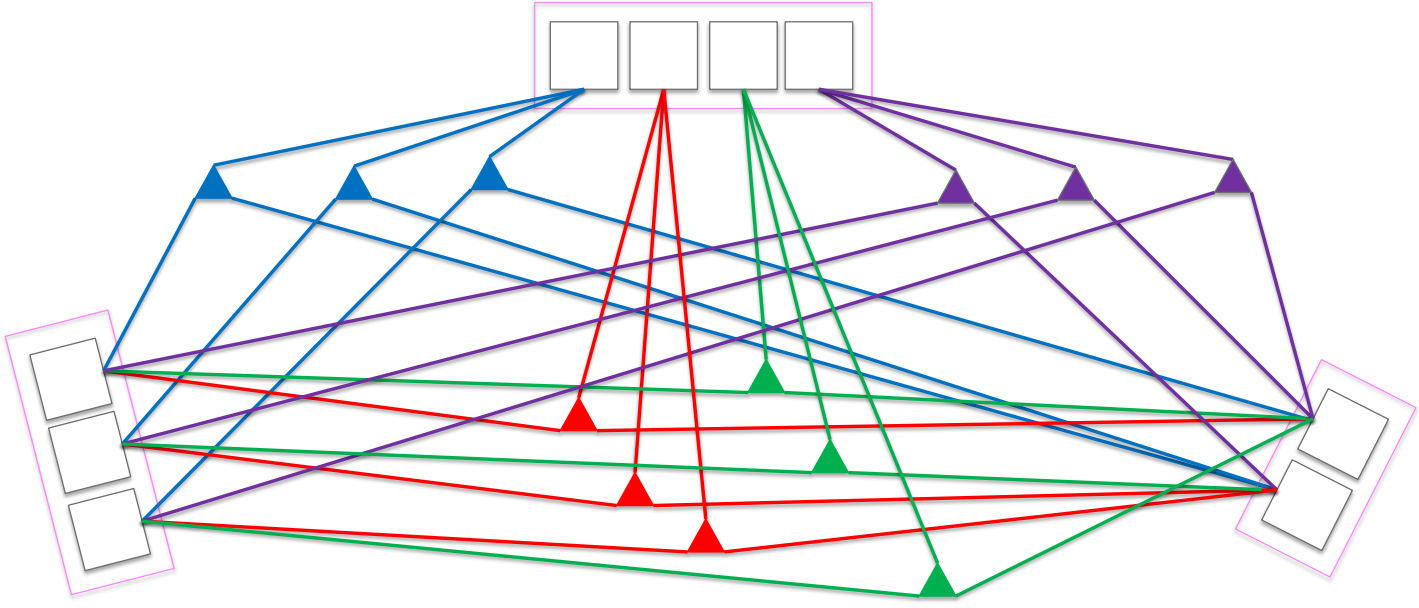
3 extra test cases cover 5 more unique pairs (total coverage 20)

All-Pairs: 12 test cases



3 extra test cases cover 6 more unique pairs (**total coverage 26 – all pairs**)

All-Pairs



With AP, we have 12 test cases (not 26) compared to 4 x 3 x 2 = 24 for AC

What's special about All-Pairs?

- Empirically supported
- Drastically reduces number of test cases
- Strong recommendation from NIST
- Well-received in the agile methods community
- Supported by commercial products

The NIST findings

- A *post-mortem* analysis of software-controlled medical systems

  "98% of the reported software defects in recalled medical devices could have been detected by testing all pairs of parameter settings"

  Empirically: pair-wise characteristic interactions are a major source of faults

D.R. Wallace, D.R. Kuhn, *"Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," Intl. Journal of Reliability, Quality, and Safety Engineering, vol. 8, no. 4. (2001)*

Efficient testing with All-Pairs

- Bernie Berger, STAREast 2003 International Conference on Software Testing
- Twelve variables (each is an attribute converted directly to a characteristic), with varying numbers of values (blocks), have

  7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2 = 725,760

  combinations of values (test cases) and

  66 ways to chose pairs of variables to test
- "All-Pairs can do the job effectively with just 50 test cases"

  Why?

Efficient testing with All-Pairs

- Bernie Berger, STAREast 2003 International Conference on Software Testing
- Twelve variables (each is an attribute converted directly to a characteristic), with varying numbers of values (blocks), have

  7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2 = 725,760

  combinations of values (test cases) and

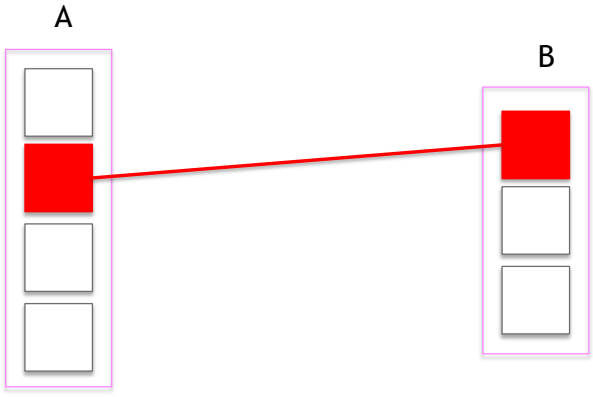
  66 ways to chose pairs of variables to test
- "All-Pairs can do the job with just 50 test cases"
  - Because we assume interactions among more than two characteristics aren't as likely to cause faults
  - Because each test case covers more than one pairwise interaction (pair of values) at a time
    - In this example, each test case covers at most C(12, 2) = 66 pairs

Choosing combinations of blocks on base cases: BC

- Testers sometimes recognize that certain blocks are most “ordinary”, most “normal”
- This uses domain knowledge of the program
- *Isolate non-ordinary blocks within a characteristic one at a time to test*

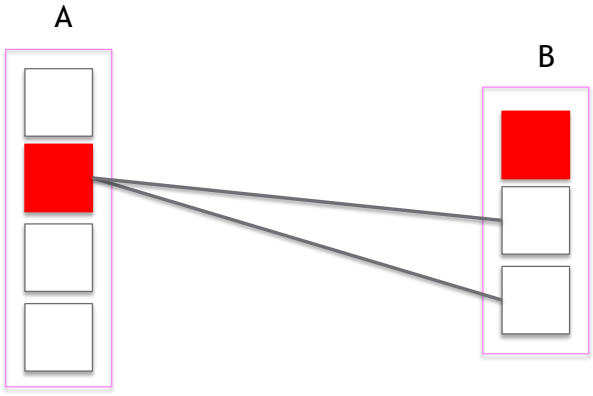
Base-Choice (BC) : A base-choice block is chosen for each characteristic, and a base case is formed by using the base choice for each characteristic. Subsequent test cases are chosen by holding *ALL but one* base choice constant and varying the remaining characteristic to cover non-base-choice blocks in that characteristic.

Base-Choice: Base Case



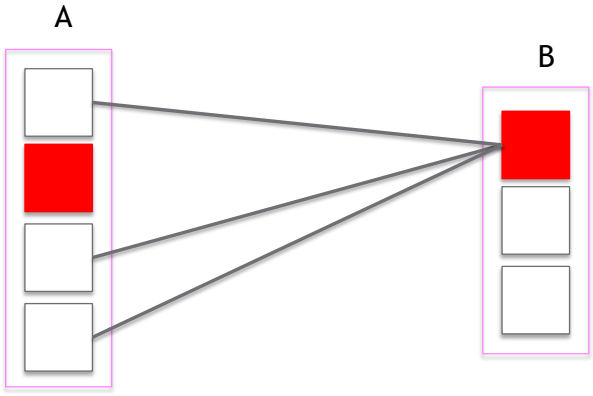
1 Base Case

Base-Choice: cases for B



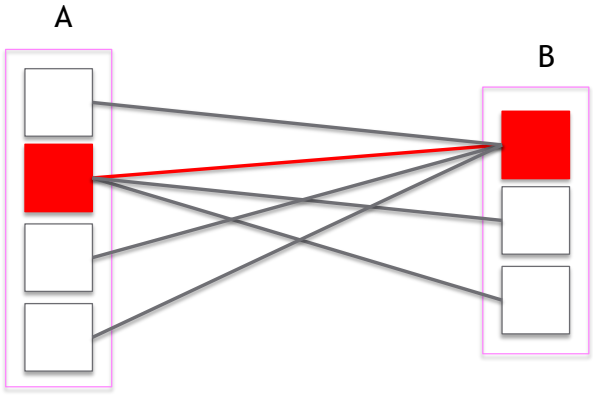
Fix A, vary B: 2 more test cases

Base-Choice: cases for A



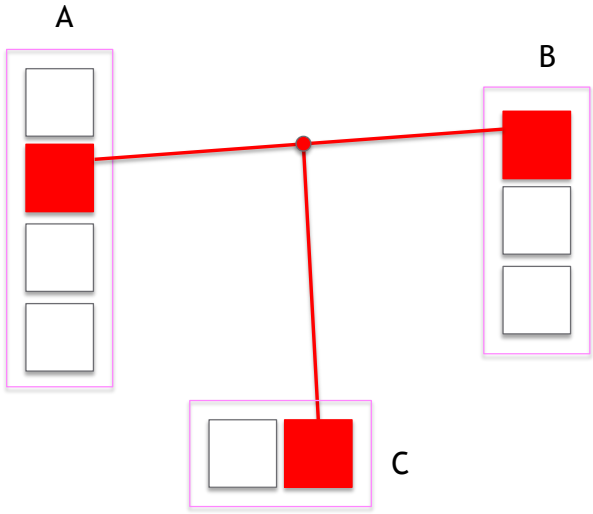
Fix B, vary A: 3 more test cases

Base-Choice: Total Cases



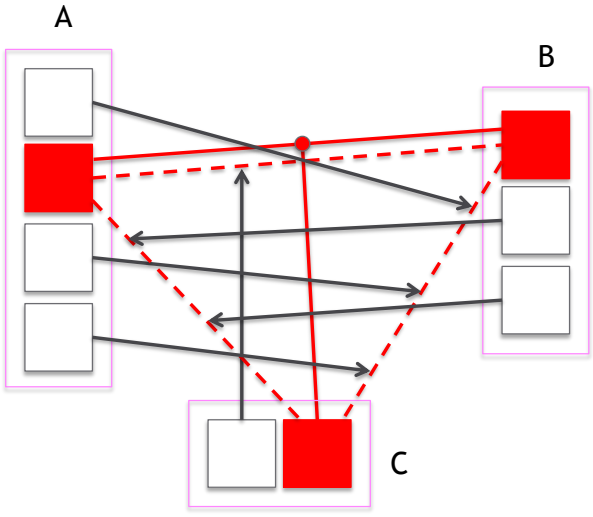
1 (Base Case) + 2 + 3 = 6 test cases compared to 12 for AC

Base Choice – 3 characteristics



1 (Base Case) + 3 + 2 + 1 = 7 test cases compared to 24 for AC, why?
Homework: think about it?

Base Choice – 3 characteristics



1 (Base Case) + 1 + 2 + 3 = 7 test cases instead of 24 AC

Base choice selection

- The base case must be feasible
  - That is, base choices must be compatible/combinable
- Base choice blocks can be
  - most likely (nominal)
  - ordinary classes of values or “happy-path” triggers
  - simplest
  - most obvious
  - smallest
  - “first” in some ordering
- The base choice is a *crucial design* decision
  - Test designers should justify why the choices were made

BC can be generalized to MBC

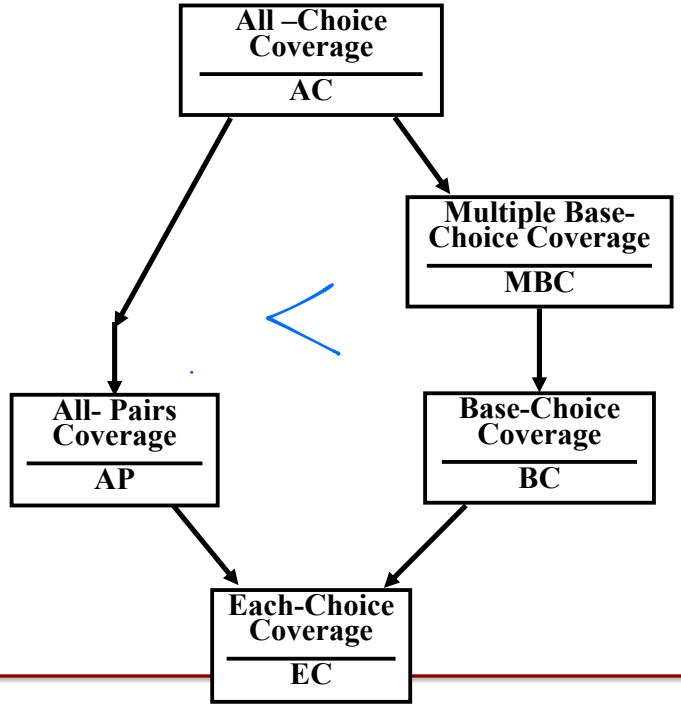
Testers sometimes identify more than one logical base choice ("ordinary" block) for each characteristic

Multiple Base Choice (MBC) : One or more base-choice blocks are chosen for each characteristic, and base cases are formed by using each base choice for each characteristic. Subsequent tests are chosen by holding ***ALL but one*** base choice constant for each base case and varying the non-base-choice blocks in the remaining characteristic.

Example in Appendix

An input space model coverage criterion can subsume another criterion

A lattice!



Constraints among characteristics reduce number of combinations

- Some combinations of blocks are infeasible
  - "less than zero" and "scalene" … not possible at the same time
- These are represented explicitly as *constraints* among blocks
- Two general types of constraints
  - **A block from one characteristic *cannot* be combined with a specific block from another**
  - **A block from one characteristic can *only* be combined with a specific block from another characteristic**
- Handling infeasible combinations depends on the criterion used
  - AC, AP, EC: drop the infeasible combinations
  - BC, MBC: re-evaluate base choices, then drop infeasible combinations if necessary

TriTyp: Constraints

	Block			
Characteristic	b_1	b_2	b_3	b_4
q_0 = "Geometric Classification" *semantic*	scalene	isosceles, not equilateral	equilateral	invalid
	q_1, q_2, q_3 must be positive; q_4 must be other			
q_1 = "Positivity of a"	non-positive	positive		
q_2 = "Positivity of b"	non-positive	positive		
q_3 = "Positivity of c"	non-positive	positive		
q_4 = "Validity of Positive Inputs" *semantic*	invalid: a + b <= c	invalid: b + c <= a	invalid: a + c <= b	other
	q_1, q_2, q_3 must be positive			

Not very easy upfront, but you'll run into constraints when generating the test inputs!

Combinatorial strategies summary

- Creating test cases for all-combinations may not be feasible
- Optimal combinatorial strategy depends on the EUT
- Simplest strategies may not be very effective
- Compromise strategies like All Pairs (AP) and Base Choice (BC) maybe very effective
  - domain information about interactions among characteristics may allow us to remove combinations unlikely to represent possible faulty interactions (BC) or sample the input space model in a more targeted way by focusing only on pairwise interactions (AP)

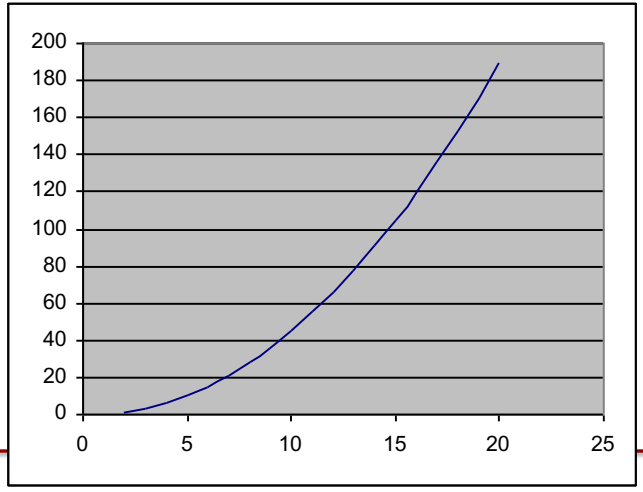
Appendix

More on All-Pairs Combinatorial Testing

The NIST Example Explained

Combinatorial explosion

- All Pairs Testing seeks to alleviate combinatorial explosion
- Normally, a program with n characteristics will have ${}_nC_2 = n!/(2!(n-2)!)$ different pairs to test if we were to choose all possible pairwise attribute interactions



Total number of test cases: if each characteristic has m possible blocks, multiply this by m^2 to cover all interactions

… if each test case covered only one pair-wise interaction

Efficient testing with All-Pairs

- Bernie Berger, STAREast 2003 International Conference on Software Testing
- Twelve variables (each is an attribute converted directly to a characteristic), with varying numbers of values (blocks), have

  7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2 = 725,760

  combinations of values (test cases) and

  66 ways to chose pairs of variables to test
- "All-Pairs can do the job with just 50 test cases"

  Why?

Efficient testing with All-Pairs

- Bernie Berger, STAREast 2003 International Conference on Software Testing
- Twelve variables (each is an attribute converted directly to a characteristic), with varying numbers of values (blocks), have

  7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2 = 725,760

  combinations of values (test cases) and

  66 ways to chose pairs of variables to test
- "All-Pairs can do the job with just 50 test cases"
  - Because each test case covers more than one interaction (pair of values) at a time

Online Mortgage Application

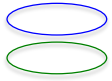
(courtesy of James Bach)

Variables (Attributes)

Values (Blocks)	Region	Tier	Property	Credit	Residence	LTV	NIV	NAV	Refinance	Closing Cost	Intro Rate	Bank Emp
	NY	L	1 fam	A+	Pri	80%	Yes	Yes	Yes	Cust	Yes	Yes
	NJ	M	2 fam	A	Vac	90%	No	No	No	Bank	No	No
	FL	H	3 fam	A-	Inv	100%						
	TX	H+1	4 fam	B								
	CA	H+2	Coop	<B								
	DC	H+3	Condo									
	Other											

7 x 6 x 6 x 5 x 3 x 3 x 2 x 2 x 2 x 2 x 2 x 2
= 725,760 possible combinations

Selected All-Pairs test cases



6 Pairs overlapping between cases 1 & 3

6 Pairs overlapping between cases 2 & 3

$6 = {}_4C_2$

case	Region	Tier	Property	Credit	Residence	LTV	NIV	NAV	Refinance	ClosingCost	IntroRate	BankEmp	pairings
1	NY	L	1 fam	A+	Pri	80%	Yes	Yes	Yes	Cust	Yes	Yes	66
2	NY	M	2 fam	A	Vac	90%	No	No	No	Bank	No	No	66
3	NJ	L	2 fam	A-	Inv	100%	Yes	No	Yes	Bank	Yes	No	54
4	NJ	M	1 fam	B	Pri	90%	No	Yes	No	Cust	No	Yes	46
47	NY	H+1	~Coop	B	~Inv	~80%	~No	~No	~Yes	~Cust	~Yes	~Yes	1
48	Other	H	2 fam	~B	~Vac	~100%	~No	~No	~Yes	~Cust	~No	~No	1
49	Other	H+2	3 fam	~A-	~Pri	~80%	~Yes	~No	~No	~Cust	~No	~No	1
50	Other	H+1	Condo	<B	~Vac	~90%	~No	~No	~Yes	~Cust	~No	~No	2

Normally would need between 7 x 6 and 66 x 7 x 6 test cases

New pairings not covered by previous test cases

Efficiency vs. fault localization

- The efficiency comes at the expense of fault isolation. There are ${}_{12}C_2$ = 12!/(2! x (12 - 2)!) = 66 pairs in the first row of the table
- If that test case fails, which pairwise interaction (of the 66 pairs) caused the failure?
- Good strategy only for regression testing ("is anything broken? yes/no" vs. "what is broken?")

All-Pairs test generation tools

- ~~Automatic Efficient Test Generator (AETG)~~
  - commercial tool
  - approximate cost? $7,500

- James Bach's program: allpairs
  - free at: http://www.satisfice.com/tools.shtml
  - input is a text file

- Other tools: http://www.pairwise.org/tools.asp

Summary of the popular story

Excerpt from Bach and Schroeder: "Pair-wise testing: the best practice that isn't"

- "Pairwise testing protects against pairwise bugs"
- "while dramatically reducing the number of tests to perform"
- "which is especially cool because pairwise bugs represent the majority of combinatorial bugs"
- "and such bugs are a lot more likely to happen than the ones that only happen with more variable."
- "Plus, the availability of tools means you no longer need to create these tests by hand."

Complete story for All-Pairs testing

- "Pairwise testing **might find** some pairwise bugs"
- "while dramatically reducing the number of tests to perform, **compared to testing all combinations, but not necessarily compared to testing just the combinations that matter,"**
- "which is especially cool because pairwise bugs represent the majority of combinatorial bugs, **or might not, depending on the actual dependencies among the variables in the product**,"
- "and such bugs are more likely to happen than ones that only happen with more variables, **or less likely to happen, because user inputs are not randomly distributed."**
- "Plus, the availability of tools means you no longer need to create these tests by hand, **except for the work of analyzing the product, selecting variables and values, actually configuring and performing the tests, [choosing oracles,] and analyzing the results."**

All-Pairs assumptions

Pure AP testing:

- Characteristics/attributes have clear *uniform* blocks that represent true equivalence classes
- Characteristics/attributes are independent
- Failures are the result of the interaction of a pair of characteristics/attributes

All-Pairs - Reminder

A value from each block for each characteristic is combined with a value from every block for each other characteristic

Each combination is covered by at least one test case

Each test case typically covers several combinations

All-Pairs: the `allpairs` program

A free AP test case generator

Available at:

https://www.satisfice.com/download/allpairs

All-Pairs example: TriTyp revisited

- sides *a*, *b*, and *c* are integers (characteristics)
- Blocks (equivalence classes) for each side
  - {x: x is an integer and x < 1 } invalid values
  - {x: x is an integer and 1 <= x <= 200 } valid values
  - {x: x is an integer and x > 200 } invalid values

Input to allpairs

Attributes/characteristics

Partitions	side *a*	side *b*	side *c*
	a < *1*	*b* < *1*	*c* < 1
	1<= *a* <= 200	1<= *b* <= 200	1<= *c* <= 200
	a > 200	*b* > 200	*c* > 200

Normally 3 x 3 x 3 = 27 combinations

Test cases generated by allpairs (without the oracle)

case	side a	side b	side *c*	new pairings
1	*a* < 1	*b* < 1	*c* < 1	3
2	*a* < 1	1<= *b* <= 200	1<= *c* <= 200	3
3	a < 1	*b* > 200	*c* > 200	3
4	1<= *a* <= 200	*b* < 1	1<= *c* <= 200	3
5	1<= *a* <= 200	1<= *b* <= 200	*c* < 1	3
6	1<= *a* <= 200	*b* > 200	*c* < 1	2
7	*a* > 200	*b* < 1	*c* > 200	3
8	*a* > 200	1<= *b* <= 200	*c* < 1	2
9	*a* > 200	*b* > 200	1<= *c* <= 200	3
10	1<= *a* <= 200	1<= *b* <= 200	*c* > 200	2

Test cases generated by allpairs (with representative values and oracle)

case	side *a*	side *b*	side *c*	Expected Output
1	-3	-2	-4	Not a Triangle
2	-3	5	7	Not a Triangle
3	-3	201	205	Not a Triangle
4	6	-2	9	Not a Triangle
5	6	5	-4	Not a Triangle
6	6	201	-4	Not a Triangle
7	208	-2	205	Not a Triangle
8	208	5	-4	Not a Triangle
9	208	201	7	Not a Triangle
10	6	5	205	Not a Triangle

?

What happened?

Used a purely syntactic characteristics…

- Cannot choose values (the algorithm does it!)
- Cannot specify dependencies between attributes/characteristics (constraints)
  - *Invalid triangle:* $(a \geq b + c) \vee (b \geq a + c) \vee (c \geq a + b)$
- Ended up exercising only data validity, not other functionality
  - dependencies may generate too many invalid/impossible/infeasible combinations if not explicitly handled
- Easy, but not effective in this instance

What to do?

What to do?

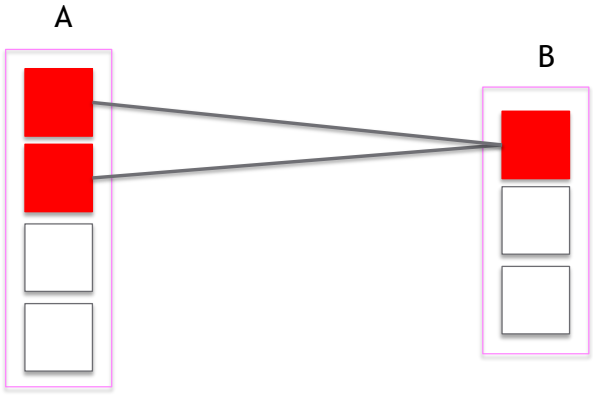
Have semantic characteristics!

Have constraints!

Combine with other partition-testing strategies!

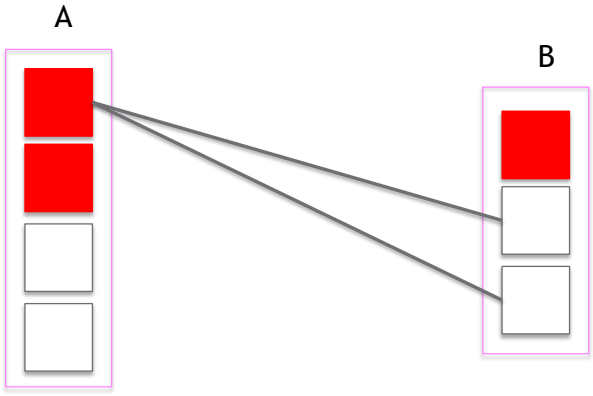
Multiple Base Choice Example

Multiple Base Choice



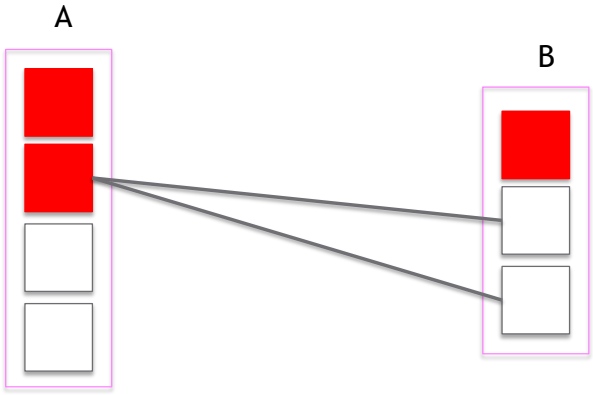
2 x 1 Base Cases

Multiple Base Choice



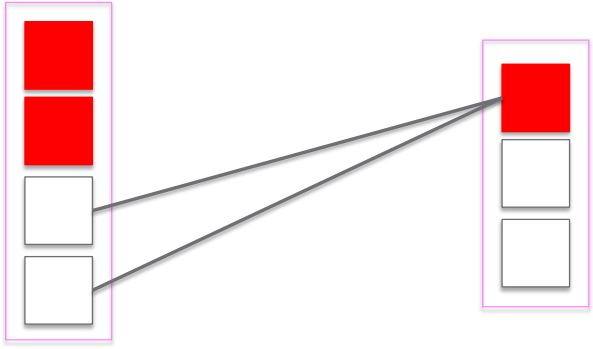
Fix 1st BC of A, vary B: 2 more test cases

Multiple Base Choice



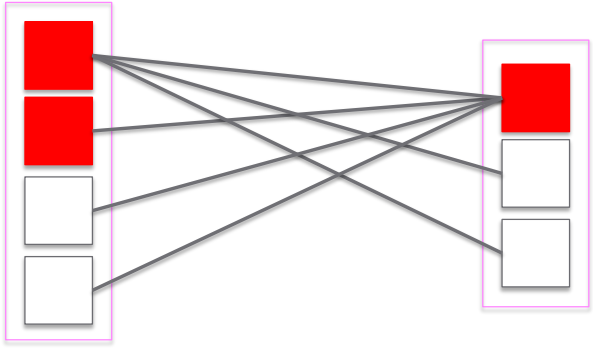
Fix 2nd BC of A, vary B: 2 more test cases

Multiple Base Choice



Fix B, vary A: 2 more test cases

Multiple Base Choice



2 (Base Cases) + 2 + 2 + 2 = 8 test cases instead of 12 AC

Sources

- Pezze & Young, *Software Testing & Analysis*
- Jorgensen, *Software Testing: A Craftsman's Approach*
- Ammann & Offutt: *Introduction to Software Testing*

Several illustrations and examples have been adopted from these sources and accompanying instructor materials, with various adaptations