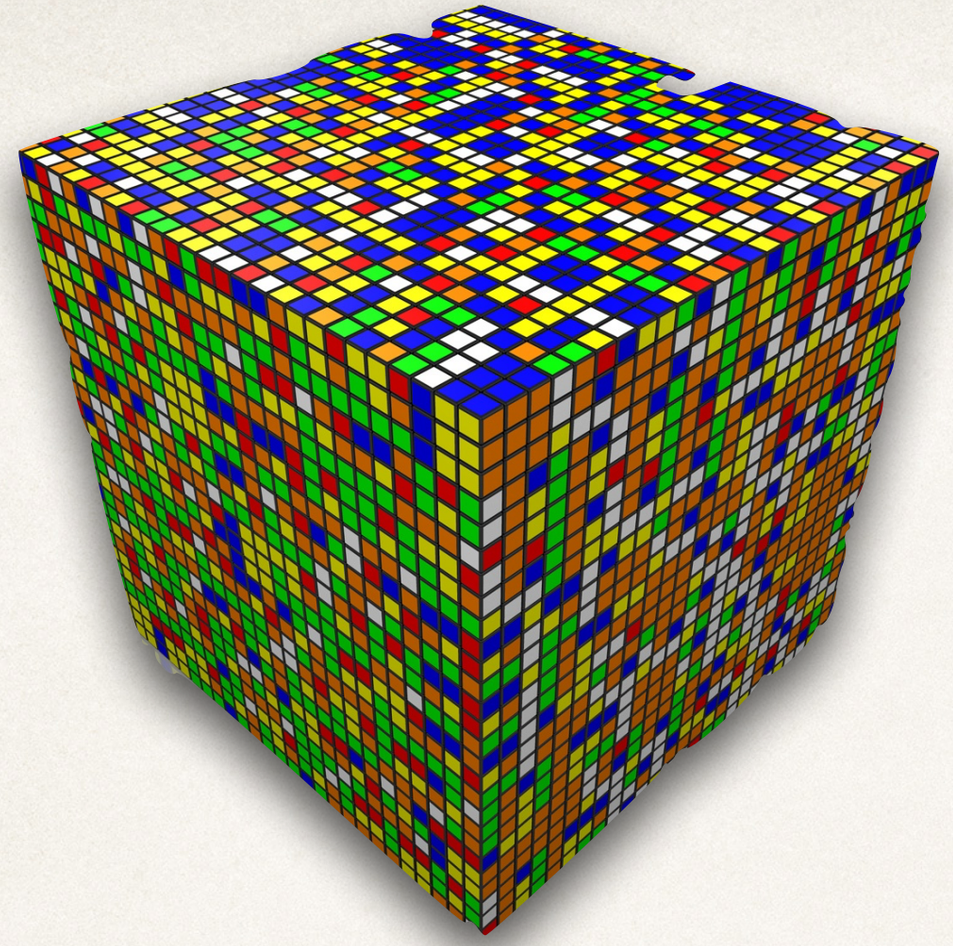


Combinatorial Interaction Testing

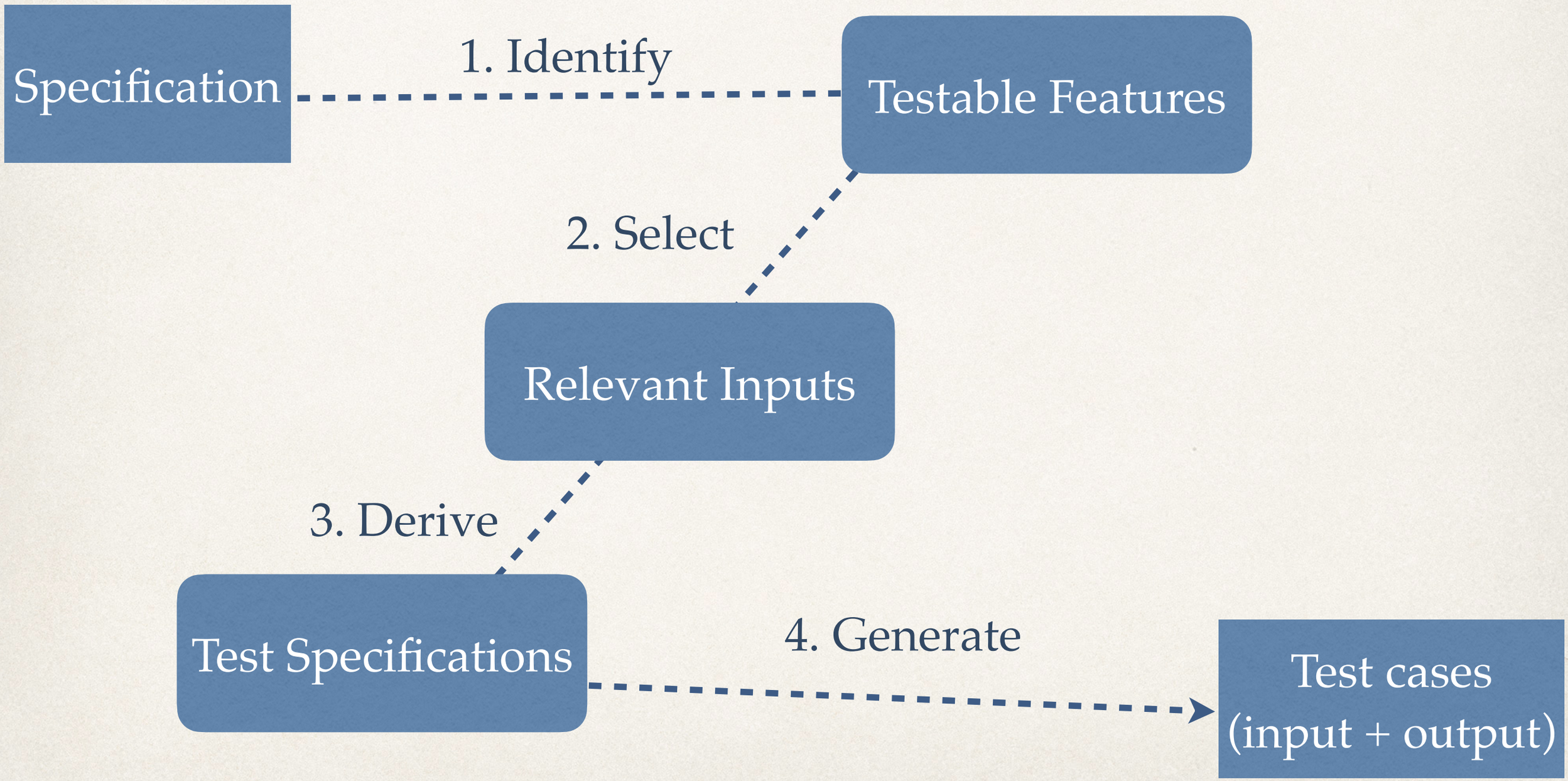
COMP0103-A7P
COMP0103-ATU

Validation & Verification

Slides are used with permission of Dr Jia and Petka, and based on the following material: http://cse.unl.edu/~citportal/, http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf, Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

f.sarro@ucl.ac.uk

Recap: Black-box Testing



Combinatorial Interaction Testing (CIT)

UCL

Specification

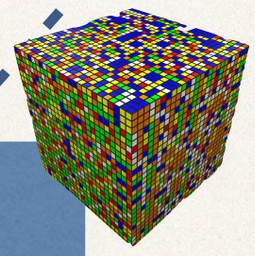
1. Identify

Testable Features

2. Select

Relevant Inputs

3. Derive



CIT

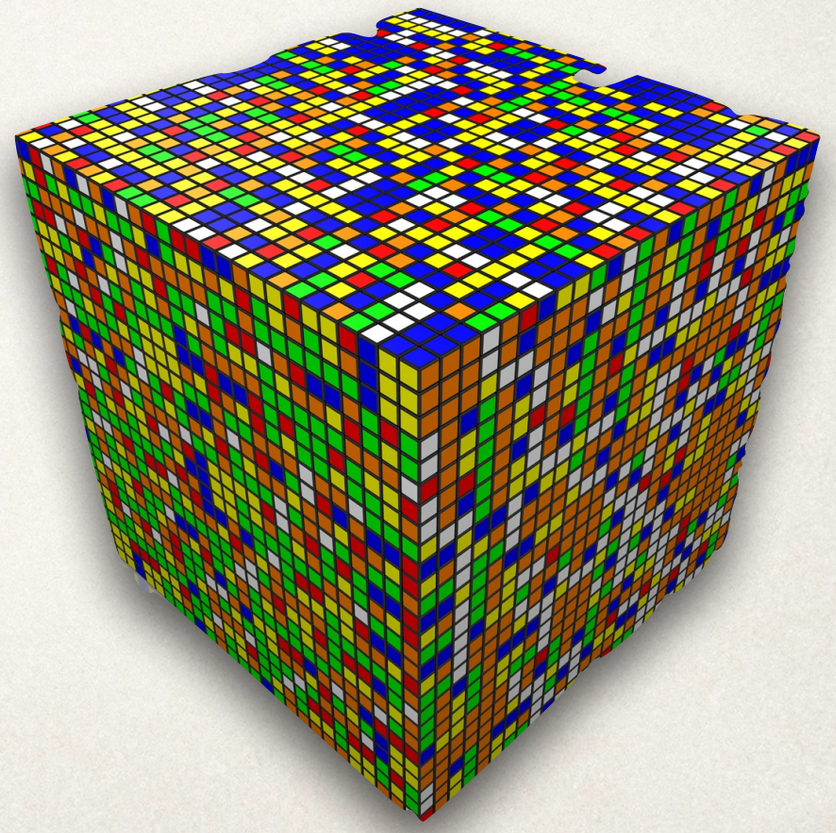
Test Specifications

4. Generate

Test cases
(input + output)

Outlines

- Combinatorial Interaction Testing (CIT)
  - Motivation
  - History
  - Pairwise and t-way testing
  - Approaches for generating t-way interaction test suites



CIT: Motivation

http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=910001

UCL

What causes software failures?

- Logic errors?
- Calculation errors?
- Interaction faults?
- Inadequate input checking?

Interactions Involved in Software Failures

http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=910001

UCL

How does an interaction fault manifest itself in code?

```
if (pressure < 10) {
    // do something

    if (volume > 300)  {
        faulty code!  BOOM!
      }
     else { good code, no problem}

} else {
    // do something else
}
```

pressure < 10 & volume > 300
(2-way interaction)

A test that includes pressure = 5 and volume = 400 would trigger a failure

CIT: Motivation

UCL

- The behaviour of a software application can be affected by many factors, e.g., input parameters, environment configurations, state variables

- Techniques like equivalence partitioning and boundary-value analysis can be used to identify the possible values of each factor

- It is impractical to test all possible combinations of values of all those factors

CIT: Motivation

- Modern software systems have many inputs and highly configurable system have hundreds of parameters
  - Combinatorial Explosion!
- Can we reduce them further?
  - Combinatorial Design!

CIT: Combinatorial Design

UCL

- A subset of combinations is generated to satisfy some well-defined combination strategies (instead of testing all possible combinations)

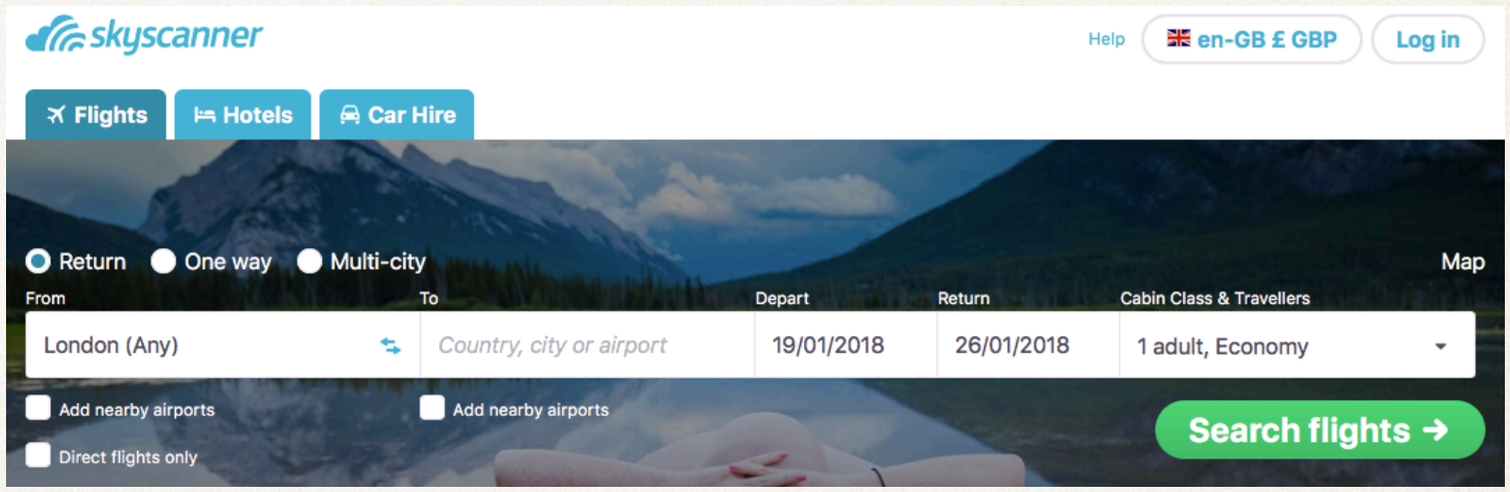
- Key observation: not every factor contributes to every fault, and it is often the case that a fault is caused by interactions among a few factors

- Combinatorial design can dramatically reduce the number of combinations to be covered and remain very effective in terms of fault detection

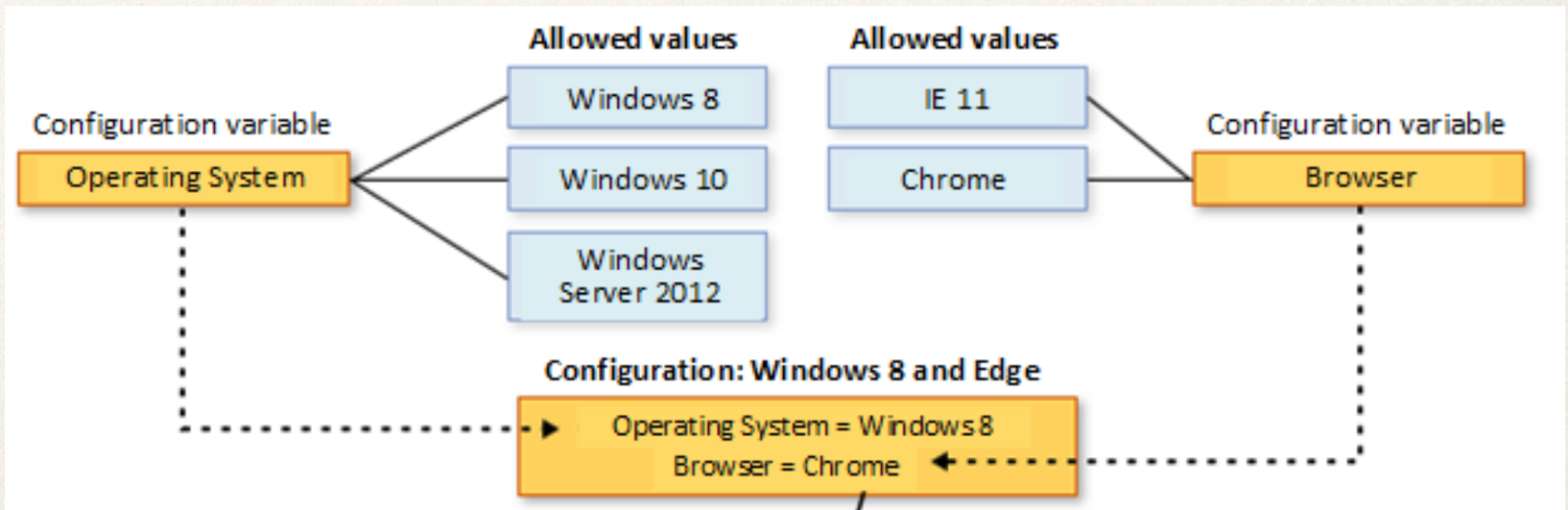
Two Ways of Using CIT

UCL

Test data inputs



Test configurations



https://docs.microsoft.com/en-us/vsts/manual-test/mtm/test-configurations-specifying-test-platforms

CIT Applications

UCL

- Web Forms
- Web Browsers
- Automotive Industry
- Cellphone Industry
- .. and many other configurable systems

History of Combinatorial Testing

HISTORY OF SOFTWARE TESTING

1950 1960 1970 1980 '85 1990 '95 2000 '05 2010

(1957: Decision table)

~1967: Equivalence partitioning, Boundary value analysis

1970: Cause Effect Graph [Elmendorf]

Combinatorial techniques

Fujitsu

(1983~)1984: Orthogonal array / Combination Table [Satoh, Shimokawa]

1987: Test Case Design Support System [Tatsumi]

(199x~)2004: Orthogonal Array (HAYST method) [Akiyama(Fuji Xerox)]

1998: IPO [Lei, Tai]

2007: FireEye 2009:ACTS (IPOG) [Lei, Kuhn]

(1983~)1985: Orthogonal Latin Squares [R. Mandl]

2000: CTE XL [Daimler Chrystler]

AT&T, Bellcore

(198x~)1992: OATS [Brownlie, Prowse, Phadke]

(1990~)1994: CATS [Sherwood]

2000: Covering arrays [Williams]

(1992~)1994: AETG [Cohen, et. al]

(2000~)2004: PICT [Microsoft]

Input condition analysis techniques

(1976: Test Factor Analysis Method) [Fujitsu]

1988: Category-partition method [Ostrand, Balcer]

1993: Classification-tree method [Grochtmann, Grimm]

HISTORY OF SOFTWARE TESTING

History of CIT

UCL

Combinatorial testing for software has its roots in the statistical field of Design of Experiments (DoE)

source: https://en.wikipedia.org/wiki/The_Design_of_Experiments#/media/File:R._A._Fischer.jpg

Ronald Fisher was an english statistician whose work included a book “Design of Experiments” (1935) that serves as the basis for many of the principles of CIT.

History of CIT

UCL

The lady tasting tea

https://en.wikipedia.org/wiki/Lady_tasting_tea

Ms Muriel Bristol claimed to be able to tell whether the tea or the milk was added first to a cup.

Fisher proposed to give her eight cups, four of each variety, in random order. One could then ask what the probability was for her getting the specific number of cups she identified correct, but just by chance.

History of CIT

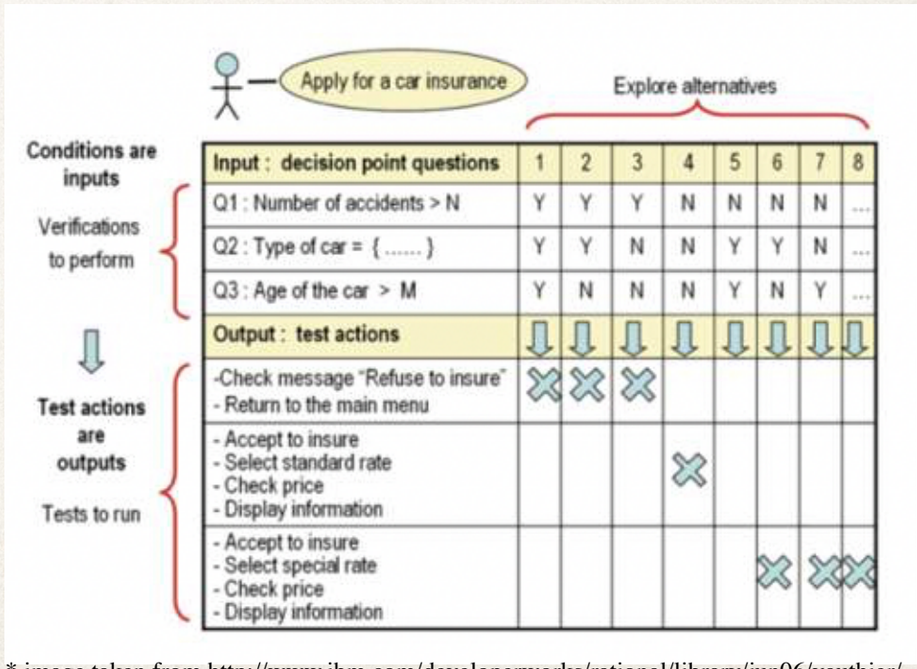
UCL

Decision tables

Initially developed by General Electric and Sutherland Corp. (independently in 1957) as a means to express product design logic, operational planning, management decision rules.

	Rules					
Conditions	1	2	3	4	5	6
C1. Address proof provided	N		Y	Y	Y	Y
C2. Identity proof provided		N	Y	Y	Y	Y
C3. Loan amount < monthly salary			Y			
C4. Loan amount >= monthly salary					Y	Y
C5. Loan purpose				Home purchase	Pay tax	Other
Actions	1	2	3	4	5	6
A1. Approve loan request immediately			X		X	
A2. Review loan request manually				X		X
A3. Reject loan request	X	X				

https://www.visual-paradigm.com/tutorials/align-business-goal-business-logic-decision-table.jsp



* image taken from http://www.ibm.com/developerworks/rational/library/jun06/vauthier/

History of CIT

UCL

Orthogonal Arrays

An (n x k) array A with entries from some set S with s levels, strength t within the range $0 \leq t \leq k$ and index λ where every N x t subarray of A contains each t-tuple based on S exactly λ times as a row.

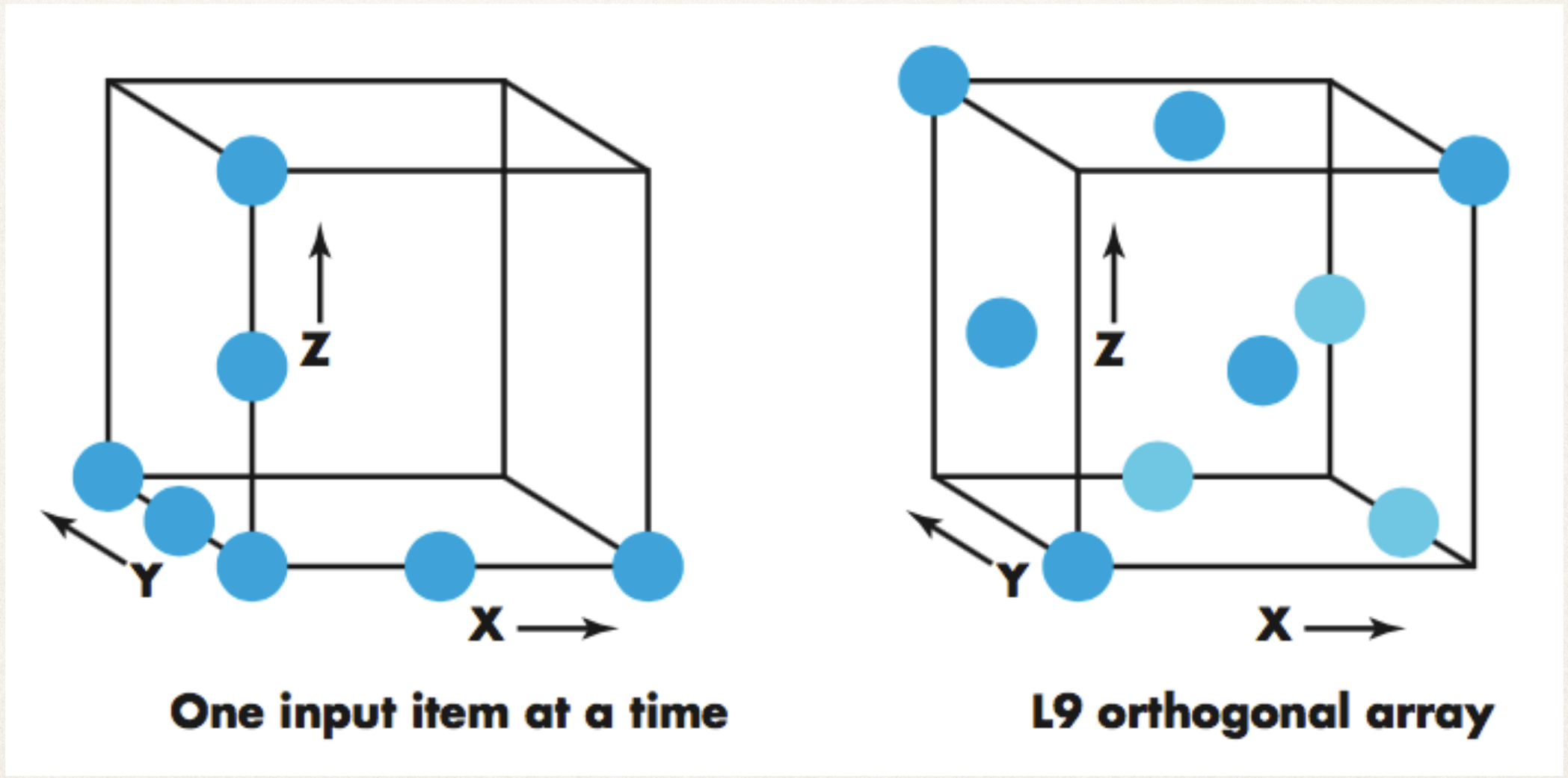
In 1992 R. Brownlie et al. published a paper in the AT&T technical journal on using Orthogonal Arrays to design tests in a real world setting.

1	1	1
2	2	1
1	2	2
2	1	2

History of CIT

UCL

Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases.



Phadke [Pha97], from Software Engineering: A Practitioner's Approach, by Pressman & Maxim, 8th Edition

History of CIT

UCL

Covering Arrays

A t-way covering array is a set of configurations, in which each possible combination of option settings for every combination of t options appears at least once.

In 1994 the first paper on the Automatic Efficient Test Generator (AETG) system was published by D.M. Cohen et al. It showed how earlier work in software testing could be improved by using **Covering Arrays** instead of Orthogonal Arrays.

CAN(2,k,2) for k up to 20000

k	N
3	4
4	5
10	6
15	7
35	8
56	9
126	10
210	11
462	12
792	13
1716	14
3003	15
6435	16
11440	17
20000	18

http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

CIT: Definition

> "Combinatorial Interaction Testing (CIT) is a black-box system testing technique that samples inputs, configurations and parameters, and combines them in a systematic fashion."

Overview of CIT http://cse.unl.edu/~citportal/tutorials.php

Pairwise Combinatorial Testing

UCL

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases
  - Without many constraints, the number of combinations may be unmanageable
- Pairwise combination (instead of exhaustive)
  - Generate combinations that efficiently cover all pairs (triples,…) of classes
  - Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples,…) reduces the number of test cases, but reveal most faults

CIT: Example

UCL

Booking a flight from London:

4 inputs (or parameters)

- Country
- City
- Date
- Return date

CIT: Example

UCL

Booking a flight from London:

Parameters and *Values*

Country	City	Date	Return Date
Spain	Barcelona	20 Aug	27 Aug
	Madrid	21 Aug	26 Aug

CIT: Example

UCL

Country	City	Date	Return Date
Spain	Barcelona	20 Aug	27 Aug
	Madrid	21 Aug	26 Aug

Spain → Barcelona → 20 Aug → 27 Aug
→ 26 Aug
→ 21 Aug → 27 Aug
→ 26 Aug
→ Madrid → 20 Aug → 27 Aug
→ 26 Aug
→ 21 Aug → 27 Aug
→ 26 Aug

1 choice x 2 choices x 2 choices x 2 choices

CIT: Example

UCL

How many possible inputs?

All possible parameter-value combinations:

Country	City	Date	Return Date
Spain	Barcelona	20 Aug	27 Aug
Spain	Madrid	21 Aug	26 Aug
Spain	Barcelona	21 Aug	26 Aug
Spain	Madrid	20 Aug	27 Aug
Spain	Barcelona	20 Aug	26 Aug
Spain	Barcelona	21 Aug	27 Aug
Spain	Madrid	20 Aug	26 Aug
Spain	Madrid	21 Aug	27 Aug

Test suite for the web form

CIT: Example

UCL

Booking a flight from London-Heathrow:

Parameters and *Values*

Country	City	Date	Return
80 countries	184 cities	365 days	365 days

~ $2*10^9$ combinations !

CIT: Example

UCL

Booking a flight from London-Heatrow:

Parameters and *Values*

Country	City	Date	Return
80 countries	184 cities	365 days	365 days

Combinatorial Explosion Problem!

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

Parameters and values controlling Chipmunk Web site display

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

How many combinations are produced by using exhaustive enumeration?

hint: the number of all combinations is the product of the number of classes for each parameter

Observation

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

- Exhaustive enumeration of all n-way combinations of value classes for n parameters and coverage of individual classes, are at the extreme ends of a spectrum of strategies for generating combinations of classes. Between them lie strategies that generate all pairs of classes for different parameters, all triples, and so on.
  - If it is reasonable to expect some potential interaction between parameters (i.e., coverage of individual value classes is deemed insufficient), but covering all combinations is impractical, a good alternative is to generate k-way combinations for $k < n$, typically pairs or triples.
- How much does generating possible pairs of classes save, compared to generating all combinations?

> It turns out that the number of combinations needed to cover all possible pairs of values grows only logarithmically with the number of parameters.

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

Suppose we have just the three parameters: display mode, screen size, and fonts.

Display Mode	Language	Fonts	Color	Screen size
full-graphics	English	Minimal	Monochrome	Hand-held
text-only	French	Standard	Color-map	Laptop
limited-bandwidth	Spanish	Document-loaded	16-bit	Full-size
	Portuguese		True-color	

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

Covering all pairs of value classes for two parameters

Display mode × *Screen size*	
Full-graphics	Hand-held
Full-graphics	Laptop
Full-graphics	Full-size
Text-only	Hand-held
Text-only	Laptop
Text-only	Full-size
Limited-bandwidth	Hand-held
Limited-bandwidth	Laptop
Limited-bandwidth	Full-size

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

Covering all pairs of value classes for three parameters by extending the cross-product of two parameters

Display mode × *Screen size*		*Fonts*
Full-graphics	Hand-held	Minimal
Full-graphics	Laptop	Standard
Full-graphics	Full-size	Document-loaded
Text-only	Hand-held	Standard
Text-only	Laptop	Document-loaded
Text-only	Full-size	Minimal
Limited-bandwidth	Hand-held	Document-loaded
Limited-bandwidth	Laptop	Minimal
Limited-bandwidth	Full-size	Standard

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

UCL

Pairwise Combination

17 test cases

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

Adding constraints

example: color monochrome not compatible with screen laptop and full size

Language	Color	Display Mode	Fonts	Screen Size
English	Monochrome	Full-graphics	Minimal	Hand-held
English	Color-map	Text-only	Standard	Full-size
English	16-bit	Limited-bandwidth	-	Full-size
English	True-color	Text-only	Document-loaded	Laptop
French	Monochrome	Limited-bandwidth	Standard	Laptop
French	Color-map	Full-graphics	Document-loaded	Full-size
French	16-bit	Text-only	Minimal	-
French	True-color	-	-	Hand-held
Spanish	Monochrome	-	Document-loaded	Full-size
Spanish	Color-map	Limited-bandwidth	Minimal	Hand-held
Spanish	16-bit	Full-graphics	Standard	Laptop
Spanish	True-color	Text-only	-	Hand-held
Portuguese	-	-	Monochrome	Text-only
Portuguese	Color-map	-	Minimal	Laptop
Portuguese	16-bit	Limited-bandwidth	Document-loaded	Hand-held
Portuguese	True-color	Full-graphics	Minimal	Full-size
Portuguese	True-color	Limited-bandwidth	Standard	Hand-held

CIT: Example 2

Chapter 11 of Software Testing and Analysis: Process, Principles and Techniques by Pezzè and Young.

Adding constraints

UCL

- We can restrict the set of legal combinations of value classes by adding suitable constraints
- Constraints can be expressed as tuples with wild-cards that match any possible value class

example: color monochrome not compatible with screen laptop and full size

$$OMIT\langle *, Monochrome, *, *, Laptop\rangle$$

$$OMIT\langle *, Monochrome, *, *, Full\text{-}size\rangle$$

Constraints in CIT

UCL

- Hard constraints prohibit certain parameter-value combinations (i.e., interactions)

- Soft constraints are usually imposed by the tester to exclude interactions that do not need to be tested

Pairwise Testing (2-way Testing)

UCL

Problem: Testing all combinations is too expensive

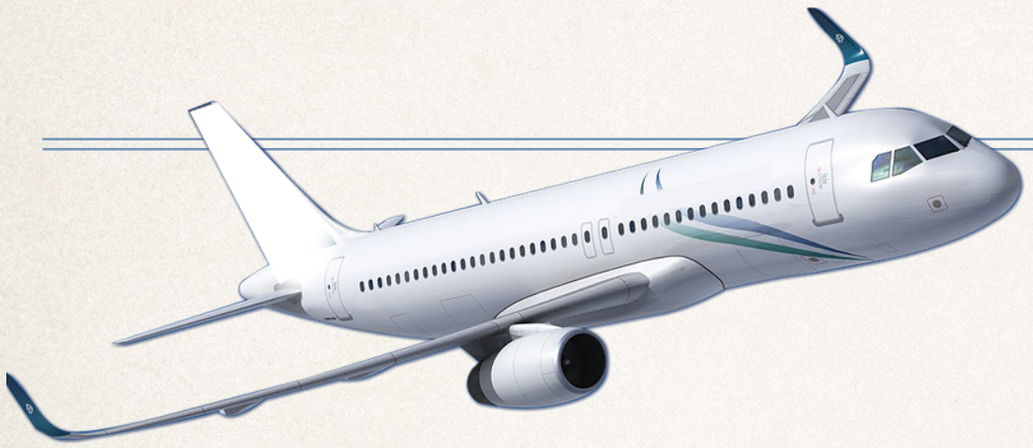
Solution: Test all **interactions** between **any pair** of parameters (**pairwise testing)**

Pairwise Testing (2-way Testing)

UCL

- A pairwise test suite covers all 2-way interactions between any 2 parameters.
- Pairwise (2-way) testing is the most widely studied CIT technique.

Pairwise Testing Example



UCL

All possible parameter-value combinations

Country	City	Date	Return Date
Spain	Barcelona	20 Aug	27 Aug
Spain	Madrid	21 Aug	26 Aug
Spain	Barcelona	21 Aug	26 Aug
Spain	Madrid	20 Aug	27 Aug
Spain	Barcelona	20 Aug	26 Aug
Spain	Barcelona	21 Aug	27 Aug
Spain	Madrid	20 Aug	26 Aug
Spain	Madrid	21 Aug	27 Aug

Pairwise (2-way) interaction test suite

Country	City	Date	Return Date
Spain	Barcelona	20 Aug	27 Aug
Spain	Barcelona	21 Aug	26 Aug
Spain	Barcelona	20 Aug	26 Aug
Spain	Barcelona	21 Aug	27 Aug
Spain	Madrid	21 Aug	26 Aug
Spain	Madrid	20 Aug	27 Aug

T-way Testing

Problem: Testing all combinations is too expensive

Solution: Test all **interactions** between **any set of *t*** of parameters (**t-way testing)**

> A CIT test suite covers all t-way interactions between any t parameters. Such a test suite is also known as a covering array.

T-way Testing

UCL

- Definition: A Covering Array CA(t, k, v) of size N is a table with N rows and k columns
  - Each field of a CA contains a value in the range 0,…, v − 1
  - CA has the following property: every combination of t values between any t parameters occurs in at least one row
  - t is called the strength of a covering array

CIT problem: Find a **minimal** test suite that covers all **t-way interactions**.
CAN(t, k, v) denotes the size of the smallest test suite (i.e., CA).

More details on covering array at http://math.nist.gov/coveringarrays/coveringarray.html

CAN(2,k,2) for k up to 20000

k	N
3	4
4	5
10	6
15	7
35	8
56	9
126	10
210	11
462	12
792	13
1716	14
3003	15
6435	16
11440	17
20000	18

T-way Testing

http://www.nist.gov/

UCL

For n variables with v values each, the number of combinations is

$$\binom{n}{k} v^k$$

Assume 30 parameters with 5 values each. All way combinations are covered by 3,800 tests

Finding a covering array is NP hard

T-way Testing

UCL

CIT problem: Find a **minimal** test suite that covers all **t-way interactions**. **CAN(t, k, v)** denotes the size of the smallest test suite (i.e., CA).

Finding a covering array is NP hard, how one can overcome this?

- Public catalog of covering arrays
- In practise most failures are caused by few parameters interacting
- New algorithms make large-scale combinatorial testing possible

Covering Array Tables

http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

UCL

Covering Array Tables for t=2,3,4,5,6

These tables are maintained by Charlie Colbourn on an irregular basis. Please report updates and corrections. For tables posted in the past, but now updated, see May 2016 March 2015 September 2013 August 2008

For given t and v, the table (t,k,v) gives the current best known upper bound on CAN(t,k,v), the smallest number of rows in a uniform covering array having k factors each with v levels, with coverage at strength t. Covering array numbers are reported for each k up to 20000 for strength two, 10000 for strengths three through six. At present, the authorities are not given with references.

`Best known' means best reported in the literature, to me via email, or implied by a recursive construction. Sizes are reported when an explicit construction is known, not when a probabilistic argument guarantees existence. However, for certain values of v when t is 4, 5, or 6, a constructive conditional expectation algorithm yields better bounds than those implied by the direct and recursive methods -- in these cases, the accompanying graph shows two lines, of which the lower one shows the bounds from the conditional expectation method.

(2,k,2) (2,k,3) (2,k,4) (2,k,5) (2,k,6) (2,k,7) (2,k,8) (2,k,9) (2,k,10) (2,k,11) (2,k,12) (2,k,13) (2,k,14) (2,k,15) (2,k,16) (2,k,17) (2,k,18) (2

(3,k,2) (3,k,3) (3,k,4) (3,k,5) (3,k,6) (3,k,7) (3,k,8) (3,k,9) (3,k,10) (3,k,11) (3,k,12) (3,k,13) (3,k,14) (3,k,15) (3,k,16) (3,k,17) (3,k,18) (3

(4,k,2) (4,k,3) (4,k,4) (4,k,5) (4,k,6) (4,k,7) (4,k,8) (4,k,9) (4,k,10) (4,k,11) (4,k,12) (4,k,13) (4,k,14) (4,k,15) (4,k,16) (4,k,17) (4,k,18) (4

(5,k,2) (5,k,3) (5,k,4) (5,k,5) (5,k,6) (5,k,7) (5,k,8) (5,k,9) (5,k,10) (5,k,11) (5,k,12) (5,k,13) (5,k,14) (5,k,15) (5,k,16) (5,k,17) (5,k,18) (5

(6,k,2) (6,k,3) (6,k,4) (6,k,5) (6,k,6) (6,k,7) (6,k,8) (6,k,9) (6,k,10) (6,k,11) (6,k,12) (6,k,13) (6,k,14) (6,k,15) (6,k,16) (6,k,17) (6,k,18) (6

If you are interested in explicit presentations of covering arrays, which are not necessarily the best known, a good place to start is at the NIST Covering Array Tables. Some explicit solutions are also available from Jose Torres Jimenez here -- click on Covering Arrays.

Covering Array Tables

http://www.public.asu.edu/~ccolbou/src/tabby/catable.html

UCL

Table for CAN(2,k,2) for k up to 20000

Locate the k in the first column that is at least as large as the number of factors in which you are interested. Then let N be the number of rows (tests) given in the second column. A CA(N;2,k,2) exists according to Kleitman and Spencer, and Katona.

Change t: ±

Change v: ±

k	N
3	4
4	5
10	6
15	7
35	8
56	9
126	10
210	11
462	12
792	13
1716	14
3003	15
6435	16
11440	17
20000	18

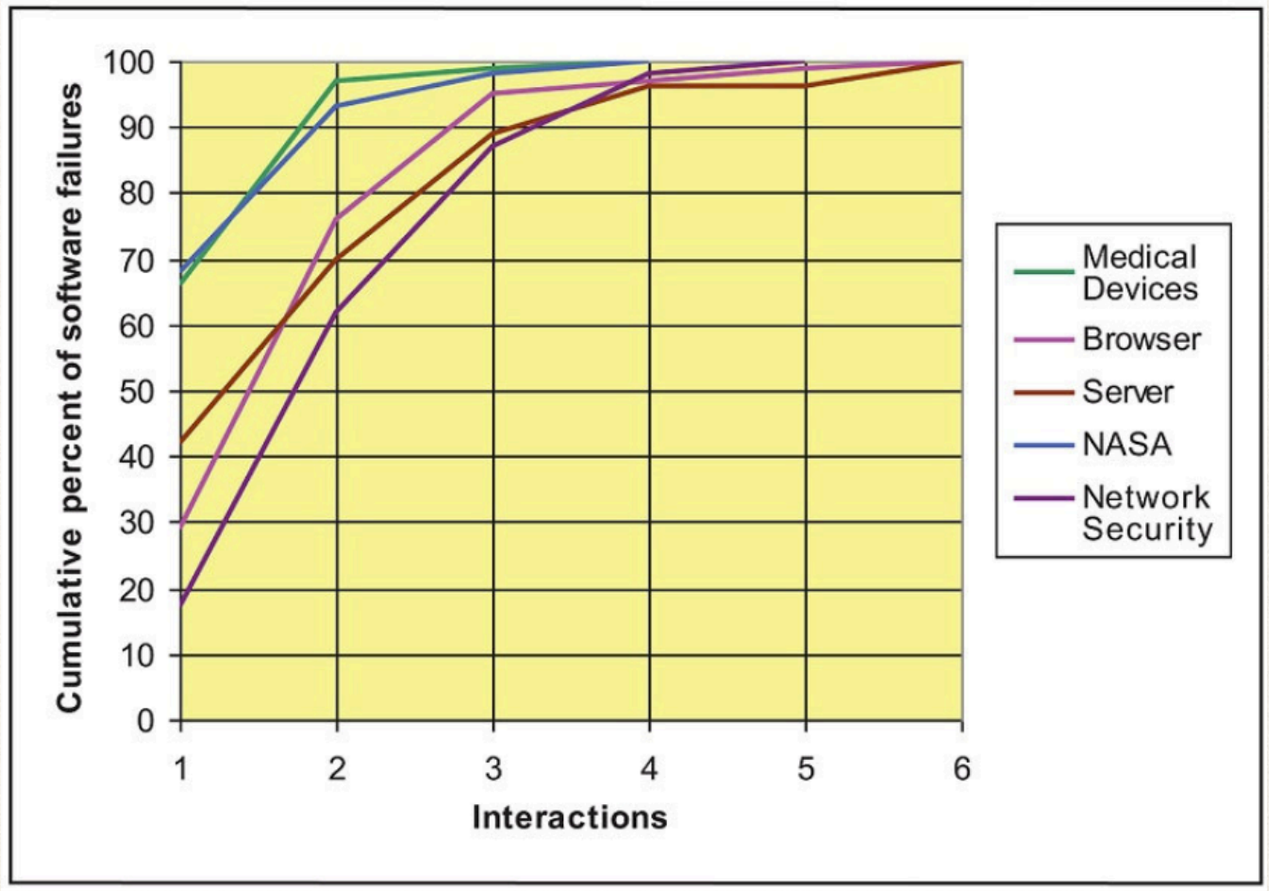
Interactions Involved in Software Failures - empirical data from NIST

http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=910001

UCL

Empirical studies have shown that software interaction faults involve 1 to 6 variables, with no failures involving more than six reported.

Pairwise testing discovers at least 53% of the known faults. 6-way testing discovers 100% of the known faults.



Interaction Rule: most failures are caused by one or two parameters interacting, with progressively fewer by 3, 4, or more parameter interactions.

Approaches for generating t-way interaction test suites

UCL

- Generating combinations that efficiently cover all pairs (or triples, or . . .) of classes is nearly impossible to perform manually for many parameters with many value classes
  - which is, of course, exactly when one really needs to use the approach
- Fortunately, efficient heuristic algorithms exist for this task, and they are simple enough to incorporate in tools

Approaches for generating t-way interaction test suites

UCL

- Greedy
- Meta-heuristics search

Greedy Approach - Example

- Three parameters: P_1, P_2, P_3
  - values for parameter P_1: 0,1
  - values for parameter P_2: 0,1
  - values for parameter P_3: 0,1,2
- Objective: find a pairwise interaction test suite

The In-Parameter-Order (IPO) Strategy

UCL

- First generate a pairwise test set for the first two parameters, then for the first three parameters, and so on

- A pairwise test set for the first n parameters is built by extending the test set for the first n – 1 parameters

  - Horizontal growth: Extend each existing test case by adding one value of the new parameter

  - Vertical growth: Adds new tests, if necessary

IPOG-Test (int *t*, ParameterSet *ps*)

IPOG (In-Parameter-Order-General)

IPOG: A General Strategy for T-Way Software Testing Yu Lei et al., 2007

UCL

Algorithm *IPOG-Test* (int *t*, ParameterSet *ps*)
{
1. initialize test set *ts* to be an empty set
2. denote the parameters in *ps*, in an arbitrary order, as P_1, P_2, ..., and P_n
3. add into test set *ts* a test for each combination of values of the first *t* parameters
4. **for** (int $i = t + 1$; $i \leq n$; i ++){
5. let π be the set of *t*-way combinations of values involving parameter P_i and $t-1$ parameters among the first $i-1$ parameters
6. // horizontal extension for parameter P_i
7. **for** (each test $\tau = (v_1, v_2, ..., v_{i-1})$ in test set *ts*) {
8. choose a value v_i of P_i and replace τ with τ' = $(v_1, v_2, ..., v_{i-1}, v_i)$ so that τ' covers the most number of combinations of values in π
9. remove from π the combinations of values covered by τ'
10. }
11. // vertical extension for parameter P_i
12. **for** (each combination σ in set π){
13. **if** (there exists a test that already covers σ) {
14. remove σ from π
15. } **else** {
16. change an existing test, if possible, or otherwise add a new test to cover σ and remove it from π
17. }
18. }
19.}
20.**return** ts;
}

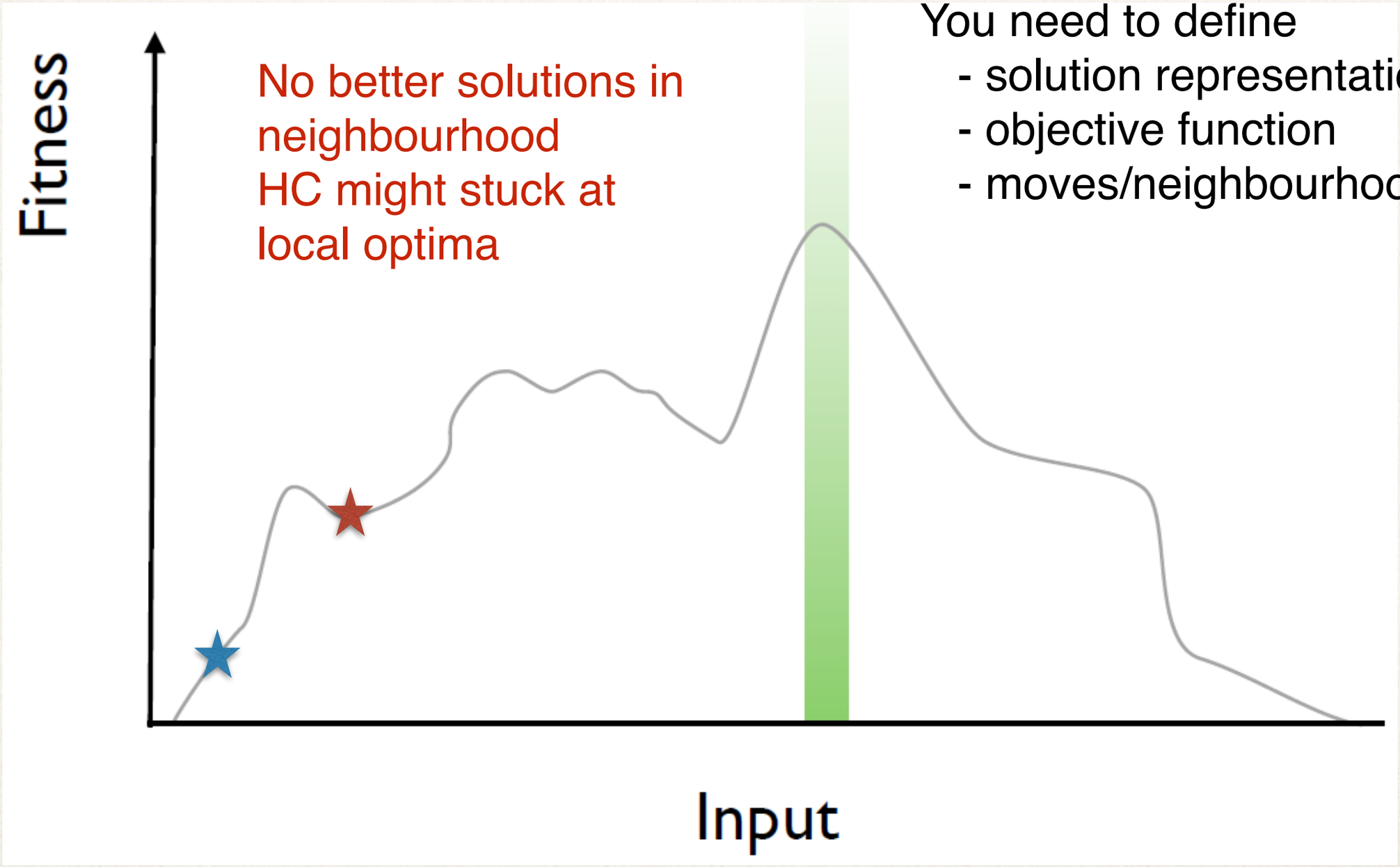
http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=50944

Meta-heuristics

UCL

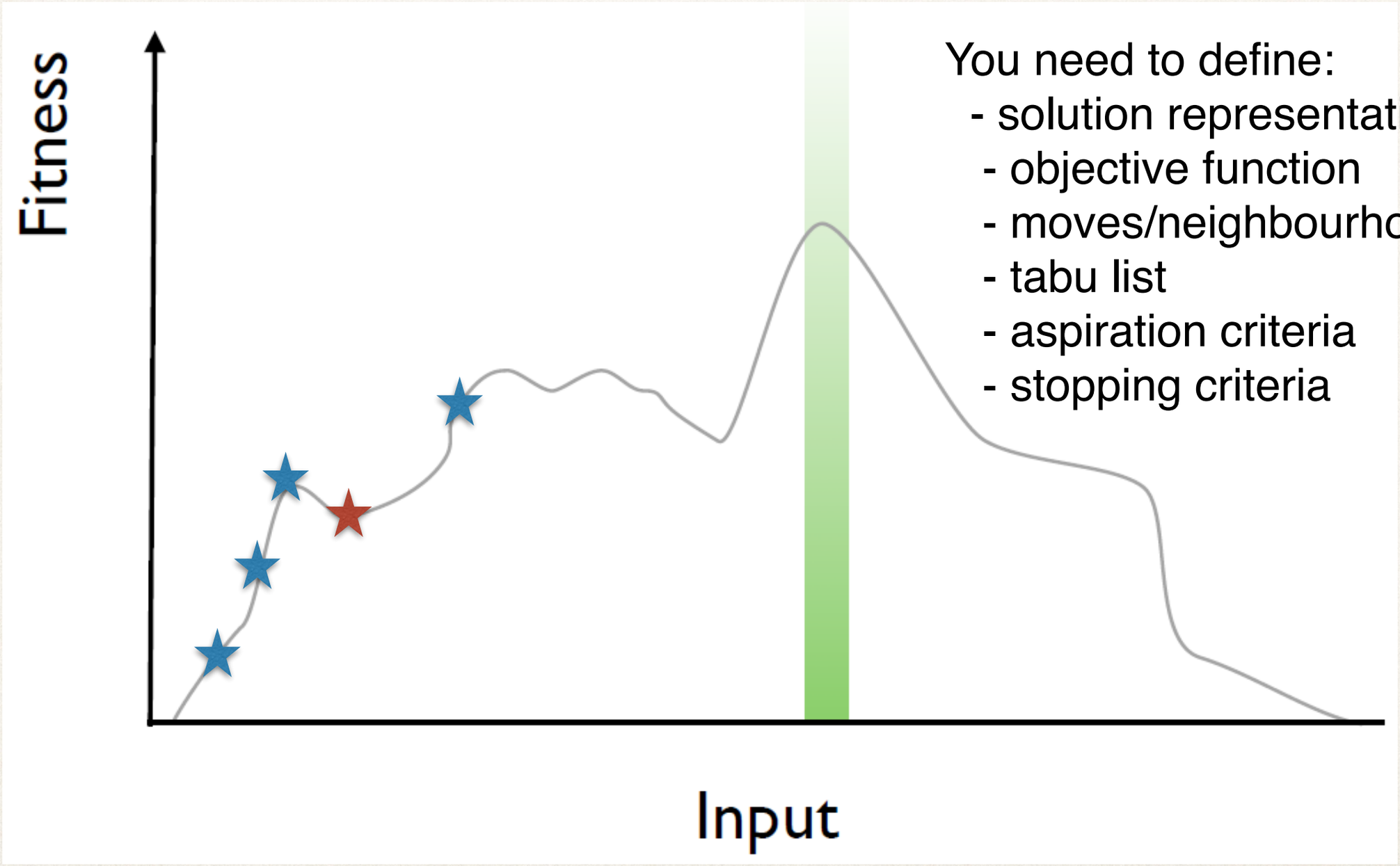
- Meta-heuristics are strategies that guide a search process
- They efficiently explore the search space in order to find near-optimal solutions
- They are not problem-specific

Example: Hill Climbing



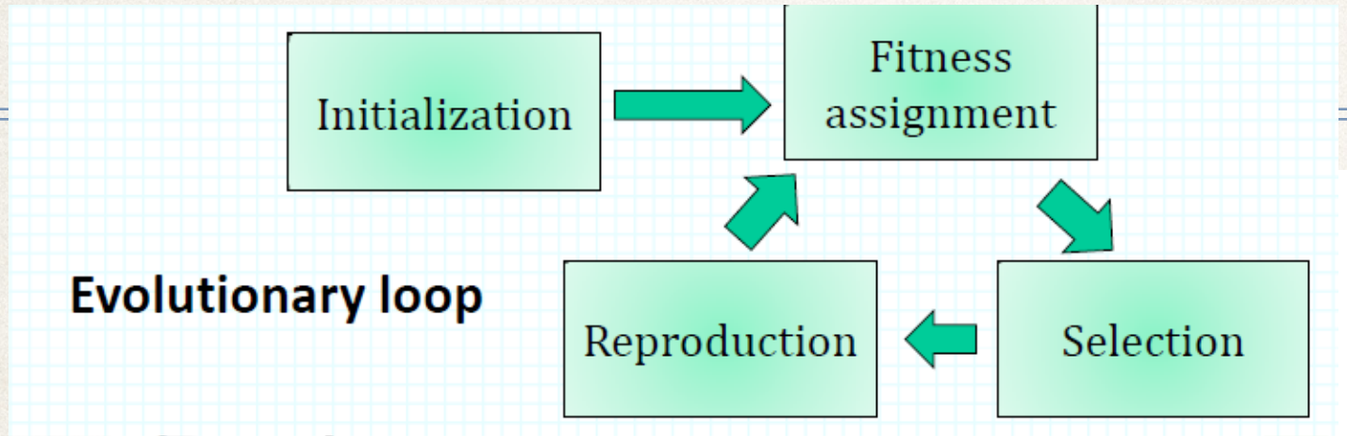
You need to define
- solution representation
- objective function
- moves/neighbourhood

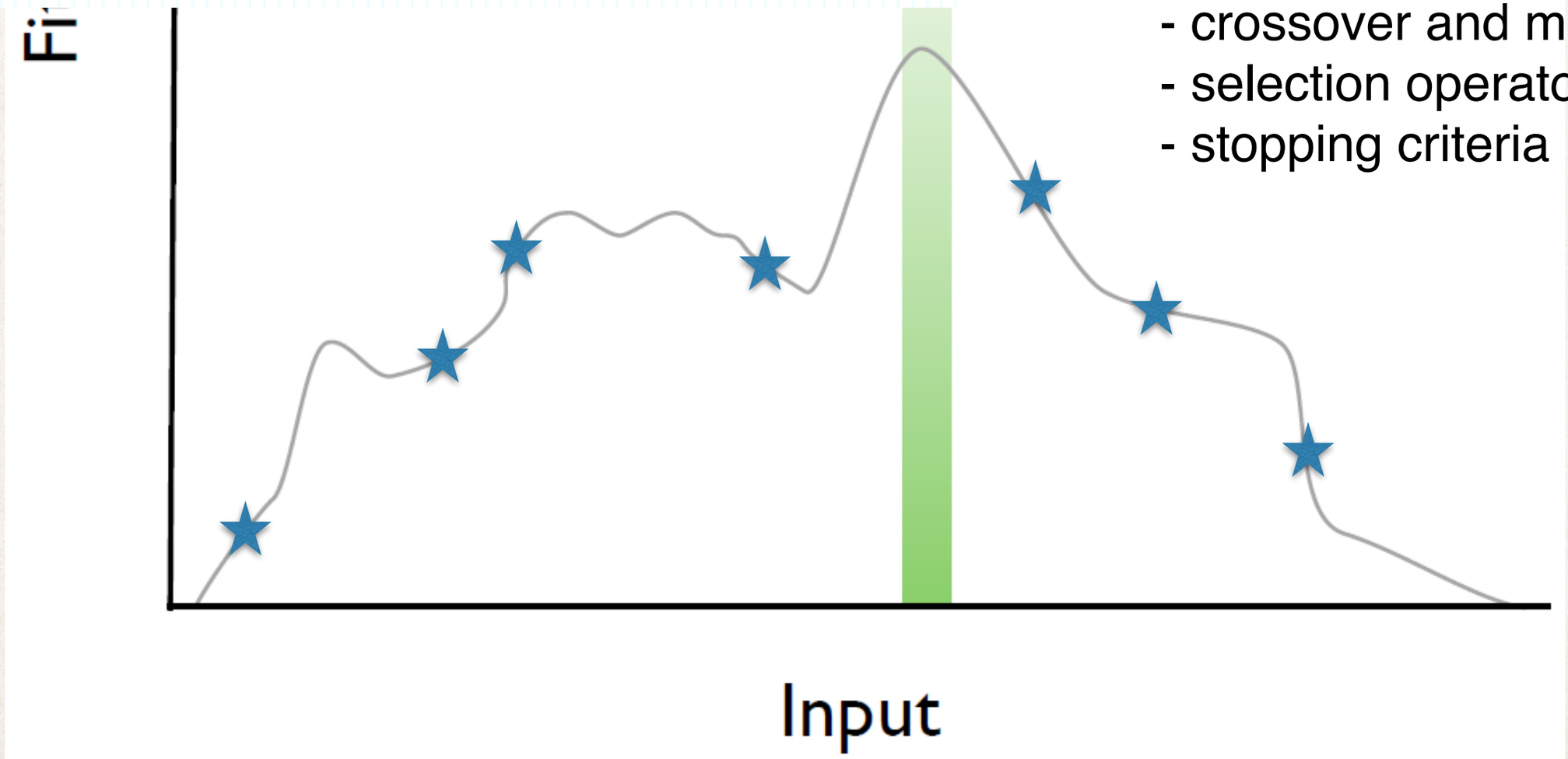
Example: Tabu Search



You need to define:
- solution representation
- objective function
- moves/neighbourhood
- tabu list
- aspiration criteria
- stopping criteria

Example: Genetic Algorithm





You need to define:

- solution representation
- objective function
- crossover and mutation
- selection operator
- stopping criteria

Meta-heuristics for SE

http://cse.unl.edu/~myra/papers/icse_03.pdf

UCL

M. Harman, P. McMinn, J. Teixeira de Souza, S. Yoo. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. Springer, 2012.

google: SBSE tutorial

M. Harman, A. Mansouri and Y. Zhang. Search Based Software Engineering: Trends, Techniques and Applications. ACM Computing Surveys. 45(1): Article 11, 2012.

google: SBSE survey

Meta-heuristics for CIT

http://cse.unl.edu/~myra/papers/icse_03.pdf

UCL

- Outer Search (guess test suite size)
- Inner Search (populates test suite)

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

pick test suite size: 6

Bluetooth	MP3	Camera

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

randomly generate a test suite of size 6

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
None	None	Colour
None	None	Black&White
Included	Included	None
None	None	Colour

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

evaluate fitness: 4 pairs missing

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
None	None	Colour
None	None	Black&White
Included	Included	None
None	None	Colour

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

apply mutation and pick a better solution
(or worse one with certain probability)
fitness: still 4 pairs missing

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
None	None	Colour
None	**Included**	Black&White
Included	Included	None
None	None	Colour

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

repeat until solution found
or run a max number of times
and then increase test suite size

Bluetooth	MP3	Camera
Included	Included	Black&White
None	Included	None
Included	None	Colour
None	None	Black&White
Included	None	None
None	Included	Colour

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

Is this test suite minimal?

Bluetooth	MP3	Camera
Included	Included	Black&White
None	Included	None
Included	None	Colour
None	None	Black&White
Included	None	None
None	Included	Colour

Meta-heuristics for CIT

UCL

Bluetooth	MP3	Camera
Included	Included	Black&White
None	None	None
		Colour

Is this test suite minimal?

Yes! We need at least 6 tests.

Bluetooth	MP3	Camera
Included	Included	Black&White
None	Included	None
Included	None	Colour
None	None	Black&White
Included	None	None
None	Included	Colour

Greedy vs. Meta-heuristics

Evaluating improvements to a meta-heuristic search for constrained interaction testing. Brady J. Garvin et. al, 2011

UCL

Size comparison

(average over 50 runs)

Subject	Greedy	Meta-heuristics
SPIN-S	27	19
SPIN-V	42	36
GCC	24	21
Apache	42	32
Bugzilla	21	16

Time (sec.) comparison

(average over 50 runs)

Subject	Greedy	Meta-heuristics
SPIN-S	0.2	8.6
SPIN-V	11.3	102.1
GCC	204	1902.0
Apache	76.4	109.1
Bugzilla	1.9	9.1

Meta-heuristic search usually generates smaller test suites, but takes more time than a greedy approach for CIT test suite generation

How to Automate Checking Correctness of Output

http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

UCL

Creating test data is the easy part!

- How do we check that the code works correctly on the test input?
  - **Crash testing** server or other code to ensure it does not crash for any test input (like 'fuzz testing') — easy but has limited value
  - **Built-in self test with embedded assertions** incorporate assertions in code to check critical states at different points in the code, or print out important values during execution
  - **Full scale model-checking** using mathematical model of system and model checker to generate expected results for each input - expensive but often intractable

Crash Testing

http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

UCL

- Like “fuzz testing” - send packets or other input to application, watch for crashes
- Unlike fuzz testing, input is non-random; cover all t-way combinations
- May be more efficient - random input generation requires several times as many tests to cover the t-way combinations in a covering array
- Limited utility, but can detect high-risk problems such as: buffer overflows - server crashes

Embedded Assertions

http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

UCL

Simple example:

```
assert( x != 0); // ensure divisor is not zero
```

Or pre and post-conditions:

```
/requires amount >= 0;

/ensures balance == \old(balance) - amount && \result == balance;
```

Embedded Assertions

http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-142.pdf

UCL

- Assertions check properties of expected result:

```
ensures balance == \old(balance) - amount && \result == balance;
```

- Reasonable assurance that code works correctly across the range of expected inputs

- May identify problems with handling unanticipated inputs

Summary (1/2)

- CIT is a black-box testing technique
  - A CIT test suite covers all t-way interactions between any t parameters.
    - Pairwise (2-way) testing is the most widely studied CIT technique.
- CIT is now a practical approach that produces high quality testing at lower cost
  - Good algorithms and user-friendly tools are available – free tools from NIST, Microsoft, others

Summary (2/2)

- Basic combinatorial testing can be used in two ways:
  - combinations of configuration values
  - combinations of input values
  - these can be used separately or at the same time
- Case studies are beginning to appear, many tools and materials available on-line
  - NIST web site csrc.nist.gov/acts
  - pairwise.org