Lecture 1 – Combinatorial Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2024 Spring

Black-box Testing

- **Black-box testing** views the software as a **black-box** without knowing its internal structure.

Black-box Testing

- **Black-box testing** views the software as a **black-box** without knowing its internal structure.
- It is also known as **functional testing** or **behavioral testing**.

Black-box Testing

- **Black-box testing** views the software as a **black-box** without knowing its internal structure.
- It is also known as **functional testing** or **behavioral testing**.
- Test data are derived from the **specification** of the software.

Black-box Testing

- **Black-box testing** views the software as a **black-box** without knowing its internal structure.
- It is also known as **functional testing** or **behavioral testing**.
- Test data are derived from the **specification** of the software.
- In general, **exhaustive testing** is not feasible. It means that we **cannot guarantee** that the software is free of defects.

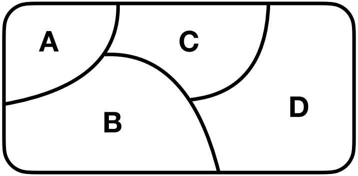
Black-box Testing

- **Black-box testing** views the software as a **black-box** without knowing its internal structure.
- It is also known as **functional testing** or **behavioral testing**.
- Test data are derived from the **specification** of the software.
- In general, **exhaustive testing** is not feasible. It means that we **cannot guarantee** that the software is free of defects.
- We need to pick a good set of test cases to **maximize** the chance of **finding software errors**.

Contents

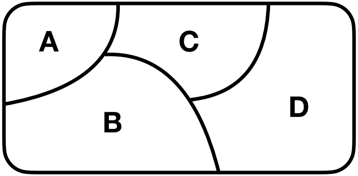
Contents

Equivalence Partitioning (EP)



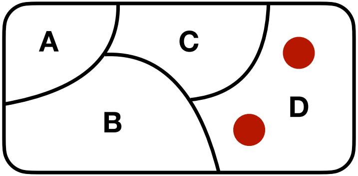
- **Equivalence partitioning** is a black-box testing technique that divides the input domain of a program into **equivalence classes**.

Equivalence Partitioning (EP)



- **Equivalence partitioning** is a black-box testing technique that divides the input domain of a program into **equivalence classes**.
- The technique is based on the observation that the program should behave the same way for all members of an equivalence class.

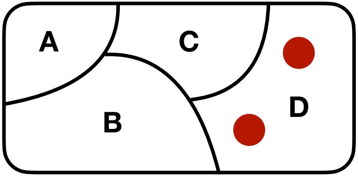
Equivalence Partitioning (EP)



- If one test case in an equivalence class reveals an error, it is likely that other test cases in the **same equivalence class** will also reveal the **same error**.

Equivalence Partitioning (EP)

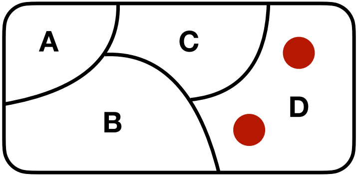
PLRG



- If one test case in an equivalence class reveals an error, it is likely that other test cases in the **same equivalence class** will also reveal the **same error**.

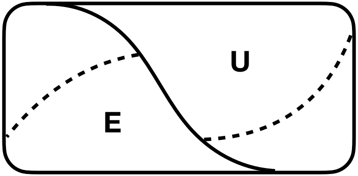
- The idea is to **reduce** the number of test cases by selecting **one test case** from each equivalence class.

Equivalence Partitioning (EP)



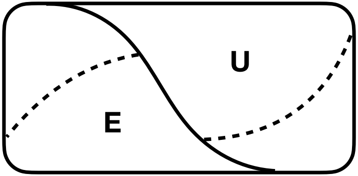
- If one test case in an equivalence class reveals an error, it is likely that other test cases in the **same equivalence class** will also reveal the **same error**.
- The idea is to **reduce** the number of test cases by selecting **one test case** from each equivalence class.
- Then, how to define the equivalence classes?

Equivalence Partitioning (EP)



- One possible way to define the equivalence classes is to divide the input domain into **expected** and **unexpected** inputs.
  - **Expected (E)** or legal inputs
  - **Unexpected (U)** or illegal inputs

Equivalence Partitioning (EP)



- One possible way to define the equivalence classes is to divide the input domain into **expected** and **unexpected** inputs.
  - **Expected (E)** or legal inputs
  - **Unexpected (U)** or illegal inputs

- We can further divide the expected inputs into smaller equivalence classes.

Equivalence Partitioning (EP)

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
  - $E = \{ \text{ a password } p \mid 6 \leq |p| \leq 20\}$
  - $U = \{ \text{ a password } p \mid |p| < 6 \vee |p| > 20\}$

Equivalence Partitioning (EP)

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
  - $E = \{ \text{ a password } p \mid 6 \leq |p| \leq 20\}$
  - $U = \{ \text{ a password } p \mid |p| < 6 \vee |p| > 20\}$
- We can divide it more finely:
  - $E_1 = \{ \text{ a password } p \mid 6 \leq |p| \leq 10\}$ for *weak passwords*
  - $E_2 = \{ \text{ a password } p \mid 11 \leq |p| \leq 15\}$ for *medium-strength passwords*
  - $E_3 = \{ \text{ a password } p \mid 16 \leq |p| \leq 20\}$ for *strong passwords*
  - $U_1 = \{ \text{ a password } p \mid |p| < 6\}$ for *too short passwords*
  - $U_2 = \{ \text{ a password } p \mid |p| > 20\}$ for *too long passwords*

Equivalence Partitioning (EP)

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- We can divide the input domain into two equivalence classes:
  - $E = \{ \text{a password } p \mid 6 \leq |p| \leq 20\}$
  - $U = \{ \text{a password } p \mid |p| < 6 \vee |p| > 20\}$
- We can divide it more finely:
  - $E_1 = \{ \text{a password } p \mid 6 \leq |p| \leq 10\}$ for *weak passwords*
  - $E_2 = \{ \text{a password } p \mid 11 \leq |p| \leq 15\}$ for *medium-strength passwords*
  - $E_3 = \{ \text{a password } p \mid 16 \leq |p| \leq 20\}$ for *strong passwords*
  - $U_1 = \{ \text{a password } p \mid |p| < 6\}$ for *too short passwords*
  - $U_2 = \{ \text{a password } p \mid |p| > 20\}$ for *too long passwords*
- We can select one test case from each equivalence class.

$$I = \{p_1, p_2, p_3, p_4, p_5\}$$

such that $|p_1| = 7$, $|p_2| = 13$, $|p_3| = 18$, $|p_4| = 3$, $|p_5| = 40$

Equivalence Partitioning (EP)

PLRG

- There are **many ways** to partition the input domain.

[1] [TSE'91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies"

Equivalence Partitioning (EP)

- There are **many ways** to partition the input domain.
- Even from the same equivalence classes, we can choose **different test cases**.

[1] [TSE'91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies"

Equivalence Partitioning (EP)

PLRG

- There are **many ways** to partition the input domain.
- Even from the same equivalence classes, we can choose **different test cases**.
- **Effectiveness** may depend on the tester's **experience** and **intuition**.

[1] [TSE'91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies"

Equivalence Partitioning (EP)

- There are **many ways** to partition the input domain.

- Even from the same equivalence classes, we can choose **different test cases**.

- **Effectiveness** may depend on the tester's **experience** and **intuition**.

  Partition testing can be better, worse, or the same as random testing, depending on how the partitioning is done.[1]

[1] [TSE'91] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies"

Contents

Off-by-one Errors

- **Logic errors** often occur at the **boundaries** of the input domain.

Off-by-one Errors

- **Logic errors** often occur at the **boundaries** of the input domain.
- They usually occur due to **off-by-one** errors caused by misunderstanding the **boundary conditions**.

Off-by-one Errors

- **Logic errors** often occur at the **boundaries** of the input domain.
- They usually occur due to **off-by-one** errors caused by misunderstanding the **boundary conditions**.
- It is **simple** but actually **very common**.

Off-by-one Errors – Looping Over Arrays

```
for (let i = 0; i < 10; i++) {
  /* body of the loop */
}
```

CORRECT

```
for (let i = 1; i < 10; i++) {
  /* body of the loop */
}
```

INCORRECT

```
for (let i = 0; i <= 10; i++) {
  /* body of the loop */
}
```

INCORRECT

```
for (let i = 0; i < 11; i++) {
  /* body of the loop */
}
```

INCORRECT

Off-by-one Errors – Fencepost error

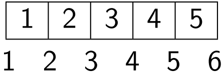
If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

$15\ /\ 3 = 5$ posts?

Off-by-one Errors – Fencepost error

If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

15 / 3 = 5 posts? No, you need 6 posts!



Off-by-one Errors – Fencepost error

If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

15 / 3 = 5 posts? No, you need 6 posts!

1	2	3	4	5

1 2 3 4 5 6

`linspace(a, b, n)` in MATLAB is a linear interpolation function that generates a row vector of n **points** instead of n **intervals** between a and b.

Off-by-one Errors – Fencepost error

If you build a straight fence 15 meters long with posts spaced 3 meters apart, how many posts do you need?

15 / 3 = 5 posts? No, you need 6 posts!

1	2	3	4	5

1 2 3 4 5 6

`linspace(a, b, n)` in MATLAB is a linear interpolation function that generates a row vector of n **points** instead of n **intervals** between a and b.

```
linspace(0, 10, 5) == [0, 2.5, 5, 7.5, 10]
                   != [0, 2, 4, 6, 8, 10]
```

Off-by-one Errors – strncat in C

```
void foo (char *s)
{
  char buf[15];
  memset(buf, 0, sizeof(buf));
  // Final parameter should be: sizeof(buf)-1
  strncat(buf, s, sizeof(buf));
}
```

- Off-by-one errors are common in using the C library because it is **not consistent** with respect to whether one needs to subtract 1 byte.

Off-by-one Errors – strncat in C

```c
void foo (char *s)
{
  char buf[15];
  memset(buf, 0, sizeof(buf));
  // Final parameter should be: sizeof(buf)-1
  strncat(buf, s, sizeof(buf));
}
```

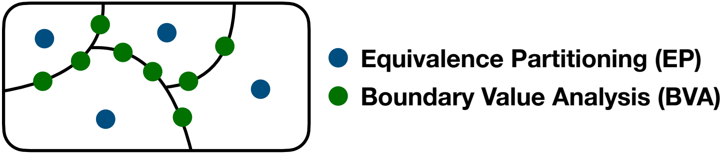
- Off-by-one errors are common in using the C library because it is **not consistent** with respect to whether one needs to subtract 1 byte.
- For example, we need to subtract 1 byte from the length of the buffer in `strncat` but not in `fgets` or `strncpy`.

Off-by-one Errors – `strncat` in C

```
void foo (char *s)
{
  char buf[15];
  memset(buf, 0, sizeof(buf));
  // Final parameter should be: sizeof(buf)-1
  strncat(buf, s, sizeof(buf));
}
```

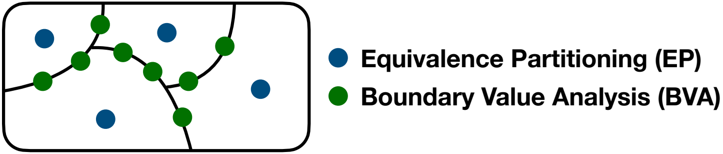
- Off-by-one errors are common in using the C library because it is **not consistent** with respect to whether one needs to subtract 1 byte.
- For example, we need to subtract 1 byte from the length of the buffer in `strncat` but not in `fgets` or `strncpy`.
- So, the programmer has to remember for which functions they need to subtract 1.

Boundary Value Analysis (BVA)



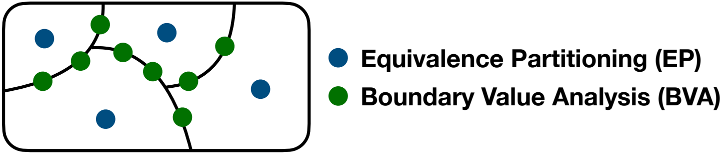
- **Boundary value analysis** is a black-box testing technique that focuses on the **boundaries** of the input domain.

Boundary Value Analysis (BVA)



- **Boundary value analysis** is a black-box testing technique that focuses on the **boundaries** of the input domain.
- The idea is to select test cases at the **boundaries** of the equivalence classes.

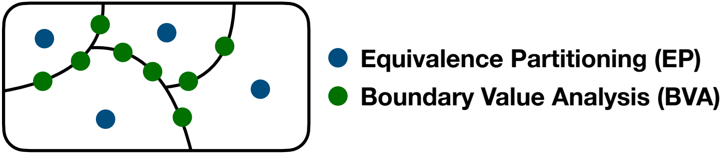
Boundary Value Analysis (BVA)



- **Boundary value analysis** is a black-box testing technique that focuses on the **boundaries** of the input domain.
- The idea is to select test cases at the **boundaries** of the equivalence classes.
- The technique is based on the observation that the program is more likely to fail at the **boundaries** of the input domain.

Boundary Value Analysis (BVA)

PLRG



- **Boundary value analysis** is a black-box testing technique that focuses on the **boundaries** of the input domain.
- The idea is to select test cases at the **boundaries** of the equivalence classes.
- The technique is based on the observation that the program is more likely to fail at the **boundaries** of the input domain.
- It is usually used in combination with **equivalence partitioning**.

Boundary Value Analysis (BVA)

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- Consider the equivalence classes:
  - $E_1 = \{$ a password $p \mid 6 \leq |p| \leq 10\}$ for *weak passwords*
  - $E_2 = \{$ a password $p \mid 11 \leq |p| \leq 15\}$ for *medium-strength passwords*
  - $E_3 = \{$ a password $p \mid 16 \leq |p| \leq 20\}$ for *strong passwords*
  - $U_1 = \{$ a password $p \mid |p| < 6\}$ for *too short passwords*
  - $U_2 = \{$ a password $p \mid |p| > 20\}$ for *too long passwords*

Boundary Value Analysis (BVA)

Example

Consider a program that takes a **password** as input. The length of the password must be between 6 and 20 characters.

- Consider the equivalence classes:
  - $E_1 = \{$ a password $p \mid 6 \leq |p| \leq 10\}$ for *weak passwords*
  - $E_2 = \{$ a password $p \mid 11 \leq |p| \leq 15\}$ for *medium-strength passwords*
  - $E_3 = \{$ a password $p \mid 16 \leq |p| \leq 20\}$ for *strong passwords*
  - $U_1 = \{$ a password $p \mid |p| < 6\}$ for *too short passwords*
  - $U_2 = \{$ a password $p \mid |p| > 20\}$ for *too long passwords*
- We can select test cases at the boundaries of the equivalence classes.

$$I = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$$

such that

$$\begin{array}{llll} |p_1| = 5 & |p_2| = 6 & |p_3| = 10 & |p_4| = 11 \\ |p_5| = 15 & |p_6| = 16 & |p_7| = 20 & |p_8| = 21 \end{array}$$

Contents

Category Partition Method (CPM)

- Most programs behave differently when they receive different **parameters** or are executed under different **environments**.

Category Partition Method (CPM)

- Most programs behave differently when they receive different **parameters** or are executed under different **environments**.
- **Category partition method (CPM)** is a black-box testing technique that systematically generates test cases by considering the combinations of the categories of the input domain.
  1. Analyze **specification**
  2. Identify **parameters** and **environments**
  3. Identify **categories** for each parameter and environment
  4. **Partition** categories into equivalence classes
  5. Identify **constraints**
  6. **Generate** test cases

Category Partition Method (CPM)

Example

Unix command `grep` searches for files in a directory hierarchy with the following syntax:

```
grep <pattern> <filename>
```

For example,

- `grep park myfile` displays all lines in `myfile` that contain the word "`park`".
- `grep "hello world" myfile` displays all lines in `myfile` that contain the phrase "`hello world`".
- `grep " said \"hello " myfile` displays all lines in `myfile` that contain the phrase "` said "hello `".

Perform category partition method for the `grep` command.

Category Partition Method (CPM)

① Analyze **specification**

② Identify **parameters** and **environments**

- **Parameters** – (1) `<pattern>` and (2) `<filename>`
  - The `<pattern>` is a pattern to search for.
  - To include spaces in the pattern, it must be enclosed in quotes (`"`).
  - To include a quotation mark in the pattern, it must be escaped with a backslash (`\"`).
  - ...
- **Environments** – (3) file contents
  - ...

Category Partition Method (CPM)

③ Identify **categories** for each parameter and environment

1. **Parameter** – `<pattern>`
   - **Size**
   - **Quotation marks**
   - **Embedded spaces**
   - **Embedded quotation marks**
2. **Parameter** – `<filename>`
   - **Validity**
3. **Environment** – file contents
   - **Number of occurrences of the pattern**
   - **Number of occurrences of the pattern in a line**

Category Partition Method (CPM)

④ **Partition** categories into equivalence classes

1. **Parameter** – `<pattern>`
   - **Size** – 0 / 1 / ≥ 2
   - **Quotation marks** – quoted (Q) / unquoted (U) / improper (I)
   - **Embedded spaces** – none (N) / single (S) / multiple (M)
   - **Embedded quotation marks** – none (N) / single (S) / multiple (M)
2. **Parameter** – `<filename>`
   - **Validity** – exists (E) / not exists (N) / omitted (O)
3. **Environment** – file contents
   - **Number of occurrences of the pattern** – 0 / 1 / ≥ 2
   - **Number of occurrences of the pattern in a line** – 0 / 1 / ≥ 2

Category Partition Method (CPM)

How many combinations of the partitioned categories?

$$3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 = 2,187$$

Category Partition Method (CPM)

How many combinations of the partitioned categories?

$$3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 = 2,187$$

⑤ Identify **constraints**

- For example, no embedded space for unquoted pattern.

$$\text{unquoted (U)} \quad \not\Rightarrow\!\Leftarrow \quad \text{single (S) for } \textbf{embedded spaces}$$
$$\text{unquoted (U)} \quad \not\Rightarrow\!\Leftarrow \quad \text{multiple (M) for } \textbf{embedded spaces}$$

Category Partition Method (CPM)

How many combinations of the partitioned categories?

$$3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 = 2,187$$

⑤ Identify **constraints**

- For example, no embedded space for unquoted pattern.

$$\text{unquoted (U)} \quad \Rightarrow\!\Leftarrow \quad \text{single (S) for } \textbf{embedded spaces}$$
$$\text{unquoted (U)} \quad \Rightarrow\!\Leftarrow \quad \text{multiple (M) for } \textbf{embedded spaces}$$

⑥ **Generate** test cases

- Pick one **test case** from each combination **satisfying** the constraints.

Size	Q	Space	Emb. Q	Valid	Occur	Occur
0	Q	N	N	E	0	0
1	U	S	S	N	1	1
≥ 2	I	M	M	O	≥ 2	≥ 2

```
grep "hello world" myfile        # `myfile` is an empty file
```

Contents

Combinatorial Testing (CT)

- Testing all combinations is still **too expensive** because of the **combinatorial explosion**!

Combinatorial Testing (CT)

- Testing all combinations is still **too expensive** because of the **combinatorial explosion**!
  - For example, we need 1,799,736,525 test cases required for the following airport system:

Airline	Destination	Departure Date	Return Date
79	171	365	365

Combinatorial Testing (CT)

- Testing all combinations is still **too expensive** because of the **combinatorial explosion**!
  - For example, we need 1,799,736,525 test cases required for the following airport system:

Airline	Destination	Departure Date	Return Date
79	171	365	365

- **Combinatorial testing (CT)** or **combinatorial interaction testing (CIT)** constructs test cases by considering the **interactions** between the parameters.

Covering Array (CA)

Definition (Interaction)

For k parameters with v values each, a t-**way interaction** is a combination of values for t parameters.

Covering Array (CA)

PLRG

Definition (Interaction)

For k parameters with v values each, a t-**way interaction** is a combination of values for t parameters.

Definition (Covering Array (CA))

A **covering array** $A = CA(N; t, k, v)$ is a $N \times k$ matrix such that every field is an element from the set $[0, v-1]$, and every t-way interaction is covered at least once by a row of A.

Covering Array (CA)

PLRG

Definition (Interaction)

For k parameters with v values each, a t-**way interaction** is a combination of values for t parameters.

Definition (Covering Array (CA))

A **covering array** $A = CA(N; t, k, v)$ is a $N \times k$ matrix such that every field is an element from the set $[0, v-1]$, and every t-way interaction is covered at least once by a row of A.

A	B	C
0	0	0
1	1	1

$\Rightarrow$

$CA(N = 4; t = 2, k = 3, v = 2)$

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

Mixed Covering Array (MCA)

PLRG

Definition (Mixed Covering Array (MCA))

A **mixed covering array** $A = CA(N; t, k, v = (v_1, v_2, \ldots, v_k))$ is a $N \times k$ matrix such that every field in the i-th column is an element from the set $[0, v_i - 1]$, and every t-way interaction is covered at least once by a row of A.

Mixed Covering Array (MCA)

Definition (Mixed Covering Array (MCA))

A **mixed covering array** $A = CA(N; t, k, v = (v_1, v_2, \dots, v_k))$ is a $N \times k$ matrix such that every field in the i-th column is an element from the set $[0, v_i - 1]$, and every t-way interaction is covered at least once by a row of A.

A	B	C
0	0	0
1	1	1
		2

$\Rightarrow$

$CA(N = 6; t = 2, k = 3, v = (2, 2, 3))$

A	B	C
0	0	0
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

Constraint Mixed Covering Array (CMCA)

Definition (Constrinat Mixed Covering Array (CMCA))

A **mixed covering array** $A = CA(N; t, k, v = (v_1, v_2, \ldots, v_k), P)$ is a $N \times k$ matrix such that every field in the i-th column is an element from the set $[0, v_i - 1]$, and every **valid** t-way interaction is covered at least once by a row of A. We say that a t-way interaction is **valid** if it satisfies the predicate P.

Constraint Mixed Covering Array (CMCA)

Definition (Constrinat Mixed Covering Array (CMCA))

A **mixed covering array** $A = CA(N; t, k, v = (v_1, v_2, \ldots, v_k), P)$ is a $N \times k$ matrix such that every field in the i-th column is an element from the set $[0, v_i - 1]$, and every **valid** t-way interaction is covered at least once by a row of A. We say that a t-way interaction is **valid** if it satisfies the predicate P.

A	B	C
0	0	0
1	1	1
		2

$\Rightarrow$

$CA(N = 5; t = 2, k = 3, v = (2, 2, 3), P)$

A	B	C
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

$$P(x, y) = x + y > 0$$

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Definition (Pairwise Testing)

Pairwise testing is a special case of combinatorial testing with $t = 2$.

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Definition (Pairwise Testing)

Pairwise testing is a special case of combinatorial testing with $t = 2$.

Pairwise testing produces 5 test cases for the following system:

A	B	C
0	0	0
1	1	1
		2

$$P(x, y) = x + y > 0$$

$\Rightarrow$

$$CA(N = 5; t = 2, k = 3, v = (2, 2, 3), P)$$

A	B	C
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Definition (Pairwise Testing)

Pairwise testing is a special case of combinatorial testing with $t = 2$.

Combinatorial Testing (CT)

Definition (Combinatorial Testing (CT))

Combinatorial testing with a strength t produces a test suite from a covering array $CA(N; t, k, v)$ for a system with k parameters, each with

Definition (Pairwise Testing)

Pairwise testing is a special case of combinatorial testing with $t = 2$.

Pairwise testing produces 5 test cases for the following system:

A	B	C
0	0	0
1	1	1
		2

$$P(x, y) = x + y > 0$$

$\Rightarrow$

$$CA(N = 5; t = 2, k = 3, v = (2, 2, 3), P)$$

A	B	C
0	1	1
1	0	2
1	1	0
1	0	1
0	1	2

Fault Detection Effectiveness

- Pairwise testing is a **good trade-off** between test effort and test effectiveness.
  - For a system with 20 parameters each with 15 values, **pairwise testing** only requires 412 tests, whereas exhaustive testing requires $15^{20} = 3.5 \times 10^{25}$ tests.

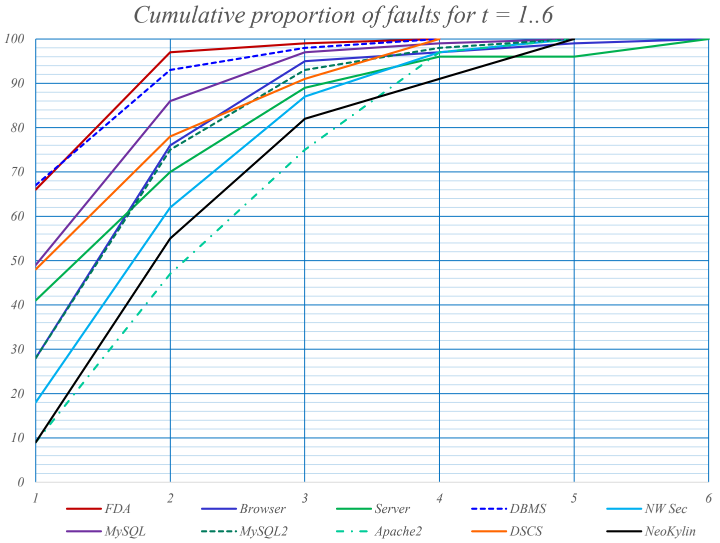
Fault Detection Effectiveness

- Pairwise testing is a **good trade-off** between test effort and test effectiveness.
  - For a system with 20 parameters each with 15 values, **pairwise testing** only requires 412 tests, whereas exhaustive testing requires $15^{20} = 3.5 \times 10^{25}$ tests.
- Is higher strength always better for fault detection?

Fault Detection Effectiveness

- Pairwise testing is a **good trade-off** between test effort and test effectiveness.
  - For a system with 20 parameters each with 15 values, **pairwise testing** only requires 412 tests, whereas exhaustive testing requires $15^{20} = 3.5 \times 10^{25}$ tests.
- Is higher strength always better for fault detection?
- It depends on the target program, but we can analyze the **general trend** against **a set of known faults**.
  - **Pairwise testing** discovers at least **53%** of the known faults.
  - **6-way testing** discovers **100%** of the known faults.

Fault Detection Effectiveness



"Combinatorial Methods in Software Testing" by Rick Kuhn, NIST,

Greedy Algorithm – IPOG Strategy

- The problem of generating a minimum covering array is **NP-complete**.
  - It can be reduced to the **vertex cover problem**.

[2] **[ECBS'07]** LEI, Yu, et al. "*IPOG: A general strategy for t-way software testing.*

Greedy Algorithm – IPOG Strategy

- The problem of generating a minimum covering array is **NP-complete**.
  - It can be reduced to the **vertex cover problem**.
- Let's learn a **polynomial time** greedy algorithm called **IPOG (In-Parameter-Order-General)**[2] that generates a covering array with a strength t. It is not optimal but practical.

[2] **[ECBS'07]** LEI, Yu, et al. "*IPOG: A general strategy for t-way software testing.*

Greedy Algorithm – IPOG Strategy

1. Initialize **test set** *ts* to be an empty set.
2. **Parameters** are P_1, P_2, ..., P_k.
3. Add a test into *ts* for **all interactions of the first t parameters**.
4. `for` ($i = t + 1$; $i \leq n$; i++) (**Horizontal Growth**)
   1. Let π be the **set of t-way interactions** involving **parameter** P_i and $t - 1$ parameters among the first $i - 1$ parameters.
   2. `for` (test $\gamma = (v_1, v_2, \cdots, v_{i-1}) \in ts$)
      1. **Choose** a value v_i of P_i
      2. **Replace** γ with $\gamma' = (v_1, v_2, \cdots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π.
      3. **Remove** from π the interaction covered by γ'.
5. `for` (interaction $\alpha \in \pi$) (**Vertical Growth**)
   1. `if` ($\exists$ a test covers α) **Remove** α from π.
   2. `else if` (possible) **Change** an existing test
   3. `else` **Add** a new test to cover α and **Remove** it from π.

Greedy Algorithm – IPOG Strategy

1. Initialize **test set** *ts* to be an empty set.
2. **Parameters** are $P_1, P_2, \ldots, P_k$.
3. Add a test into *ts* for **all interactions of the first** t **parameters**.

Greedy Algorithm – IPOG Strategy

PLRG

Example

We want to **pairwise testing** for the following system:

P_1	P_2	P_3
0	0	0
1	1	1
		2

Adding all combinations of values between the first 2 parameters:

$ts =$

P_1	P_2
0	0
0	1
1	0
1	1

Greedy Algorithm – IPOG Strategy

1. Initialize **test set** *ts* to be an empty set.
2. **Parameters** are $P_1, P_2, \ldots, P_k$.
3. Add a test into *ts* for **all interactions of the first t parameters**.
4. `for` $(i = t + 1;\ i \leq n;\ i{+}{+})$ (**Horizontal Growth**)
    1. Let π be the **set of t-way interactions** involving **parameter P_i** and $t - 1$ parameters among the first $i - 1$ parameters.

Greedy Algorithm – IPOG Strategy

Set π as pairs to cover involving P_3:

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

1. Initialize **test set** *ts* to be an empty set.
2. **Parameters** are P_1, P_2, ..., P_k.
3. Add a test into *ts* for **all interactions of the first** t **parameters**.
4. `for` $(i = t + 1;\ i \leq n;\ i{+}{+})$ (**Horizontal Growth**)
    1. Let π be the **set of** t**-way interactions** involving **parameter** P_i and $t - 1$ parameters among the first $i - 1$ parameters.
    2. `for` (test $\gamma = (v_1, v_2, \cdots, v_{i-1}) \in ts$)
        1. **Choose** a value v_i of P_i
        2. **Replace** γ with $\gamma' = (v_1, v_2, \cdots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π.
        3. **Remove** from π the interaction covered by γ'.

Greedy Algorithm – IPOG Strategy

Adding values for P_3 in ts:

$ts =$

P_1	P_2	P_3
0	0	
0	1	
1	0	
1	1	

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

Adding values for P_3 in ts:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	
1	0	
1	1	

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

Adding values for P_3 in ts:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	
1	1	

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

PLRG

Adding values for P_3 in ts:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

Adding values for P_3 in ts:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

1. Initialize **test set** *ts* to be an empty set.
2. **Parameters** are $P_1, P_2, \ldots, P_k$.
3. Add a test into *ts* for **all interactions of the first t parameters**.
4. `for` ($i = t + 1$; $i \leq n$; i++) (**Horizontal Growth**)
    1. Let π be the **set of t-way interactions** involving **parameter** P_i and $t - 1$ parameters among the first $i - 1$ parameters.
    2. `for` (test $\gamma = (v_1, v_2, \cdots, v_{i-1}) \in ts$)
        1. **Choose** a value v_i of P_i
        2. **Replace** γ with $\gamma' = (v_1, v_2, \cdots, v_{i-1}, v_i)$ such that γ' covers the most number of interactions in π.
        3. **Remove** from π the interaction covered by γ'.
5. `for` (interaction $\alpha \in \pi$) (**Vertical Growth**)
    1. `if` ($\exists$ a test covers α) **Remove** α from π.
    2. `else if` (possible) **Change** an existing test
    3. `else` **Add** a new test to cover α and **Remove** it from π.

Greedy Algorithm – IPOG Strategy

Extending ts:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

PLRG

Extending *ts*:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy Algorithm – IPOG Strategy

PLRG

Extending *ts*:

$ts =$

P_1	P_2	P_3
0	0	0
0	1	1
1	0	1
1	1	0
0	0	2
1	1	2

$\pi =$

P_1	P_2	P_3
0		0
	0	0
0		1
	0	1
0		2
	0	2
1		0
	1	0
1		1
	1	1
1		2
	1	2

Greedy vs. Meta-heuristic

- Simulated Annealing, a type of local search algorithm, has been proven to be effective against CIT.[3]

[3] **[SBSE'09]** B. J. Garvin et al. *"An improved meta-heuristic search for constrained interaction testing."*

Greedy vs. Meta-heuristic

- Simulated Annealing, a type of local search algorithm, has been proven to be effective against CIT.[3]
- The followings are size and time comparisons between the greedy algorithm and the meta-heuristic algorithm (average of 50 runs).

[3] **[SBSE'09]** B. J. Garvin et al. *"An improved meta-heuristic search for constrained interaction testing."*

Greedy vs. Meta-heuristic

- Simulated Annealing, a type of local search algorithm, has been proven to be effective against CIT.[3]
- The followings are size and time comparisons between the greedy algorithm and the meta-heuristic algorithm (average of 50 runs).

Size comparison

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

Time (sec.) comparison

Subject	Greedy	Meta-heuristic
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

[3] **[SBSE'09]** B. J. Garvin et al. *"An improved meta-heuristic search for constrained interaction testing."*

Summary

1. Equivalence Partitioning (EP)

2. Boundary Value Analysis (BVA)

3. Category Partition Method (CPM)

4. Combinatorial Testing (CT)
   - Covering Array (CA)
   - Fault Detection Effectiveness
   - Greedy Algorithm – IPOG Strategy
   - Greedy vs. Meta-heuristic

Next Lecture

- Random Testing

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr