Lecture 4 – Search Based Software Testing

AAA705: Software Testing and Quality Assurance

Jihyeok Park

PLRG

2026 Fall

Recall

- Graph Coverage
  - NC / EC / k-PC / PPC – the k knob, prime paths for loops
  - Data-flow: ADC / AUC / ADUPC; subsumption is about guarantees
- Logic Coverage
  - PC / CC / CoC $\rightarrow$ determination $\rightarrow$ ACC (MC/DC), GACC / RACC / CACC
- Measuring Coverage – percentages, infeasible TRs, instrumentation
- Is Coverage a Good Criterion?
  - Necessary, not sufficient; low coverage is reliable bad news
  - Uses: a **filter**, a **search gradient**, not a quality measure
- Coverage Beyond Code – neuron coverage, spec coverage

From One Bit to a Distance

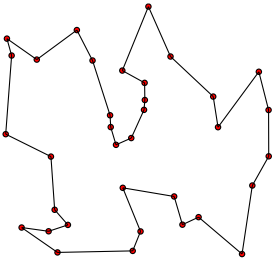
The fuzzer's signal was **one bit** per branch: taken or not. It kept an input when a bit was **new**. Last time we called that coverage as a **search gradient**.

- One bit has no **direction**. For `if (x == 42)` every input but one looks the same – a **needle in a haystack**.
- A **distance** does: $|x - 42|$ says which way to move, and how far.
- Today: turn each **test requirement** into a distance and let a **search algorithm** drive it to 0. First the algorithms, then the testing.

Contents

Contents

A Problem as a Search



Travelling salesman: visit every city once and return. With 20 cities there are $19!/2 \approx 6 \times 10^{16}$ tours – enumeration is out.

- **Representation** – what a candidate looks like: a permutation of the cities.
- **Fitness function** – how good it is: the tour length.
- **Operators** – how to move to a neighbor: swap two cities.

Nothing about salesmen is left. Any problem written as these three parts can be handed to the **same** search algorithms – that is **search based software engineering**.[1]

[1] **[IST'01]** M. Harman, B. F. Jones. "*Search-Based Software Engineering.*" **[CSUR'12]** M. Harman, S. A. Mansouri, Y. Zhang. "*Search-Based Software Engineering: Trends, Techniques and Applications.*"

Metaheuristics

A **metaheuristic** is a problem-independent recipe for finding a **good enough** solution:

- **approximate** – no guarantee of optimality,
- **randomized** – two runs give two answers,
- **iterative** – try, measure, adjust, try again.

Every one of them balances two forces:

- **Exploitation** – a good solution probably has good neighbors. Look around it.
- **Exploration** – the part of the space nobody has visited may hold something much better. Go there.

Too much exploitation and the search sits on the first hill it finds; too much exploration and it is random testing again. The algorithms today are different ways to set this balance.

Search in Software Engineering

problem	representation	fitness
test data generation (**today**)	an input vector	distance to the target branch
test suite minimization, prioritization	a subset, an ordering of tests	coverage kept, cost, time to first failure
fault localization	a suspiciousness formula, as an expression tree	how high the real fault ranks
program repair	a patched program	tests passed

- The search algorithm does not care which row it is on. All the work that **differs** between rows is in the **fitness function** – and that is where this lecture ends up.

Contents

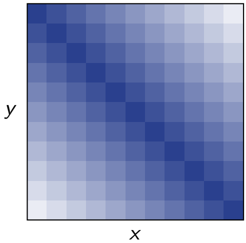
A Toy Problem

Find (x, y) with $x + y = 10$ for $0 \leq x, y \leq 10$ – but pretend we cannot solve equations. We may only **evaluate** candidates.

- **Representation**: a pair (x, y) of integers.
- **Fitness**: turn the equation into a distance to **minimize**,

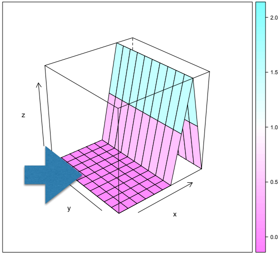
  $$f(x, y) = |10 - (x + y)|$$

  and $f = 0$ means solved.
- **Operators**: $x \pm 1$, $y \pm 1$.

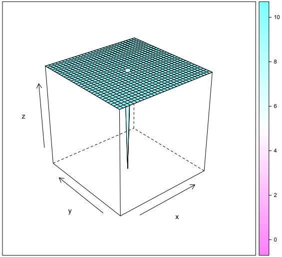


Plot f over every candidate and you get the **fitness landscape**: here a valley along the diagonal $x + y = 10$, sloping down from both corners.

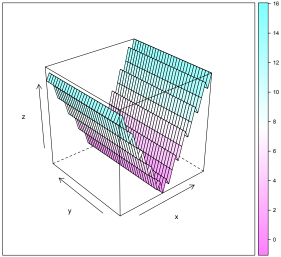
Three Shapes That Defeat Search



Plateau: all neighbors equal – no direction



Needle in a haystack: one good point, the rest flat

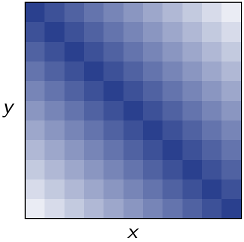


Rugged: many local optima – the nearest hill is rarely the highest

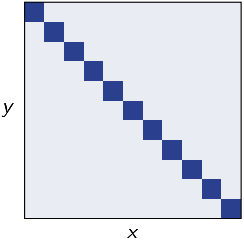
- On a plateau or a needle **no** algorithm beats random sampling: the fitness carries no information until you are already there.
- On a rugged landscape the algorithm matters – it decides how the search gets **off** a local optimum.

The Landscape Is Ours to Design

The landscape is not a property of the problem. It is a property of the **fitness function** we chose.



$f(x,y) = |10-(x+y)|$ – a slope from anywhere



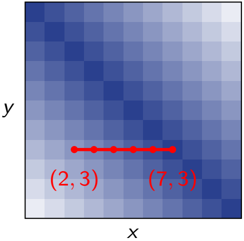
$f(x,y) = [\,x+y \neq 10\,]$ – the fuzzer's bit

- Same problem, same representation, same operators. Whether the search takes ten steps or forever was decided **before it started**.

Contents

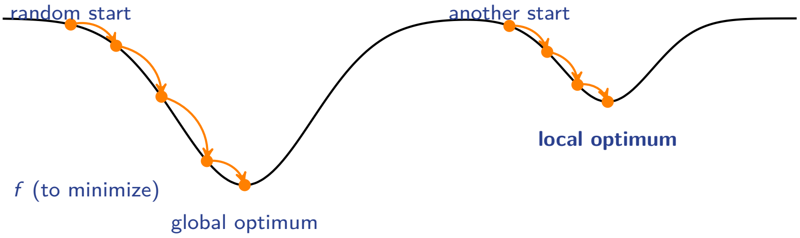
Hill Climbing

$s \leftarrow$ a random solution
loop
  $s' \leftarrow$ the best of *neighbors*(s)
  if $f(s') \geq f(s)$ **then return** s
  $s \leftarrow s'$



- From $(2, 3)$: $f = 5, 4, 3, 2, 1, 0$. Five moves, **20** evaluations. Random sampling needs about 11 tries to hit one of the 11 solutions out of 121.
- We **minimize** f throughout, so "ascent" means a **smaller** f and the search stops when no neighbor has one. Which improving neighbor to take: **steepest** (evaluate all, take the best), **first** improving (cheaper per step), or a **random** improving one.
- The operator sets the **radius**, not the size, of the search: 20 cities give 6×10^{16} tours, but any tour reaches any other in at most 190 adjacent swaps.

Local Optima



- A **local optimum**: no neighbor is better, and it is not the answer. Plain hill climbing stops here.
- Three ways out, and the rest of this section is about them:
  - **restart** from a new random point while budget remains,
  - **accept a worse move** sometimes – simulated annealing,
  - **remember** where you have been – tabu search.

Simulated Annealing

$s \leftarrow$ a random solution; $T \leftarrow T_0$
while $T > T_{\min}$ **do**
  $s' \leftarrow$ a random neighbor of s
  $\Delta \leftarrow f(s') - f(s)$
  if $\Delta < 0$ **or** $rand() < e^{-\Delta/T}$ **then**
    $s \leftarrow s'$
  $T \leftarrow cool(T)$

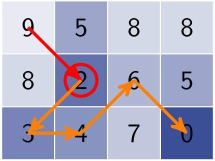
	$T = 10$	$T = 1$	$T = 0.1$
$\Delta = 1$	0.90	0.37	0.00005
$\Delta = 5$	0.61	0.007	≈ 0

probability of accepting a **worse** move

- Hot: almost a random walk (**explore**). Cold: hill climbing (**exploit**). The **cooling schedule** moves from one to the other – linear $T_0 - \alpha t$ or geometric $T_0\,\alpha^t$.[2]
- Logarithmic cooling $c/\log(t+d)$ has a theorem: with c large enough, the **probability** of sitting at a global optimum tends to 1 as $t \to \infty$. No guarantee for any finite run – and it needs more steps than trying every solution, so nobody uses it.[3]
- The directed fuzzer we saw did exactly this: the weight on the distance to the target **grew over time**.

[2] **[Science'83]** S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi. "*Optimization by Simulated Annealing.*"
[3] **[MOR'88]** B. Hajek. "*Cooling Schedules for Optimal Annealing.*"

Move to the **best neighbor that is not tabu** – even when it is worse. The **tabu list** holds the last k positions visited; here $k = 3$.[4]



8-neighborhood

at	tabu	allowed neighbors	move
2	$9, 2$	$3, 4, 5, 6, 7, 8, 8$	$\to 3$, worse – **best of 7**
3	$9, 2, 3$	$4, 8$	$\to 4$
4	$2, 3, 4$	$6, 7, 8$ (2 **is tabu**)	$\to 6$, worse
6	$3, 4, 6$	$0, 2, 5, 5, 7, 8, 8$	$\to 0$, **optimum**

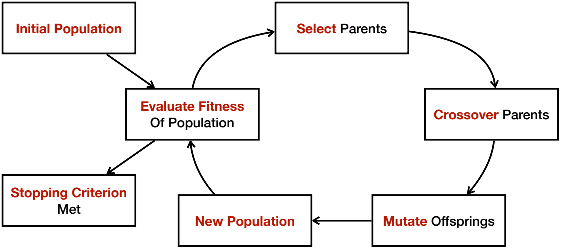
Hill climbing (red) stops at 2: all eight neighbors are worse.

- The search still looks only at neighbors and moves one step. What memory changes is **which neighbor counts as best**: at 4 the best neighbor is the 2 two steps back, and it is forbidden.
- k is the knob. With $k = 2$ the list at 4 is $\{3, 4\}$, so 2 is allowed and the search **cycles** $2, 3, 4, 2, 3, 4, \ldots$ – a loop of three needs three steps of memory. **Aspiration**: a tabu move is taken if it beats the best seen so far.

[4] **[C&OR'86]** F. Glover. "*Future Paths for Integer Programming and Links to Artificial Intelligence.*"

Contents

Evolving a Population



- Not one point but a **population**. Many regions of the space are sampled at once, and good **parts** of different solutions can be combined.[5]
- **Selection pressure** is the exploitation knob: the fitter an individual, the more children it gets. Too much and the population collapses onto one local optimum (**premature convergence**); too little and it drifts.
- **Elitism**: copy the best few unchanged, the best fitness never gets worse.

[5] **[Holland'75]** J. H. Holland. "*Adaptation in Natural and Artificial Systems.*"

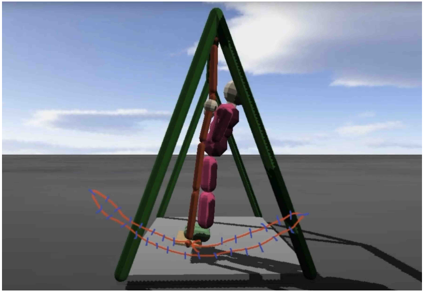
One Generation on the Salesman

	A	B	C	D	E
A	–	2	5	4	6
B		–	3	6	5
C			–	2	4
D				–	3
E					–

	tour	length
P_1	$A\ B\ D\ E\ C$	20
P_2	$D\ B\ C\ E\ A$	23
P_3	$A\ C\ D\ B\ E$	24
P_4	$B\ D\ A\ C\ E$	24

- **Select** by tournament of two: P_1 beats P_3, P_2 beats P_4.
- **Crossover** for permutations: keep a **slice** of one parent, fill the rest in the **other's order**. Slice $B\ C\ E$ from P_2, then A, D as they appear in P_1: $A\ B\ C\ E\ D$, length **16** – the optimum. The other child, $C\ B\ D\ E\ A$, has length 23.
- **Mutate**: swap two cities. $C\ B\ D\ E\ A$ with $D \leftrightarrow A$ becomes $C\ B\ A\ E\ D$ – also 16.
- Picking good parents and recombining their **pieces** is the **exploitation**; mutation is the **exploration** – the only source of material the population does not already have.

A GA Learns to Ride a Swing



Video: GA riding a swing

- **Representation**: one swing cycle cut into 32 time steps, one bit each – 1 stand, 0 sit.
- **Fitness**: how high the swing gets.
- **Operators**: flip a bit, cut two bit strings at a point and swap the tails.
- Nobody told it **when** to stand. The population found the rhythm.

Representation Decides the Operators

solution	encoding	crossover	mutation
a tour	permutation	keep a slice, fill the rest in the other parent's order	swap two cities
a program, a formula	syntax tree	swap subtrees	replace a node
a unit test	statement sequence	exchange statement sequences	add, delete, change a statement

- Crossover must produce a **valid** solution: one-point crossover of two tours duplicates cities, so permutations get their own operator.
- Mutation is the **only** source of new material; crossover only recombines what the population already has.
- The last two rows return later in the course: trees when formulas and programs are evolved, statement sequences when we generate unit tests.

Selection

Who gets to be a parent? Three individuals, fitness to **maximize**:

	fitness	rank	$P_{\text{proportional}}$	P_{rank}	$P_{\text{tournament}}$ $(k = 2)$
A	1	1	0.10	0.17	0.11
B	4	2	0.40	0.33	0.33
C	5	3	0.50	0.50	0.56

- **Fitness proportional** (roulette wheel): pick i with probability $f(i)/\sum_j f(j)$. Simple – but one very fit individual soon takes over the whole population.
- **Ranking**: sort first, then pick by **rank** instead of fitness. A fitness of 100 next to one of 5 is just first next to second, so the gap is capped.
- **Tournament**: draw k at random, keep the best. Only comparisons, so it works for minimization as is; a bigger k means more pressure. Most tools use this.

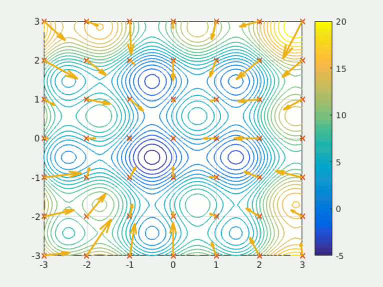
Contents

Particle Swarm Optimization (PSO)

A flock of birds. Each bird remembers **its** best spot p_i and hears the **flock's** best spot g.[6]

$$v_i \leftarrow ① \; w\, v_i + ② \; c_1 r_1 \,(p_i - x_i) + ③ \; c_2 r_2 \,(g - x_i) \qquad x_i \leftarrow x_i + v_i$$

- ① keep going ② toward my best ③ toward the flock's best
- GA **competes**: the weak are dropped. PSO **cooperates**: nobody is dropped.



Animation

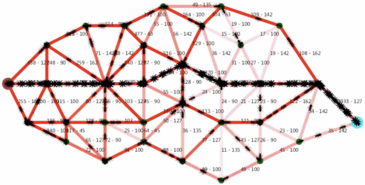
[6] **[ICNN'95]** J. Kennedy, R. Eberhart. "*Particle Swarm Optimization.*"

Ant Colony Optimization (ACO)

Ants leave **pheromone** on the path they walk and prefer edges with more of it. Short paths get walked more, so they get more.[7]

$$p_{ij} \propto \tau_{ij}^{\alpha} \cdot (1/d_{ij})^{\beta} \qquad \Delta\tau_{ij} = Q/L \qquad \tau_{ij} \leftarrow (1-\rho)\,\tau_{ij} + \sum \Delta\tau_{ij}$$

- Pheromone τ: **shared memory** of good edges.
- Evaporation ρ: **forgetting**, so the colony keeps exploring.
- Made for **paths on graphs**.



Video

[7] **[SMC'96]** M. Dorigo, V. Maniezzo, A. Colorni. "*Ant System: Optimization by a Colony of Cooperating Agents.*"

Which Algorithm?

	explores by	**exploits by**
hill climbing	random restarts	always taking a better neighbor
simulated annealing	accepting worse moves while hot	cooling
tabu search	forbidding recent moves	best allowed neighbor
genetic algorithm	population, mutation	selection, crossover
PSO / ACO	inertia / evaporation	global best / pheromone

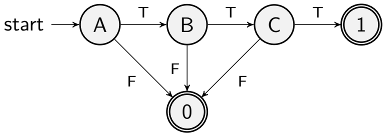
- **No free lunch**: averaged over **all** possible fitness functions, every search algorithm performs exactly the same. An algorithm wins only where its moves match the **shape** of the landscape.[8]
- So the question is never "which algorithm is best" but "what does **our** landscape look like". For testing, that is the next section.

[8] **[TEC'97]** D. H. Wolpert, W. G. Macready. "*No Free Lunch Theorems for Optimization*."

Contents

Test Generation as Search

```
int f(int a, int b) {
  if (a >= 10)          // A
    if (b == 2 * a)     // B
      if (a + b > 100)  // C
        return 1;       // target
  return 0;
}
```



- **Target**: the true branch of C. Which of the 2^{64} inputs (a, b) reach it?[9]
- **Representation**: (a, b). **Operators**: ± 1 on one variable. **Fitness**: how far the run stopped from the target.

[9] **[TSE'90]** B. Korel. "*Automated Software Test Data Generation.*"

Branch Distance

PLRG

How close did a **false** condition come to being true?[10]

condition	distance when false	condition	distance when false
$a = b$	$\|a - b\|$	$p \wedge q$	$d(p) + d(q)$
$a \neq b$	K	$p \vee q$	$\min(d(p), d(q))$
$a < b,\ a \leq b$	$a - b + K$	$\neg p$	push $\neg$ inward
$a > b,\ a \geq b$	$b - a + K$	`if (flag)`	K

- 0 when true. $K > 0$ (we use 1) keeps it positive when the operands are equal but the condition is false.
- Normalize into $[0, 1)$: $d/(d+1)$.[11]

[10] **[ASE'98]** N. Tracey, J. Clark, K. Mander, J. McDermid. "*An Automated Framework for Structural Test-Data Generation.*"
[11] **[STVR'13]** A. Arcuri. "*It Really Does Matter How You Normalize the Branch Distance in Search-Based Software Testing.*"

Approach Level, and the Fitness

$$fitness(t) = \underbrace{\text{approach level}}_{\text{how many ifs away}} + \underbrace{\text{normalize(branch distance)}}_{\text{at the if where the path left}}$$

```
int f(int a, int b) {
 if (a >= 10) // A
  if (b == 2 * a) // B
   if (a + b > 100) // C
    return 1;
 return 0;
}
```

(a, b)	left at	distance	fitness
$(3, 0)$	A	$10 - 3 + 1 = 8$	$2 + 8/9 = 2.889$
$(12, 0)$	B	$\lvert 0 - 24 \rvert = 24$	$1 + 24/25 = 1.960$
$(12, 24)$	C	$100 - 36 + 1 = 65$	$0 + 65/66 = 0.985$
$(40, 80)$	–	0	0

- Approach level $= 2, 1, 0$ for leaving at A, B, C. One `if` deeper always beats any progress on the current one.[12]

[12] **[IST'01]** J. Wegener, A. Baresel, H. Sthamer. "*Evolutionary Test Environment for Automatic Structural Testing.*"
[STVR'04] P. McMinn. "*Search-Based Software Test Data Generation: A Survey.*"

Alternating Variable Method (AVM)

Hill climbing on one variable at a time.

$t \leftarrow$ a random input
repeat
    for each variable x_i of t **do**
        exploratory: try $x_i - 1$ and $x_i + 1$
        while one of them improved f **do**
            pattern: keep going that way with
              steps $2, 4, 8, \ldots$ while improving
            then try $x_i \pm 1$ again
until no variable improved   ▷ local optimum
restart from a new random input

- Exploratory moves find the **direction**.
- Pattern moves cross large ranges **fast**.

AVM – From $(3, 0)$

move	(a, b)	left at	*fitness*
start	$(3, 0)$	A	2.889
explore a	$(2, 0)$ $(4, 0)$	A	▲ 2.900 ▼ 2.875
pattern $a + 2, +4$	$(6, 0)$ $(10, 0)$	A B	▼ 2.833 ▼ 1.952
pattern $a + 8$	$(18, 0)$	B	▲ 1.973 back to $(10, 0)$
explore a	$(9, 0)$ $(11, 0)$	A B	▲ 2.667 ▲ 1.957
explore, pattern b	$(10, 1) \ldots (10, 15)$	B	▼ 1.950 . . . ▼ 1.833
pattern $b + 16$	$(10, 31)$	B	▲ 1.917 back to $(10, 15)$
explore, pattern b	$(10, 16)$ $(10, 18)$ $(10, 19)$	B	▼ 1.800 ▼ 1.667 ▼ 1.500
explore b	$(10, 20)$	C	▼ 0.986
explore a, b	$(9, 20)$ $(11, 20)$ $(10, 19)$ $(10, 21)$	A B B B	▲ ▲ ▲ ▲ all worse

- **Stuck** at $(10, 20)$ after 30 evaluations: every one-variable move breaks B. The solutions lie on the line $b = 2a$, and no one-variable move walks along it.

AVM – Restart From $(12, 90)$

PLRG

move	(a, b)	left at	*fitness*
start	$(12, 90)$	B	1.985
explore a	$(11, 90)$ $(13, 90)$	B	▲ 1.986 ▼ 1.985
pattern a +2, +4, +8, +16	$(15, 90) \dots (43, 90)$	B	▼ 1.984 . . . ▼ 1.800
pattern a +32	$(75, 90)$	B	▲ 1.984 back to $(43, 90)$
explore, pattern a	$(44, 90)$ $(46, 90)$	B	▼ 1.667 ▲ 1.667
explore a	$(45, 90)$	–	▼ **0**

- **13** evaluations. But only starts with $b > 66$ succeed – about **one in three**.
- The fitness sees only the branch where the path **left**: while B is false it says nothing about $a + b > 100$.
- A solver that reads the **whole** path condition, $a \geq 10 \wedge b = 2a \wedge a + b > 100$, returns $(34, 68)$ at once. That is the next lecture.

The Flag Problem

```
bool ok = check(x);    // computed from x, somewhere else
...
if (ok) target();      // branch distance: K when false, always
```

- A **boolean** has no distance. The landscape of `if (ok)` is a **plateau**, however much slope there was inside `check`.
- **Testability transformation**: rewrite the program **for the search only**, so the distance inside `check` flows out to the `if`.[13]
- The algorithm cannot fix a flat landscape. The fitness function can.

[13] **[TSE'04]** M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, M. Roper. "*Testability Transformation.*"

One Branch at a Time, or the Whole Suite?

	individual	fitness
one target at a time (AVM)	one input	that branch's distance
whole test suite (EvoSuite)	a **set** of tests	sum of all branches' distances
many-objective (MOSA)	a set of tests	**each branch** its own objective

- One at a time wastes its budget on **infeasible** branches. Whole suite: up to **188**$\times$ the coverage on 1,741 classes.[14]
- MOSA keeps the best test **per branch** – the default in EvoSuite today.[15]

[14] **[TSE'13]** G. Fraser, A. Arcuri. "*Whole Test Suite Generation.*"

[15] **[TSE'18]** A. Panichella, F. M. Kifetew, P. Tonella. "*Automated Test Case Generation as a Many-Objective Optimisation Problem ...*"

Does the Search Beat Random?

- **1,000 Java classes**: GA and random search reach almost the **same coverage**. Most branches give the search **no guidance**.[16]
- Local search (AVM) suits some branches, the GA others; a **hybrid** of the two did best.[17]
- So the search pays off only where the fitness has a **slope**. Elsewhere it is random testing at a higher price.

[16] **[STVR'18]** S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn. "*Random or Evolutionary Search for Object-Oriented Test Suite Generation?*"

[17] **[TSE'10]** M. Harman, P. McMinn. "*A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search.*"

Summary

1. Search Based Software Engineering (SBSE)
2. Fitness Landscape
3. Local Search
   - Simulated Annealing
   - Tabu Search
4. Genetic Algorithms
   - Representation and Operators
   - Selection
5. Bio-inspired Algorithms
   - Particle Swarm Optimization (PSO)
   - Ant Colony Optimization (ACO)
6. Search Based Software Testing (SBST)
   - Fitness for Branch Coverage
   - Alternating Variable Method (AVM)
   - From One Branch to a Test Suite

Next Lecture

- Dynamic Symbolic Execution

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