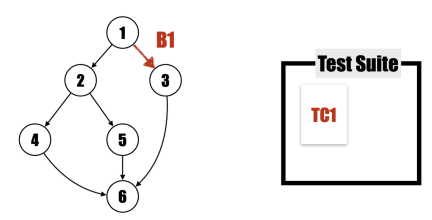
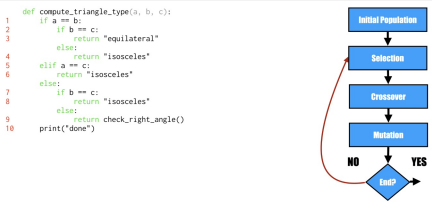
Single target approach



- Targeting one branch/statement at a time.
- Chromosome = test case.
- Running genetic algorithm multiple times.

Fitness(b1) = approach_level(b1) + branch-_distance(b1)

Single-target approach



Single-target approach:

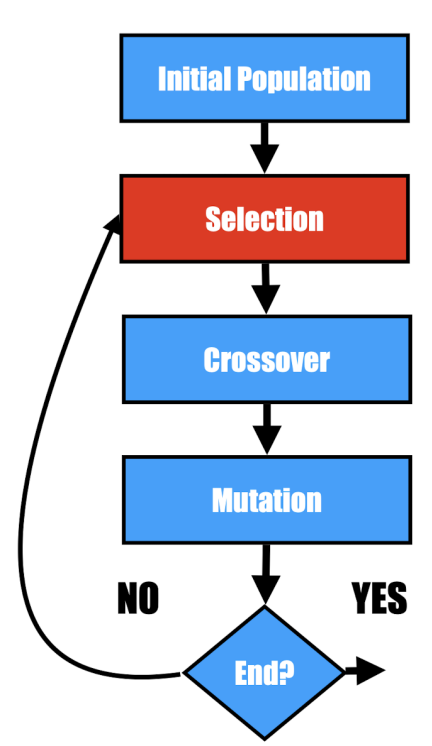
1. Select one target (statement or branch) to cover.
2. Run GAs until reaching the maximum search budget (max iterations) or when the target is covered (fitness function = 0).
3. Repeat from step 1. for a new target (statement or branch).

One-point crossover

	Parents	Cut-point	Offsprings
x1 = (2,2,2)	-	-	x1 = (2,2,2)
x2 = (3,5,4)	x2, x5	1	x2 = (2,2,3)
x3 = (2,2,6)	x3, x8	2	x3 = (2,3,4)
x4 = (2,3,7)	x4, x6	2	x4 = (2,3,5)
x5 = (7,2,3)	SEL	1	x5 = (7,3,4)
x6 = (3,4,5)	SEL	2	x6 = (3,4,7)
x7 = (3,5,7)	-	-	x7 = (3,5,7)
x8 = (6,4,6)	SEL	2	x8 = (6,4,6)

One-point crossover (probability = 0.8). It takes two parents and cuts their chromosome strings at some randomly chosen position and the produced substrings are then swapped to produce two new full-length chromosomes.

Selection



Select the best parent in the population to generate offsprings.

Automated test case generation

Ingredients:

1. Solution representation.
2. Fitness function.
3. Selection.
4. Reproduction (crossover and mutation).

The goal of automated test case generation is the automatic generation of test cases using genetic algorithms in order to achieve the maximum statement coverage.

No free lunch theorem

Theorem: given a search budget m (e.g. time), the probability of reaching the near optimal value f* using algorithm A is the same as the probability of obtaining the same near-optimal value using another arbitrary algorithm B.

Genetic algorithms (GAs)

Natural selection: individuals that best fit the natural environment survive (worst die).

Reproduction: survived individuals generate offsprings (next generation).

Mutation: offsprings inherit properties of their parents (with some mutations).

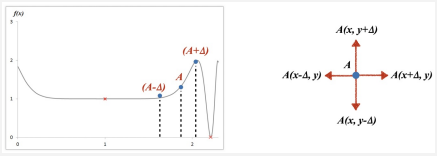
Iteration: generation by generation the new offspring fit better the environment than their parents.

Hill climbing variables

Random-restart hill climbing: random restart when no improving solution is found (or improvement is small).

Stochastic hill climbing: while classic hill climbing always picks the best neighbour (the steepest uphill move), stochastic hill climbing randomly chooses amongst the uphill moves.

Hill climbing



1. Start with a random solution A.
2. Try some 'nearby' solutions (A + delta) and (A - delta).
3. Take the best solution among A, (A + delta) and (A - delta).
4. Repeat from step 1.

Local search => converges toward local optimum.

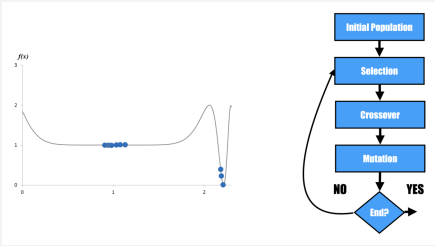
Search-based software engineering

Search-based software engineering is the application of meta-heuristic search-based optimisation techniques to find near-optimal solutions in software engineering problems.

Key elements:

- **Representation:** represent a candidate solution for a problem.
- **Fitness function:** distinguishes good solutions from bad solutions; shapes the fitness landscape.
- **Manipulation operators:** move from a solution to another.

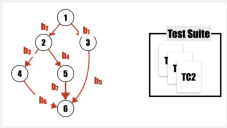
Genetic algorithms



Pseudo-code:

Pop = {x0, ..., xN}
x_opt = NULL
LOOP
assignFitness(Pop)
x_opt = best(Pop, x_opt)
Pop = select(Pop)
Pop = reproduce(Pop)
END LOOP
RETURN x_opt

Many objective optimisation



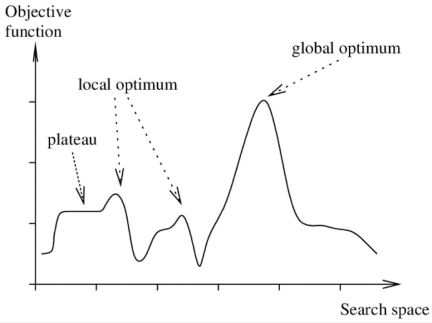
- Targeting all branches at once.
- Chromosome = test case.
- Running many-objective genetic algorithms.
- Usage of the archive.

Cheatography

ASE Notes Second Copy Cheat Sheet

by User23456 via cheatography.com/151394/cs/32759/

Disadvantages of hill climbing



Local maximum:

- A state is better than all of its neighbours but not better than some other states.

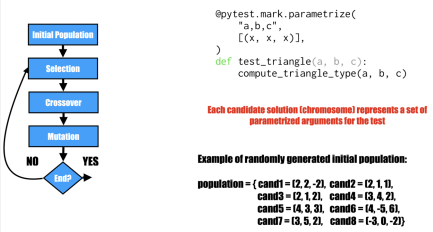
Plateau:

- A flat area of the search space in which all neighbouring states have the same value.

Ridge:

- Orientation of the high region that, compared to the set of available moves, make it impossible to climb up.

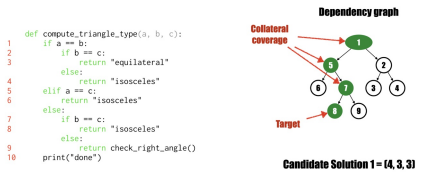
Solution representation



The chromosome used for test case generation is the **input vector** (sequence of input values used by the test case to run the program) which may be fixed length or variable length).

Each candidate solution (chromosome) represents a set of parameterised arguments for the test.

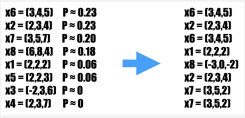
Collateral coverage



When a target is covered accidentally in case generation cycle, it is removed from the list of yet uncovered target and the corresponding (covering) test case is stored to form the final suite.

Collateral coverage: if a test case involved in collateral coverage are not detected and stored, it can be shown that random testing performs asymptotically better than search-based testing.

Roulette wheel selection



Roulette wheel selection:

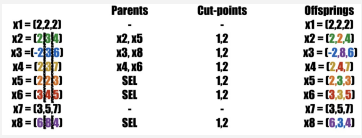
1. Assign to each test case a probability equal to 1/f (inverse of the fitness score).
2. Normalise the obtained probability.
3. Each test case has a probability to be selected that is proportional to its slice in the roulette wheel.

Tournament selection

Binary tournament selection:

1. Randomly choose pairs of test cases (solutions).
2. Select the fittest (better) individuals from each pair.

Multi-point crossover



It is an extension of the single point crossover that uses multiple cut-points instead of a single one. It takes two parents and cuts their chromosome strings at k randomly chosen cut positions and the produced substrings are then swapped over k point to produce two new full-length chromosomes.

By **User23456**

cheatography.com/user23456/

Not published yet.

Last updated 23rd June, 2022.

Search algorithms

Optimisation algorithms involve applying a search-algorithm to solve (minimise or maximise) such as the fitness function => find the best solution.

Local search:

- Hill climbing.
- Simulated annealing.
- Tabu search.

Typically single-solution.

Global search:

- Random search.
- Genetic algorithms.
- Particle swarm optimisation.

Typically population-based.

Many objective optimisation

Given: $B = \{b_1, \ldots, b_m\}$ branches of a program.
Find: test cases $T = \{t_1, \ldots, t_n\}$ minimizing the following fitness objectives:

$$\min f_1(T) = \text{approach\_level}(b_1) + \text{branch\_distance}(b_1)$$
$$\min f_2(T) = \text{approach\_level}(b_2) + \text{branch\_distance}(b_2)$$
$$\ldots$$
$$\min f_m(T) = \text{approach\_level}(b_m) + \text{branch\_distance}(b_m)$$

$$\text{Fitness} = \langle f_1, \ldots, f_m \rangle$$

Limitations of single targets

Some coverage targets may be feasible.
Some coverage targets may be very difficult to achieve.
Since a limited search budget is available for test case generation:

- Infeasible targets may use the search budget without reaching any target.
- Difficult target may use most of search budget, leaving lots of easier coverage target uncovered.
- The order in which targets are considered affects the final results.

Example

```
def compute_triangle_type(a, b, c):
    if a == b:
        if b == c:
            return "equilateral"
        else:
            return "isosceles"
    elif a == c:
        return "isosceles"
    else:
        if b == c:
            return "isosceles"
        else:
            return check_right_angle()
print("done")
```

Dependency graph

cand1 = (4, 3, 3)
cand2 = (2, 2, 4)
cand3 = (10, 1, 10)
cand4 = (4, 4, 4)
cand5 = (1, 2, 3)

The final test suite consists of all chromosome that have been found to cover (even accidentally) one or more yet to cover statements.

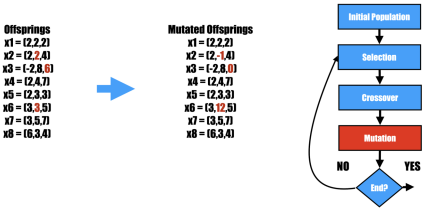
Many objective optimisation

Given: $B = \{b_1, \ldots, b_m\}$ branches of a program.
Find: test cases $T = \{t_1, \ldots, t_n\}$ minimizing the following fitness objectives:

$$\min f_1(T) = \text{approach\_level}(b_1) + \text{branch\_distance}(b_1)$$
$$\min f_2(T) = \text{approach\_level}(b_2) + \text{branch\_distance}(b_2)$$
$$\ldots$$
$$\min f_m(T) = \text{approach\_level}(b_m) + \text{branch\_distance}(b_m)$$

$$\text{Fitness} = \langle f_1, \ldots, f_m \rangle$$

Mutation



Mutation: randomly change some genes (elements within each chromosome).
Mutation probability: 1/n where n = chromosome length.

Branch distance

Condition c = atomic predicate	Distance BD(c) = d / (d + 1)
a	d = {0 if a == true; K otherwise}
!a	d = {K if a == true; 0 otherwise}
a == b	d = {0 if a == b; abs(a - b) + K otherwise}
a != b	d = {0 if a != b; K otherwise}
a < b	d = {0 if a < b; a - b + K otherwise}
a <= b	d = {0 if a <= b; a - b + K otherwise}
a > b	d = {0 if a > b; b - a + K otherwise}
a >= b	d = {0 if a >= b; b - a + K otherwise}

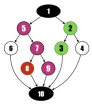
Branch distance: given the first control node where the execution diverges from the target t, the predicate at such node is converted to a distance (from taking the desired branch), normalised between 0 and 1. Such a distance measure how far the test case is from taking the desired branch. For boolean and numerical variables, the table is shown.

Approach level

Given the execution trace obtained by running program P with input vector x, the approach level is the minimum number of control nodes between an executed statement and the coverage target t.

Computing fitness

```
def compute_triangle_type(a, b, c):
    if a == b:
        if b == c:
            return "equilateral"
        else:
            return "isosceles"
    elif a == c:
        return "isosceles"
    else:
        if b == c:
            return "isosceles"
        else:
            return check_right_angle()
print("done")
```



c == 1000 instead of 10 => BD = abs(-2-1000) = 1002

cand4 = (2, 2, 2) Path(cand1) = <1, 2, 3, 10> f = AL + BD = 2 + 1 = 3

cand3 = (2, -2, 10) Path(cand3) = <1, 5, 7, 9, 10> f = AL + BD = 0 + 1002= 1002

Single-target approach, 8 is the target.

Automated test case generation

```
def method(a, b):
    if a < 0:
        print("a is negative")
    if b < 0:
        print("b is negative")
    print(a, b)
```

	Covered Targets	Test Cases
Path coverage	<1, 3, 5> <1, 2, 3, 5> <1, 3, 4, 5>, <1, 2, 3, 4, 5>	method(1, 1) method(-1, 1) method(1, -1) method(-1, -1)

```
@pytest.mark.parametrize(
    "a,b",
    [(1, 1), (-1, 1), (1, -1), (-1, -1)],
)
def test_method(a, b):
    method(a, b)
```

The goal of evolutionary testing is to archive some sort of code coverage.

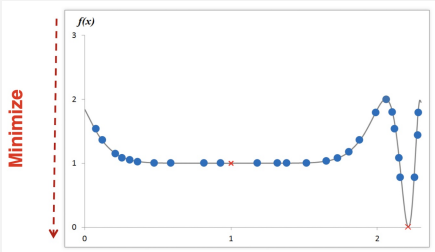
Genetic algorithms

Genetic algorithms must maintain a balance between the exploitation and exploration as to increase the probability of finding the optimal solution.

Exploitation: find nearby better solutions by promoting beneficial aspects (genes) of existing solutions.

Exploration: find new solutions in different regions of the search space by generating solutions with new aspects.

Random search

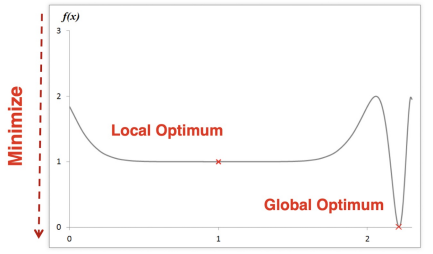


Random search involves trying random values and taking the best trial.
Trivial but effective in some cases:
- Fast generator of solution.
- Lot of time needed (budget).
- Used as baseline.

'Sophisticated' random search exists.

Search problem

$$\forall x \in X : f(x^*) \leq f(x)$$



Find a value (solution) x* which minimizes (or maximizes) the objective function f over a search space X.
Search space: all possible solutions.

By **User23456**
cheatography.com/user23456/

Not published yet.
Last updated 23rd June, 2022.
Page 5 of 5.