The University Of Sheffield.

Search-Based Testing

Phil McMinn

University of Sheffield, UK

Overview

How and why Search-Based Testing Works

Examples

- Temporal, Functional, Structural
- Applications in Mutation Testing
- Test suite prioritisation

Search-Based Test Data Generation

- Testability Transformation
- Input Domain Reduction
- Empirical and Theoretical Studies

Future Directions

Acknowledgements

The material in some of these slides has kindly been provided by:

Mark Harman (KCL/UCL)

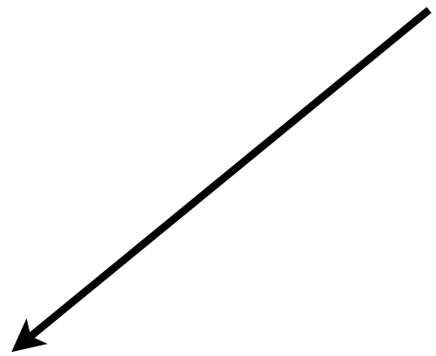
Joachim Wegener (Berner & Matner)

Conventional Testing

manual design of test cases / scenarios

Conventional Testing

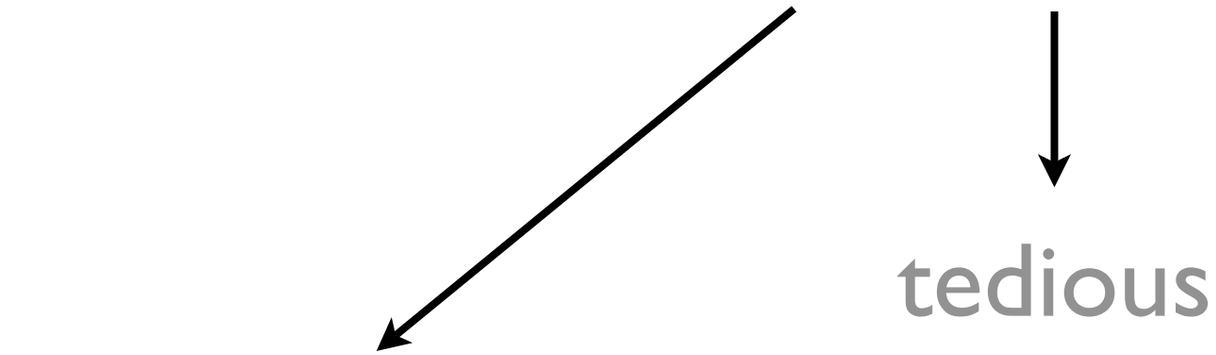
manual design of test cases / scenarios



laborious,
time-consuming

Conventional Testing

manual design of test cases / scenarios

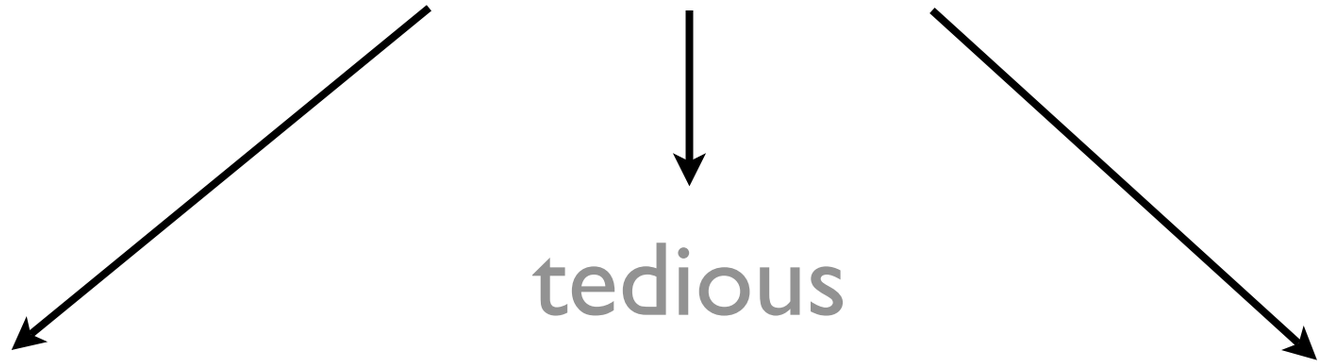


laborious,
time-consuming

tedious

Conventional Testing

manual design of test cases / scenarios



tedious

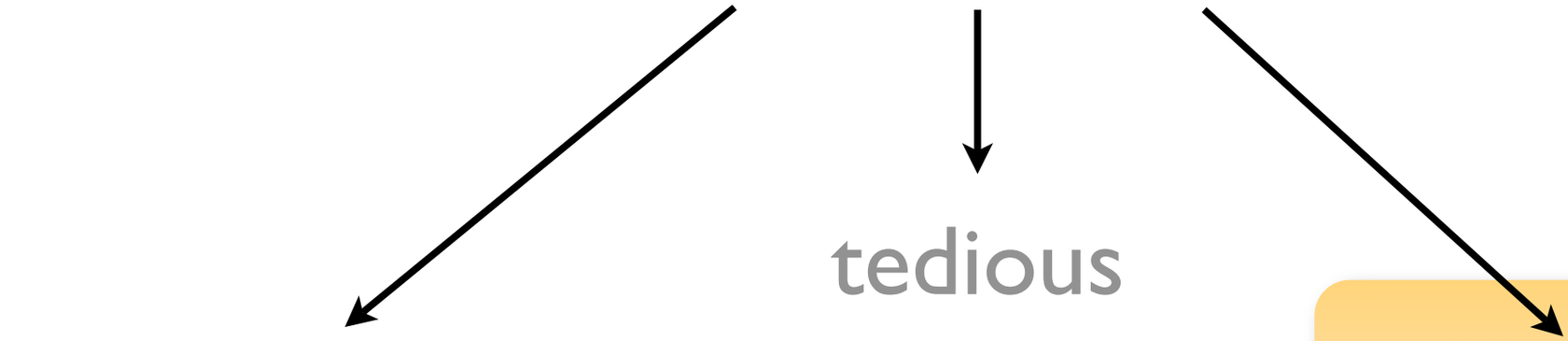
laborious,
time-consuming

difficult!

(where are the faults ???)

Conventional Testing

manual design of test cases / scenarios



laborious,
time-consuming

tedious

difficult!

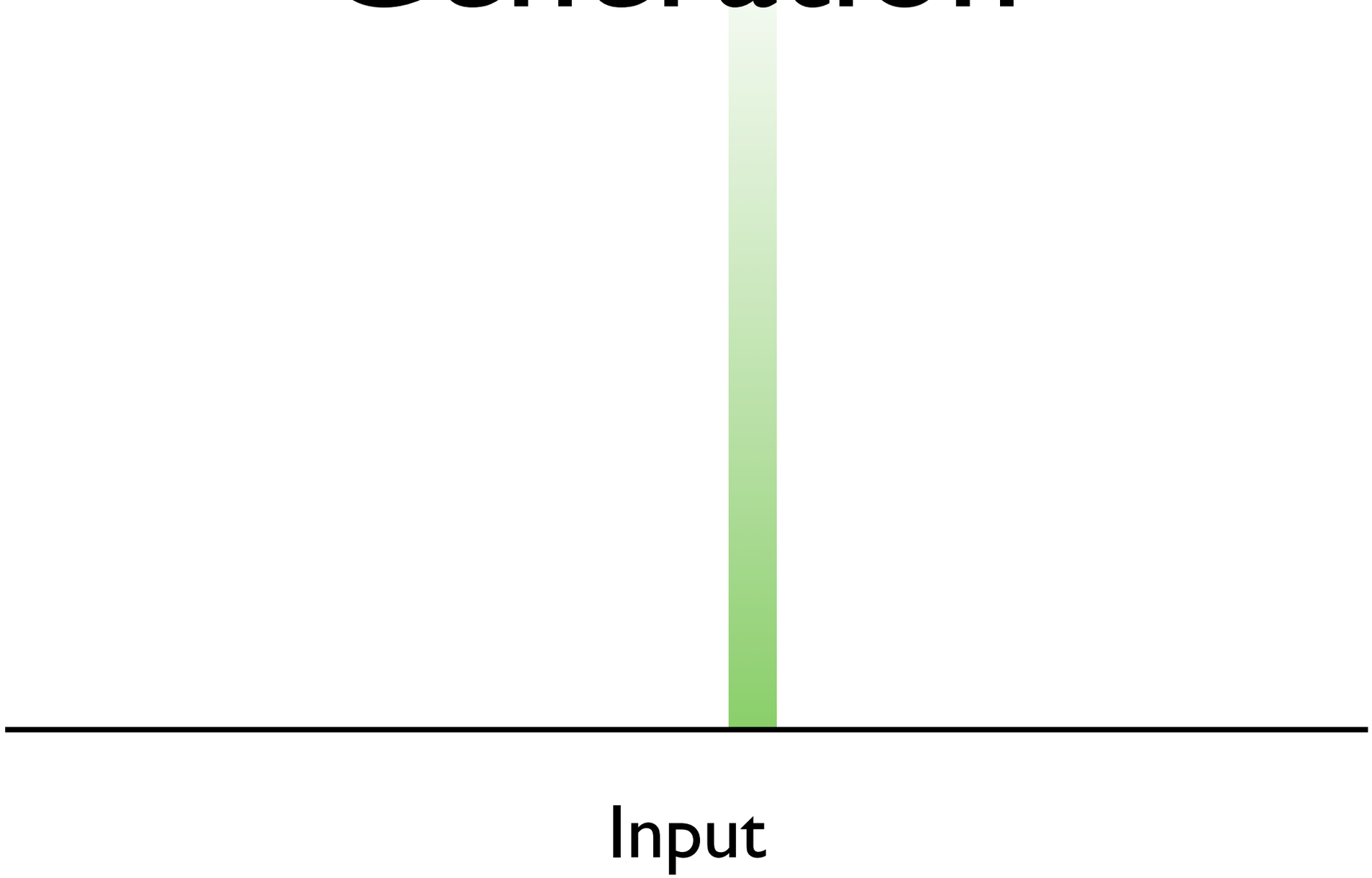
(where are the faults ???)

the hard part...

Random Test Data Generation

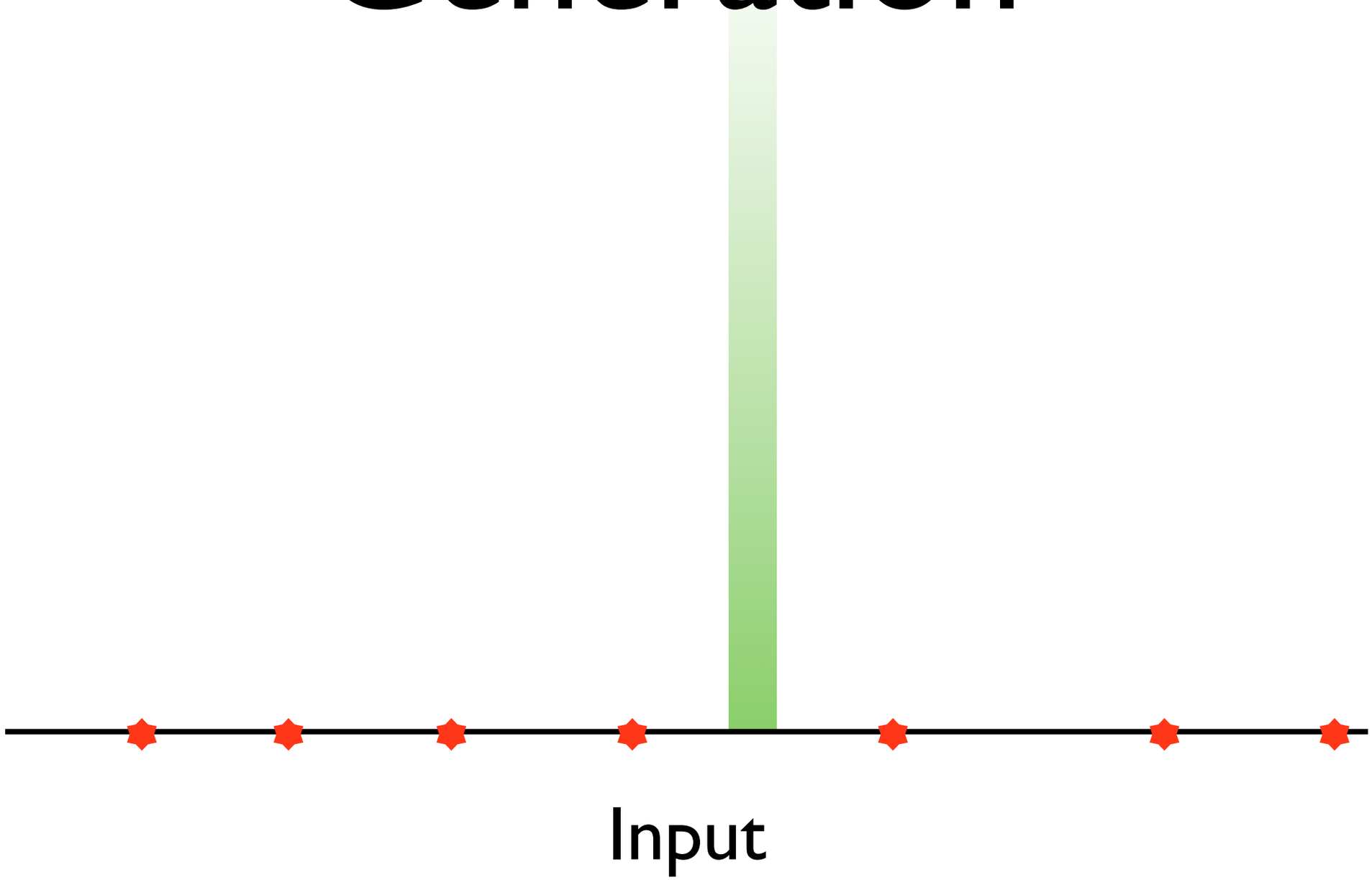
Input

Random Test Data Generation



Input

Random Test Data Generation



Search-Based Testing is an automated *search* of a potentially large input space

the search is guided by a problem-specific 'fitness function'

Search-Based Testing is an automated *search* of a potentially large input space

the search is guided by a problem-specific 'fitness function'

the fitness function guides the search to the test goal

Fitness Function

The fitness function scores different inputs to the system according to the test goal

Fitness Function

The fitness function scores different inputs to the system according to the test goal

which ones are ‘good’
(that we should develop/evolve further)

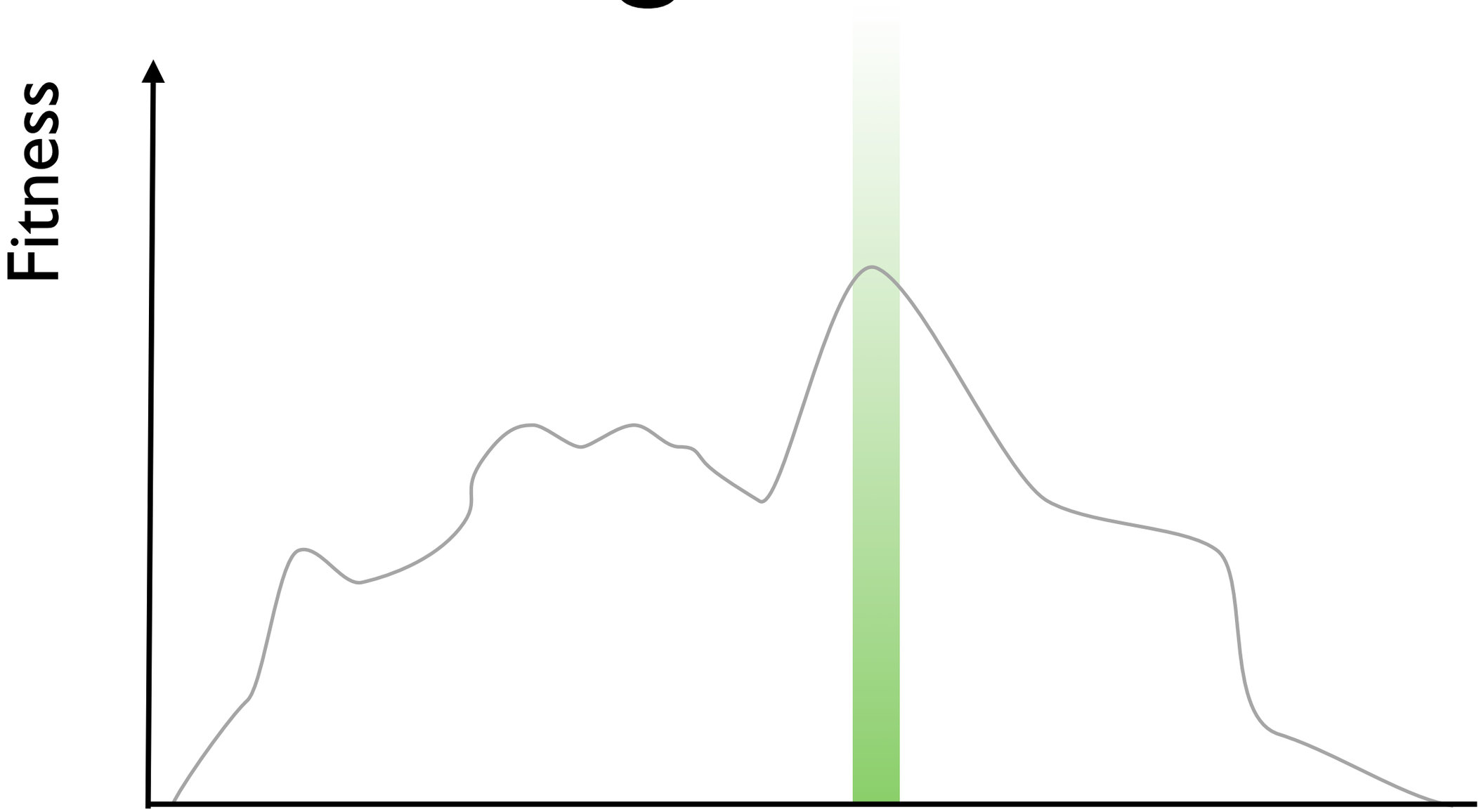
Fitness Function

The fitness function scores different inputs to the system according to the test goal

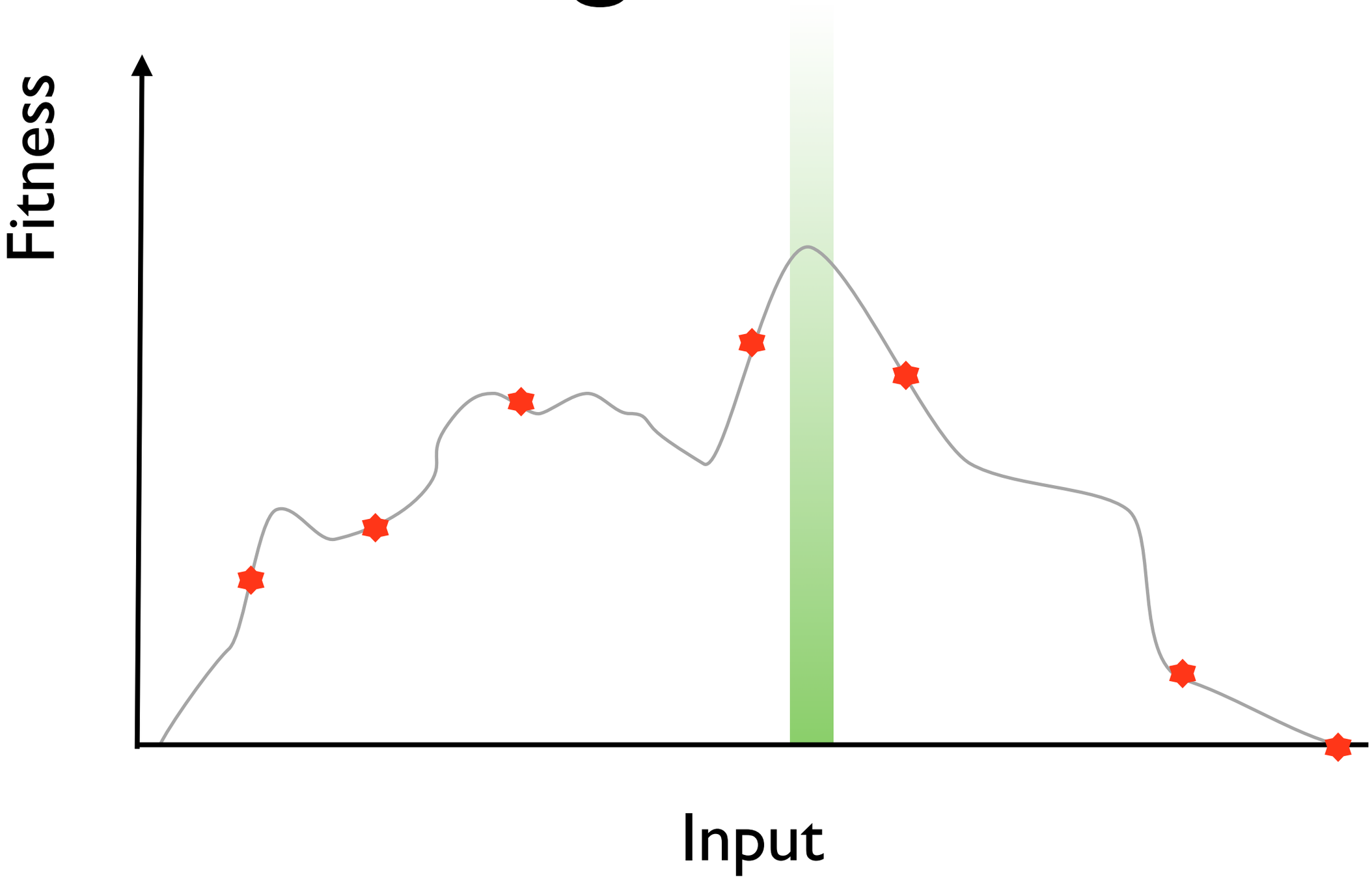
which ones are ‘good’
(that we should develop/evolve further)

which ones are useless
(that we can forget about)

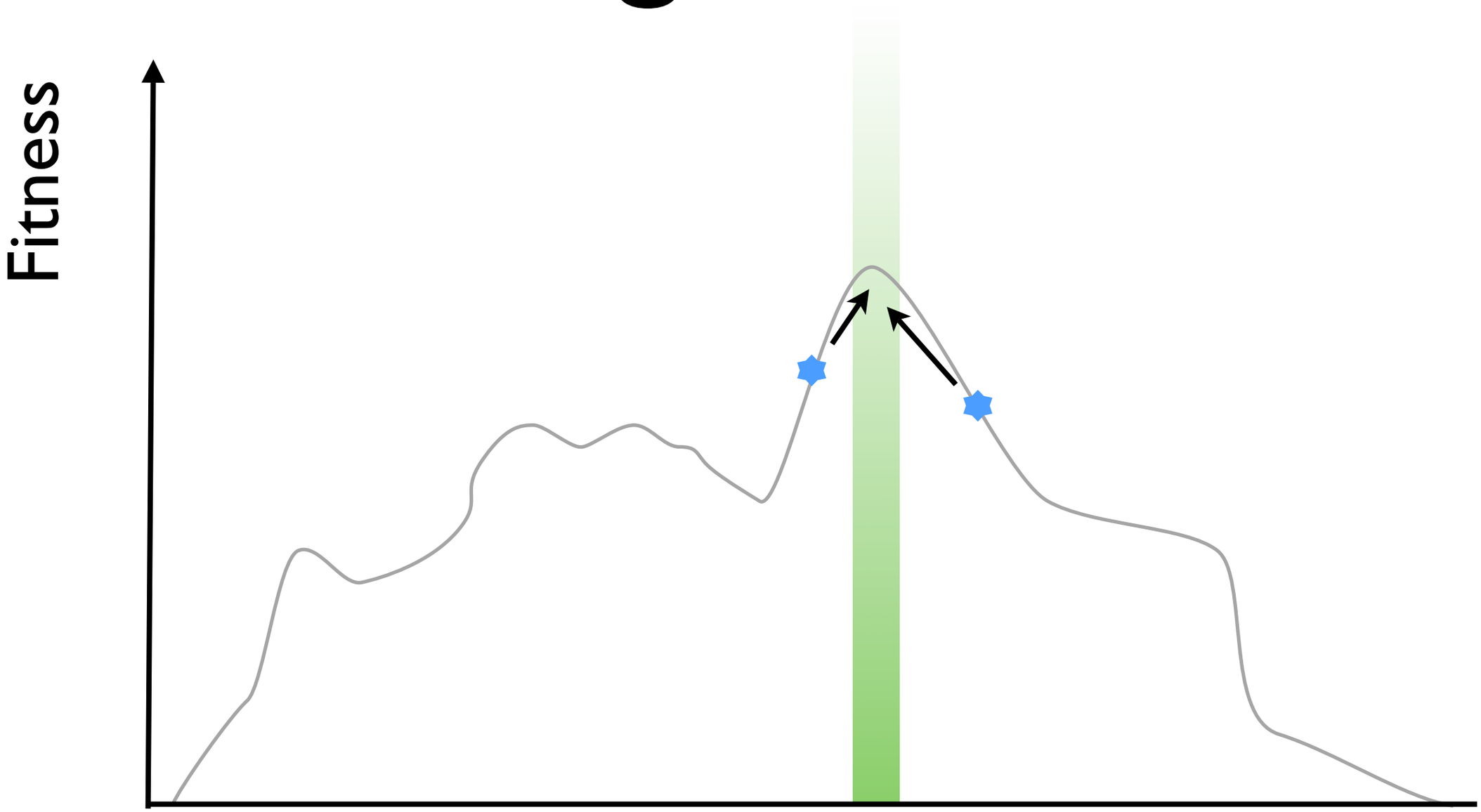
Fitness-guided search



Fitness-guided search



Fitness-guided search



First publication on SBST

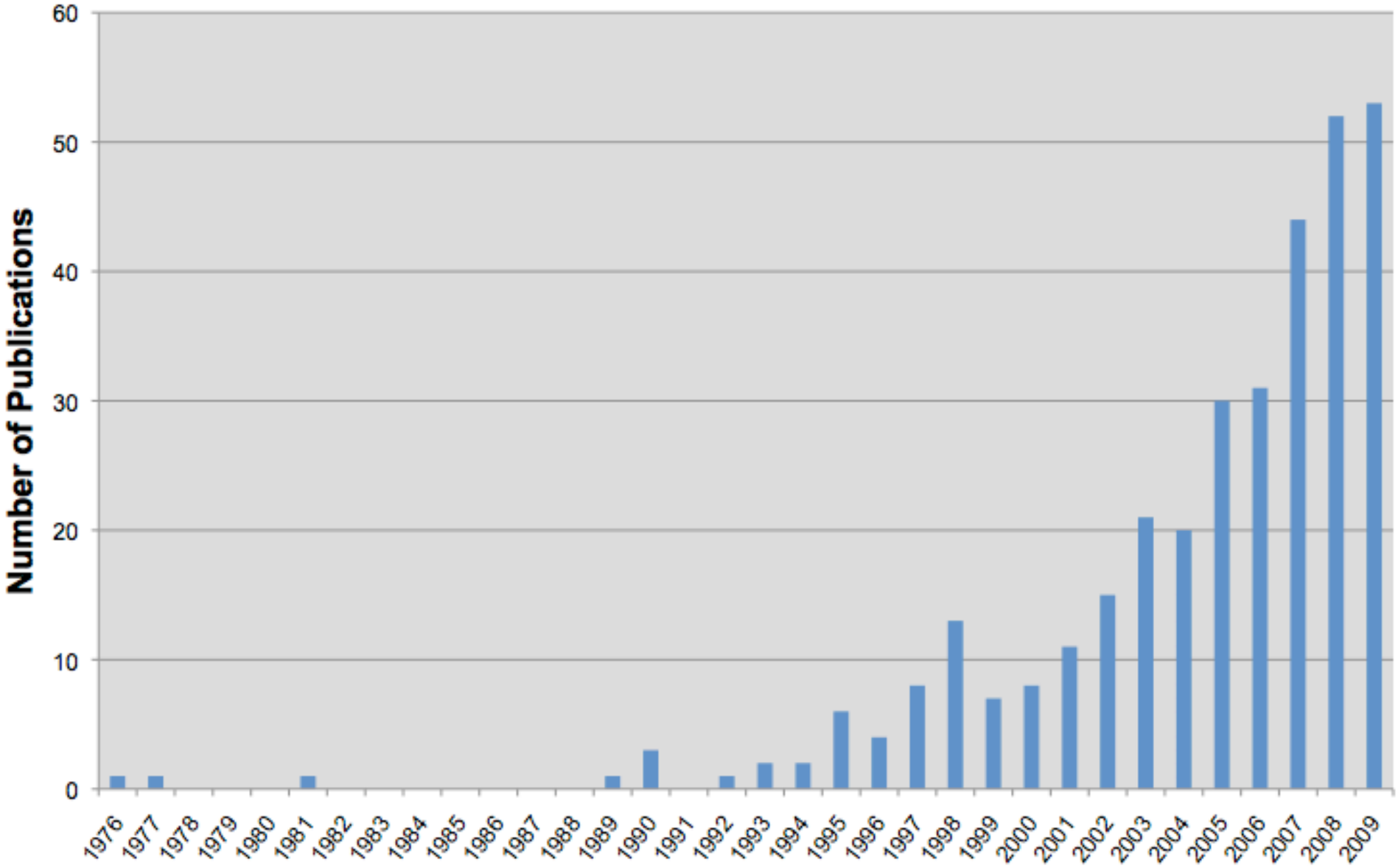
Automatic Generation of Floating-Point Test Data
IEEE Transactions on Software Engineering, 1976

Webb Miller David Spooner

Winner of the 2009 Accomplishment by a Senior Scientist Award
of the International Society for Computational Biology

2009 Time100
Scientists and Thinkers

Publications since 1976



source: SEBASE publications repository http://www.sebase.org

International Search-Based Testing Workshop

co-located with ICST

4th event at ICST 2011

next year

check websites for
submission deadlines etc.:

http://sites.google.com/site/icst2011/

http://sites.google.com/site/icst2011workshops/

International Symposium on Search-Based Software Engineering

www.ssbse.org

Benevento, Italy.
7th - 9th September 2010

International Symposium on Search-Based Software Engineering

www.ssbse.org

2011: Co-location with FSE. Szeged, Hungary

Phil McMinn
General Chair

Myra Cohen and Mel Ó Cinnéide
Program Chairs

Fitness Functions

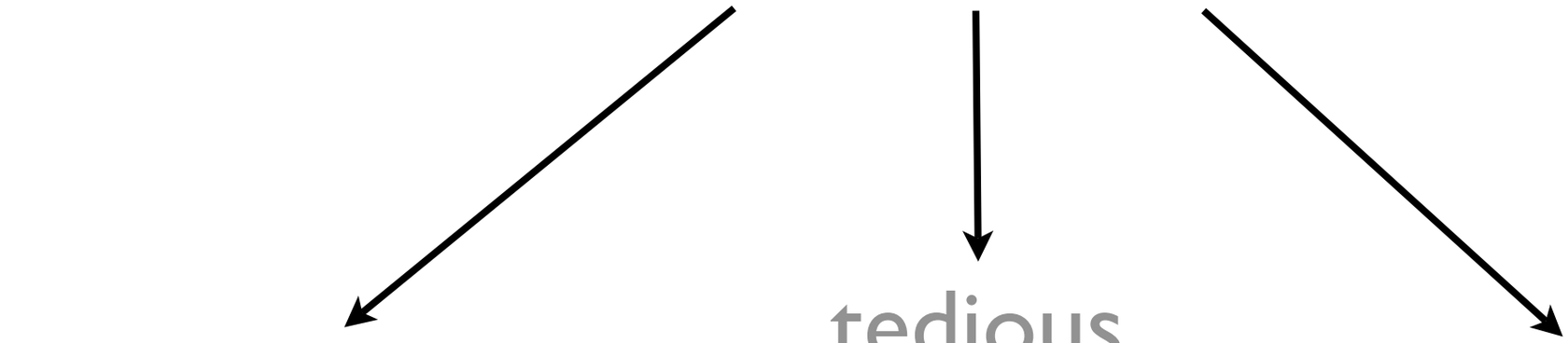
Often easy

We often define metrics

Need not be complex

Conventional testing

manual design of test cases / scenarios



laborious,
time-consuming

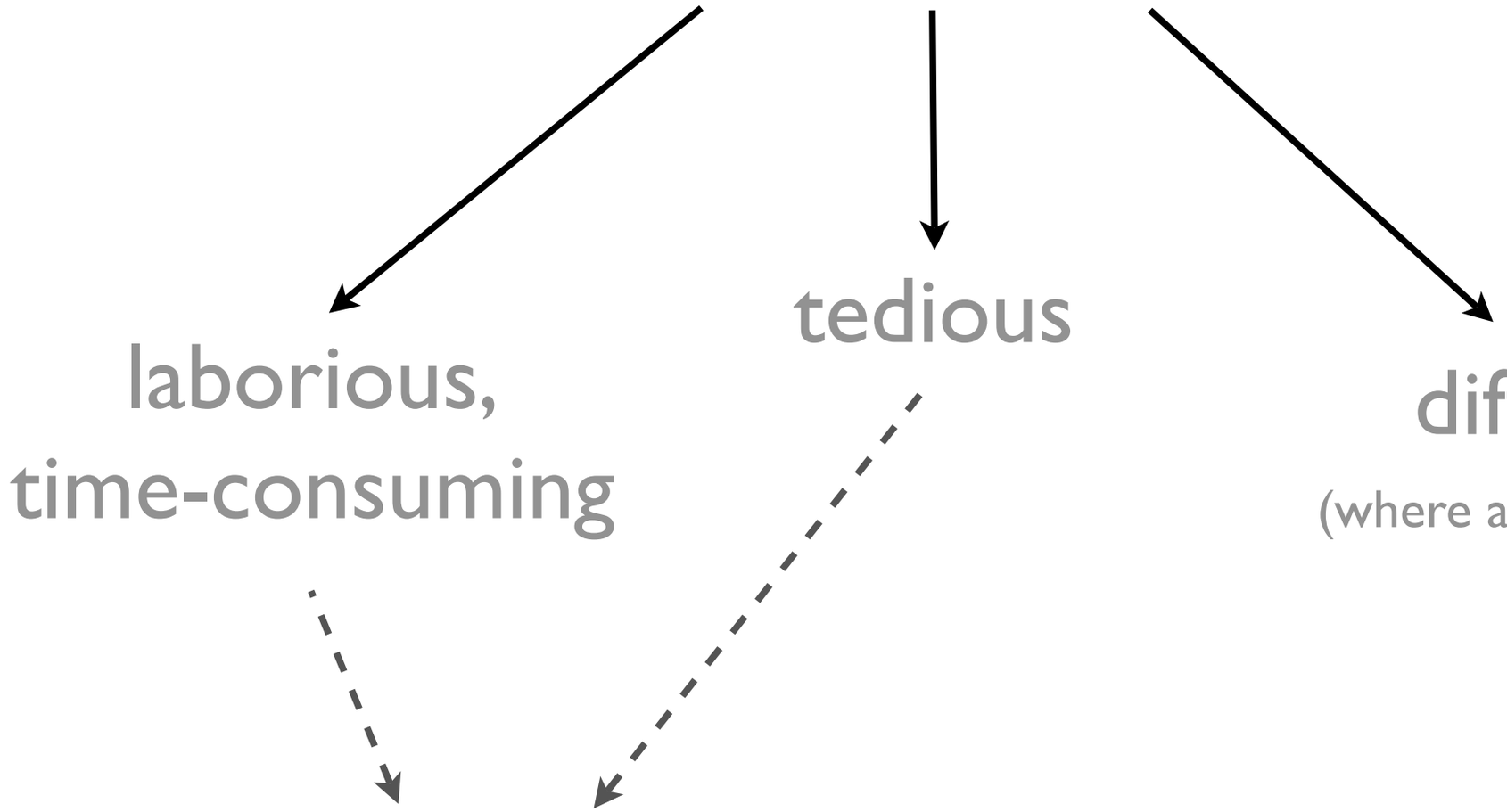
tedious

difficult!

(where are the faults ???)

Conventional testing

manual design of test cases / scenarios



Search-Based Testing:

automatic - may sometimes be time consuming, but it is not a human's time being consumed

Conventional testing

manual design of test cases / scenarios

laborious, time-consuming

tedious

difficult!

(where are the faults ???)

Search-Based Testing:

automatic - may sometimes be time consuming, but it is not a human's time being consumed

Search-Based Testing:

a good fitness function will lead the search to the faults

Generating vs Checking

Conventional Software Testing Research

Write a method to construct test cases

Search-Based Testing

Write a method
to determine how good a test case is

Generating vs Checking

Conventional Software Testing Research

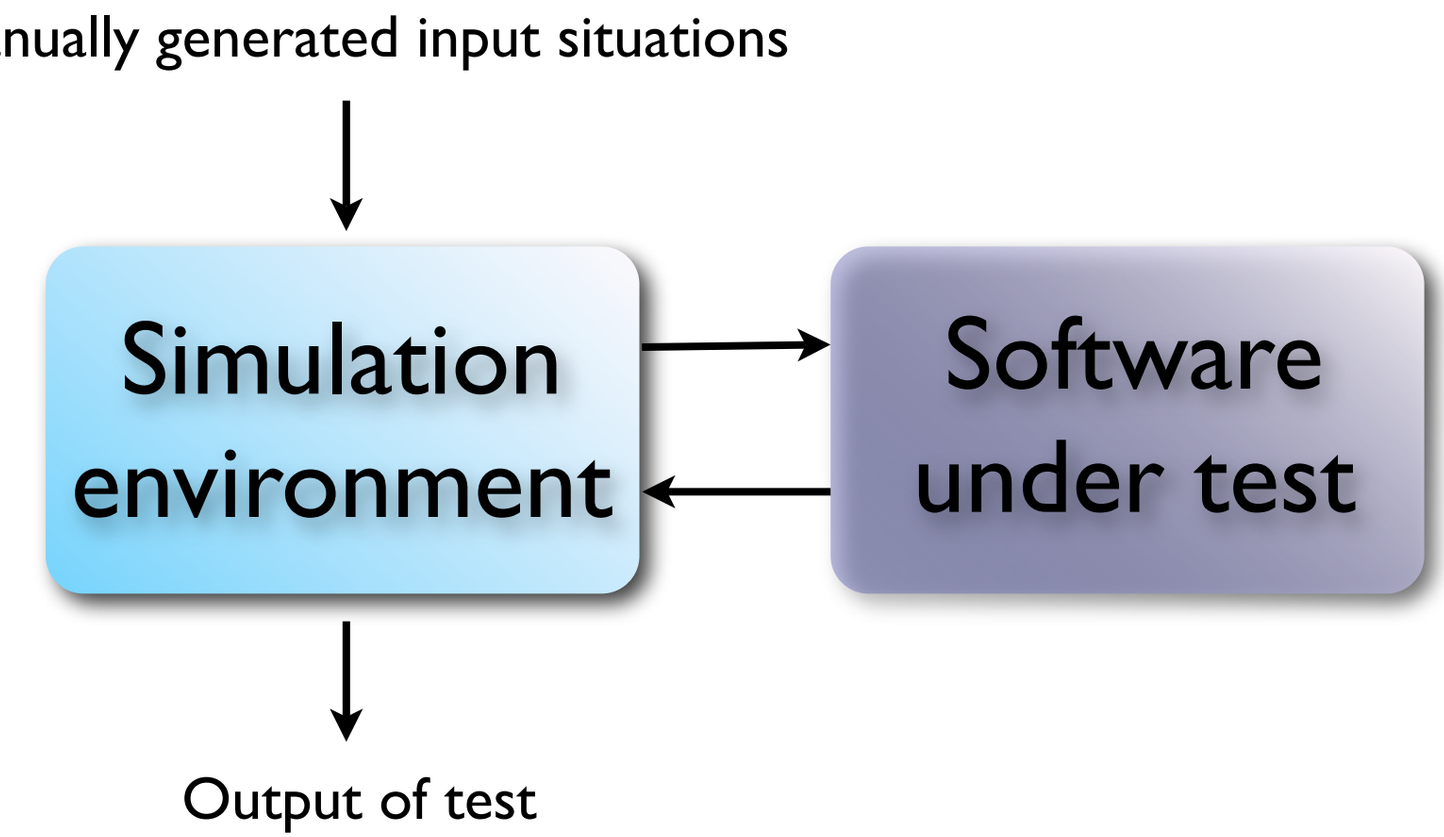
Write a method to construct test cases

Search-Based Testing

Write a fitness function
to determine how good a test case is

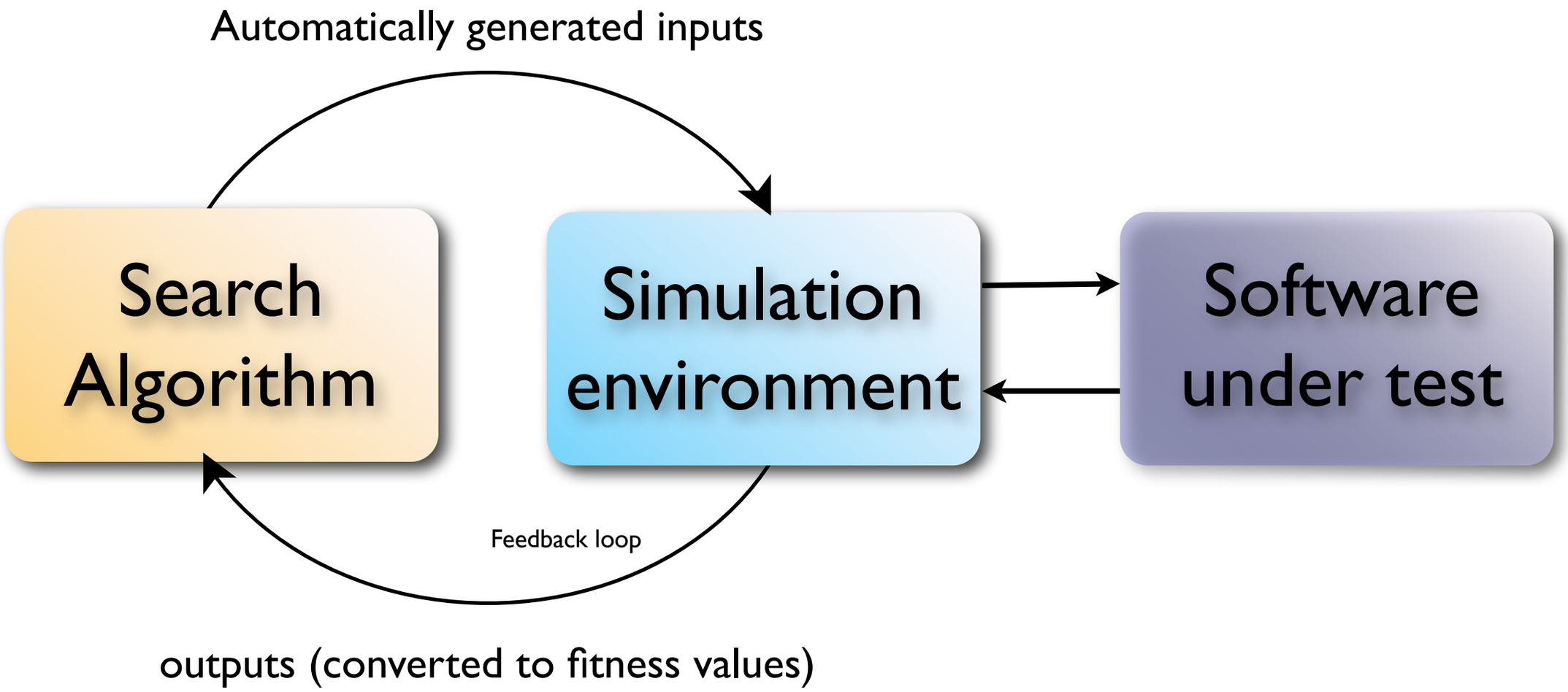
Test setup

Usual approach to testing

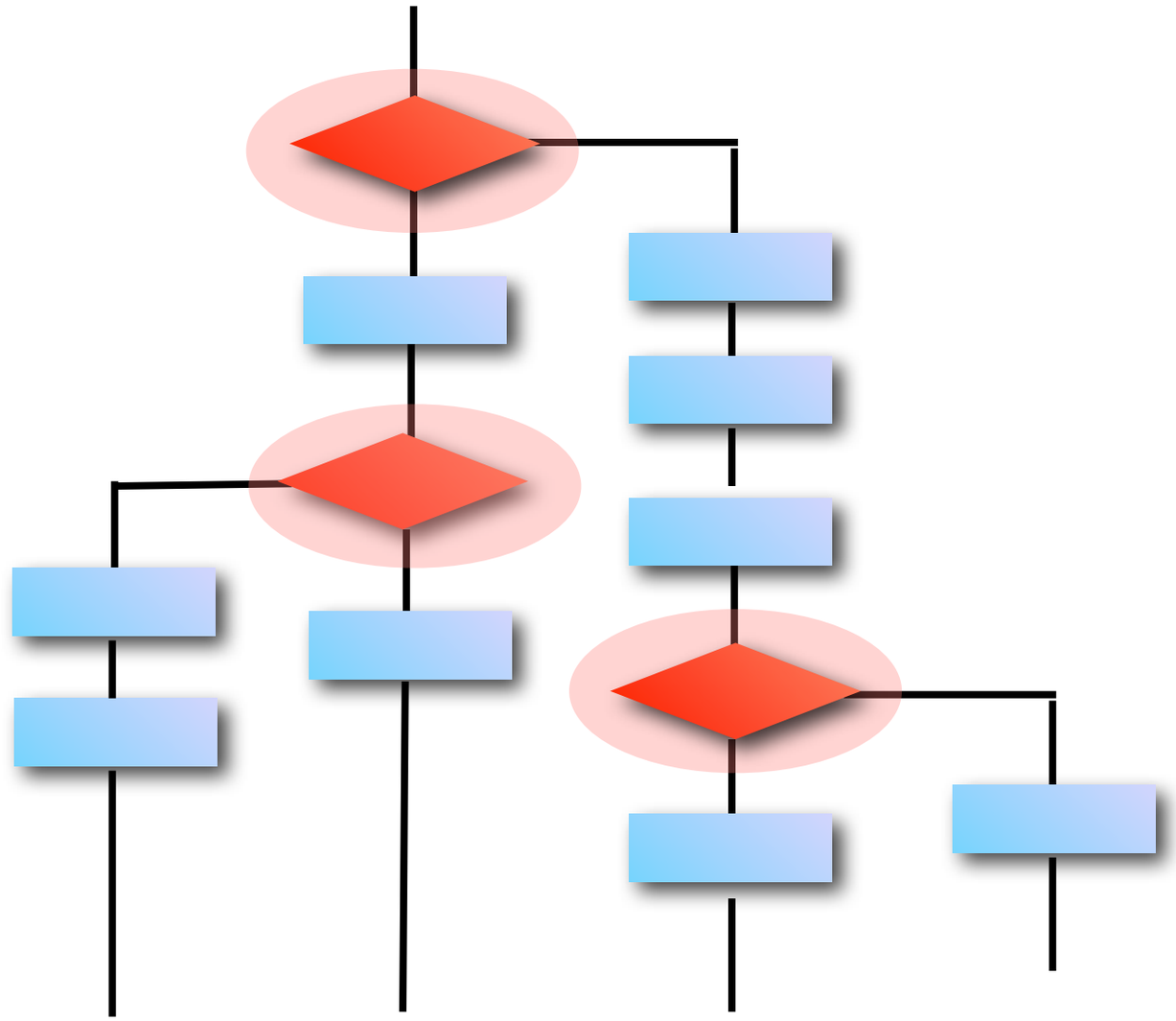


Test setup

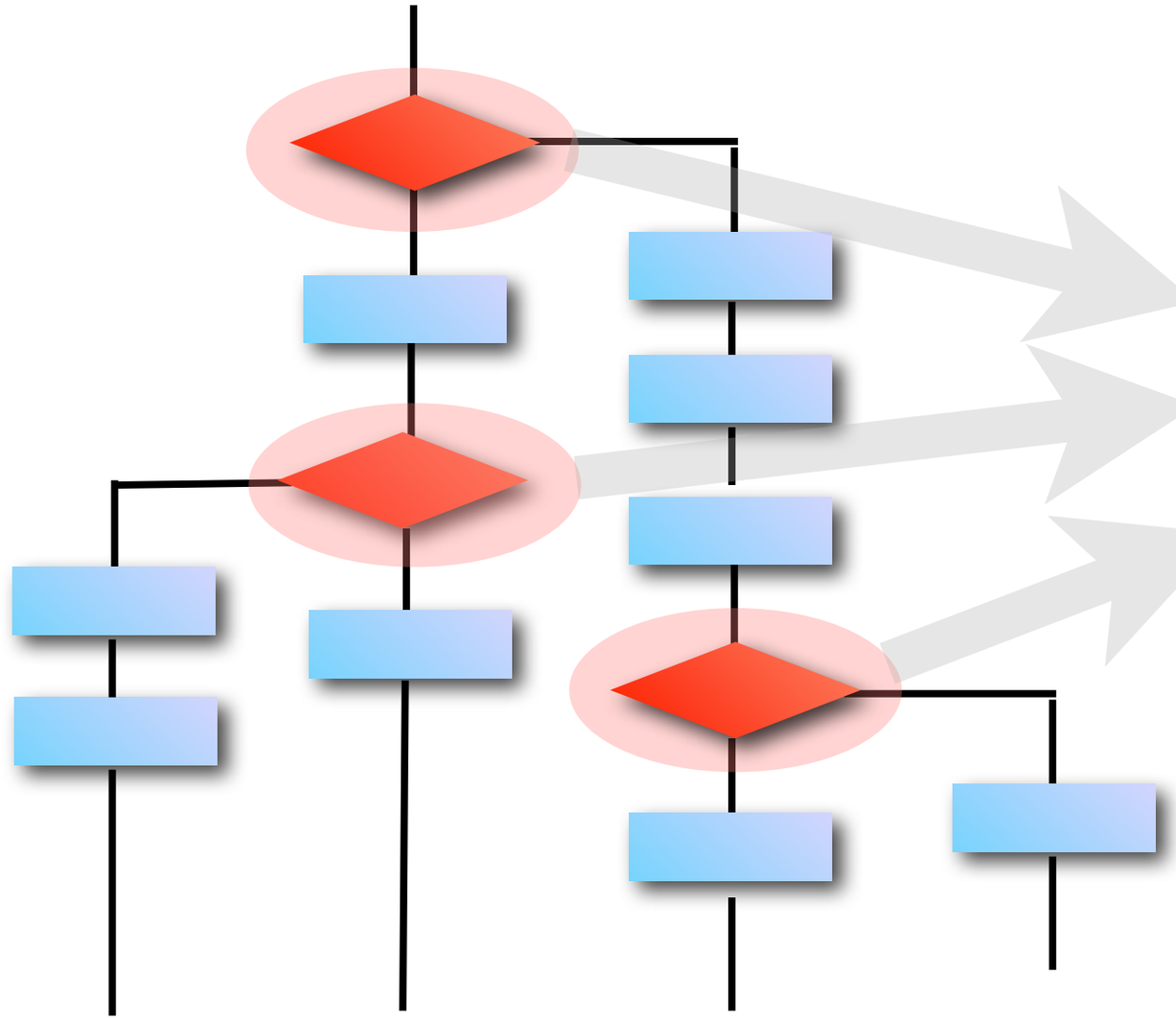
Search-Based Testing approach



Structural testing

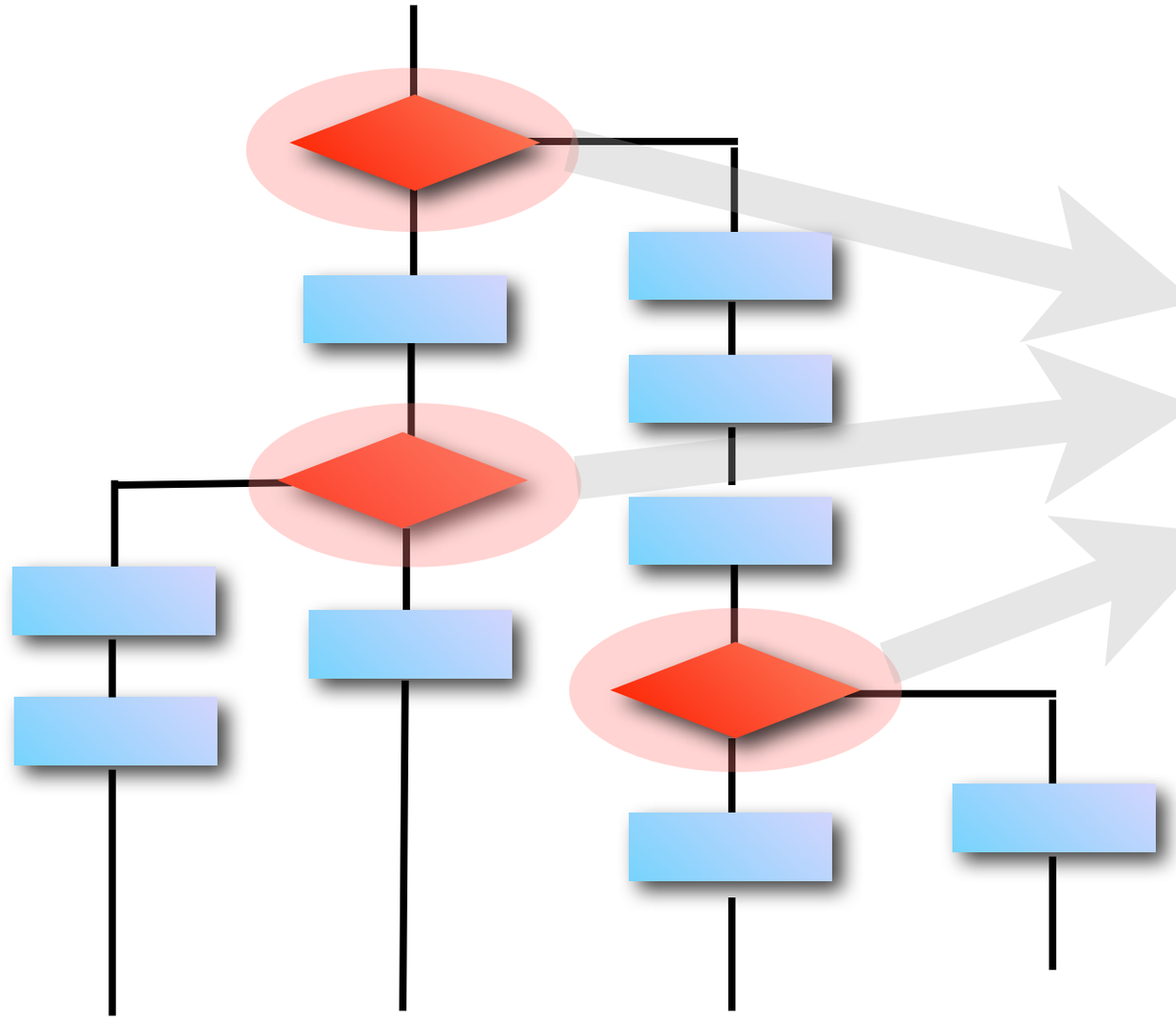


Structural testing



fitness function analyses the outcome of decision statements and the values of variables in predicates

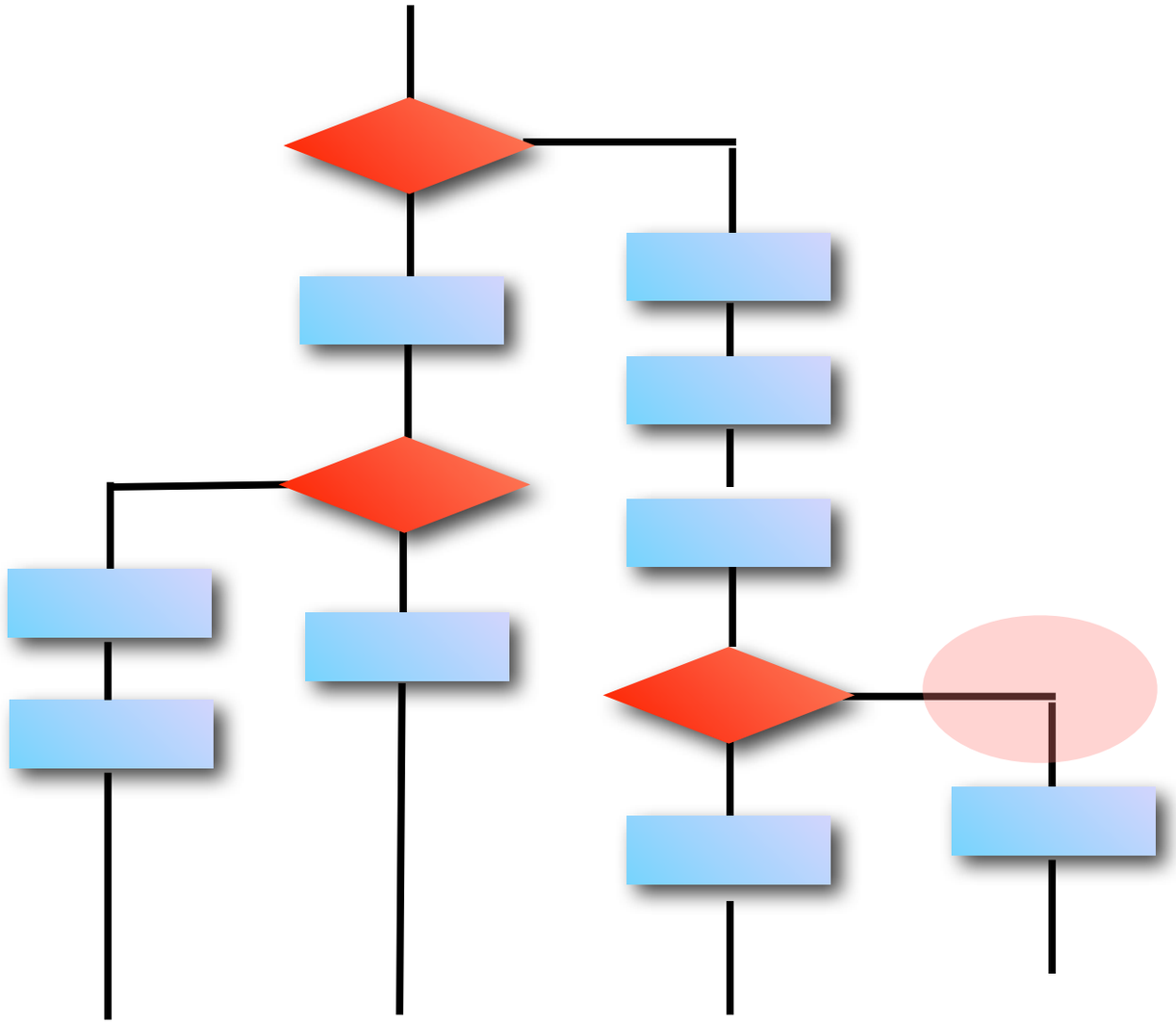
Structural testing



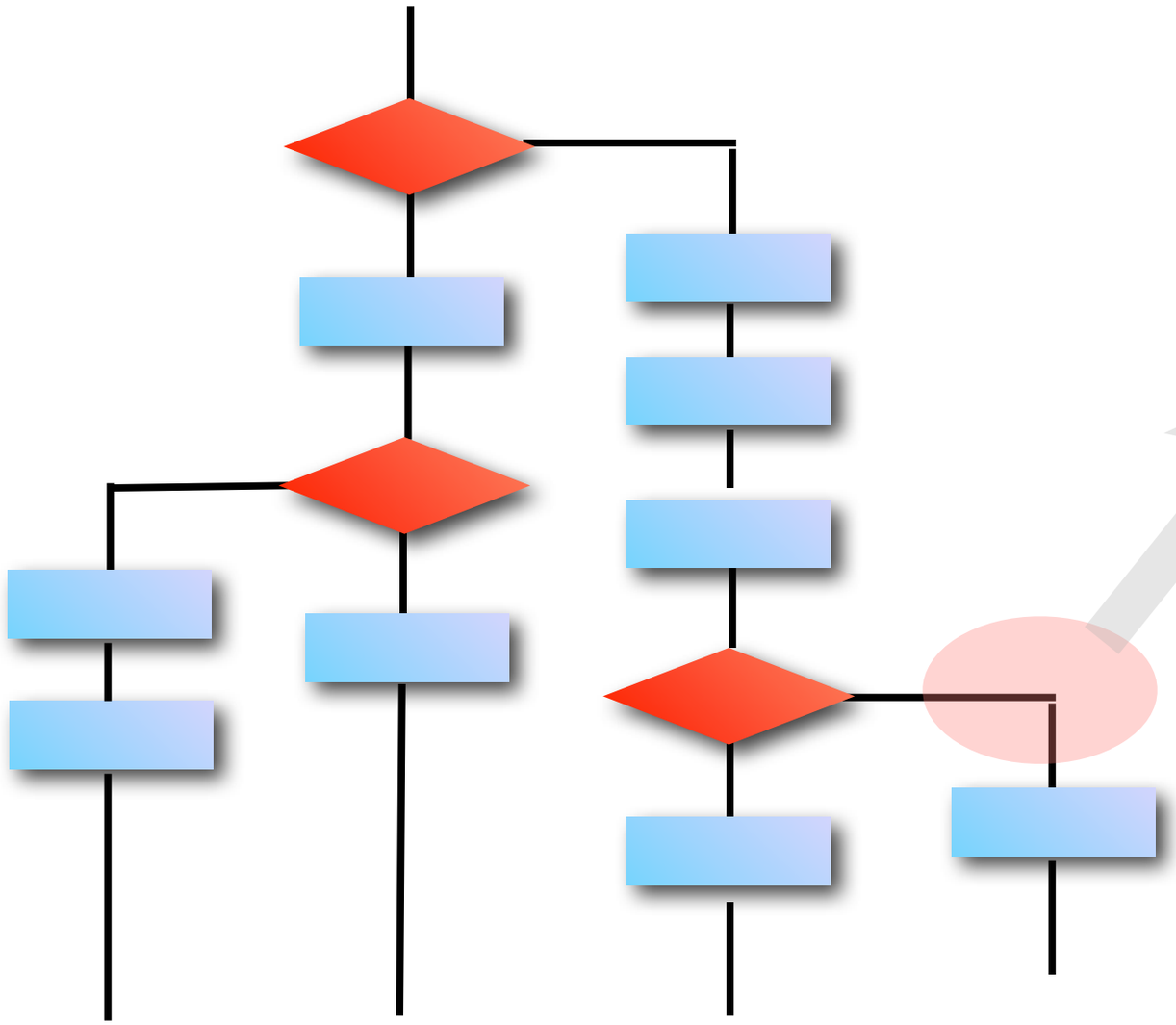
fitness function analyses the outcome of decision statements and the values of variables in predicates

More later ...

Assertion testing



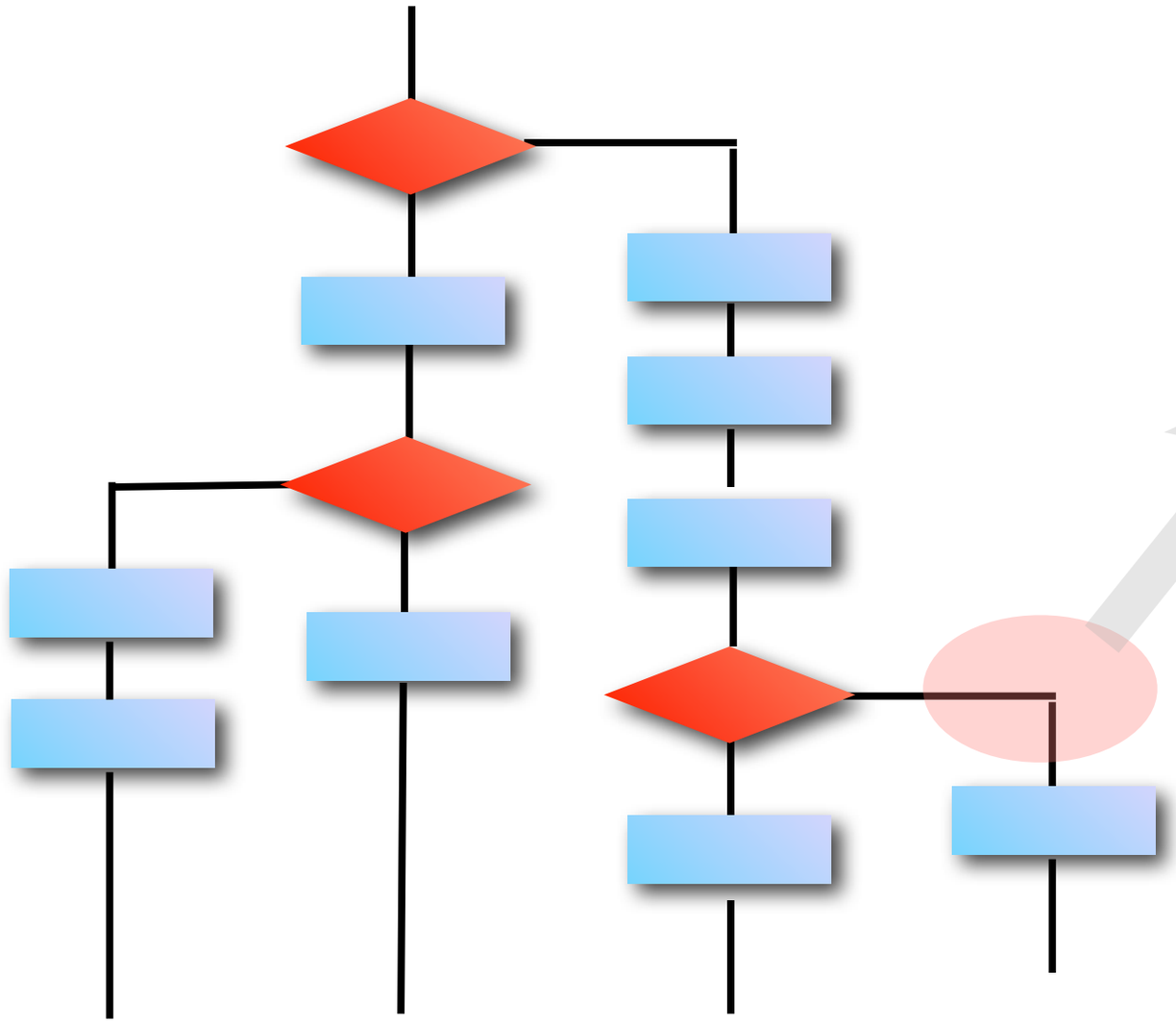
Assertion testing



assertion condition

speed < 150mph

Assertion testing



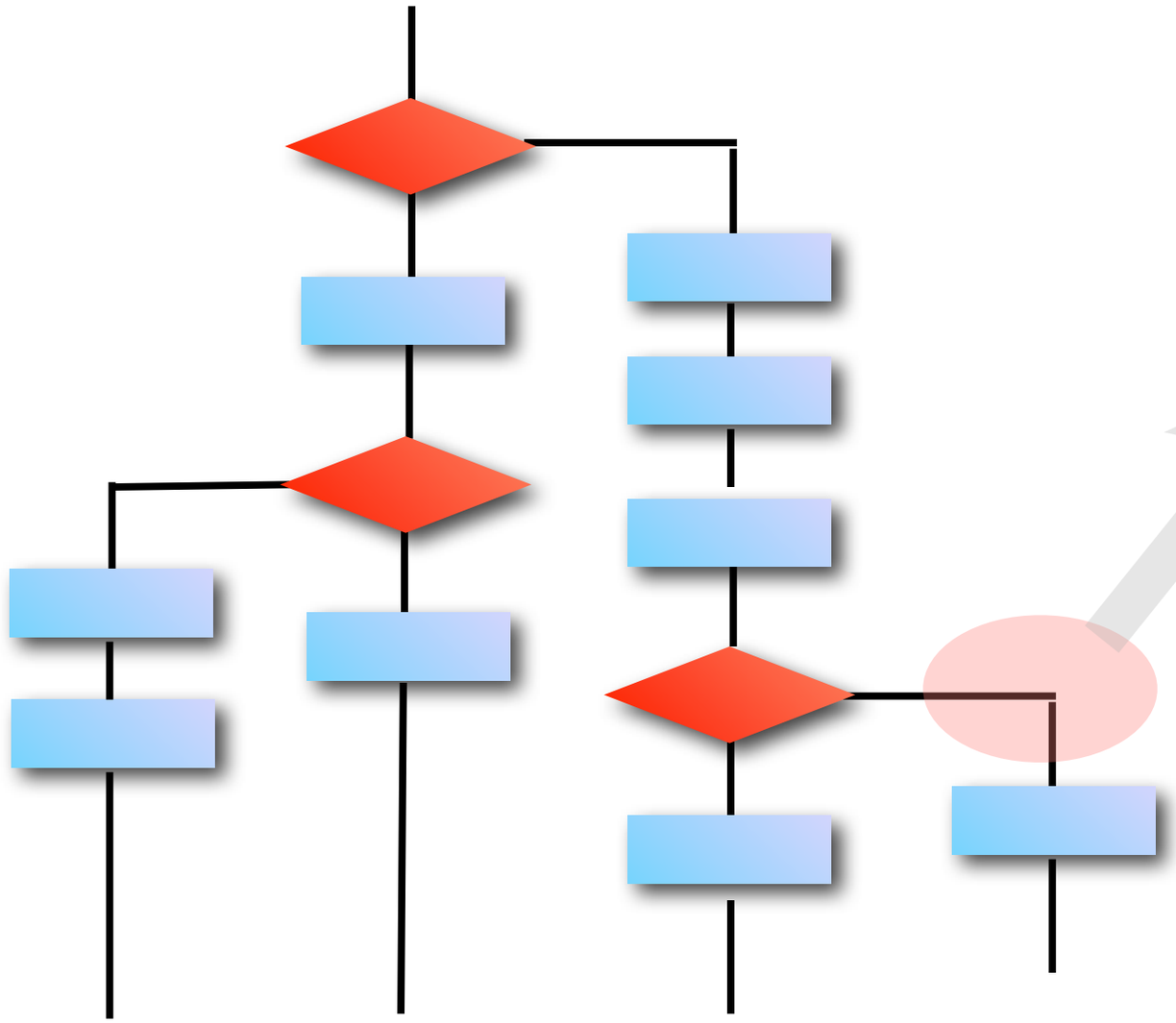
assertion condition

speed < 150mph

fitness function:

f = 150 - speed

Assertion testing



assertion condition

speed < 150mph

fitness function:

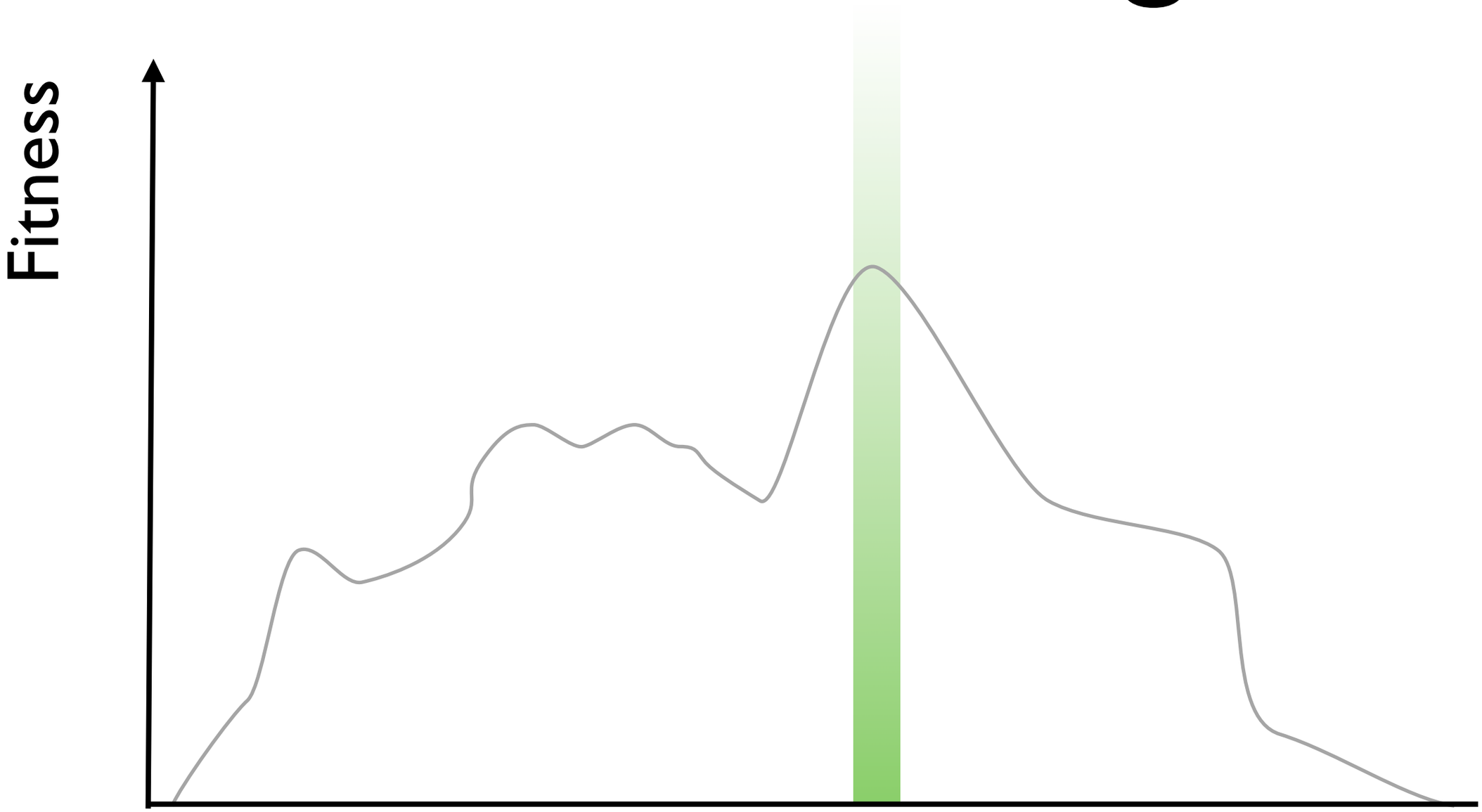
f =150 - speed

fitness minimised

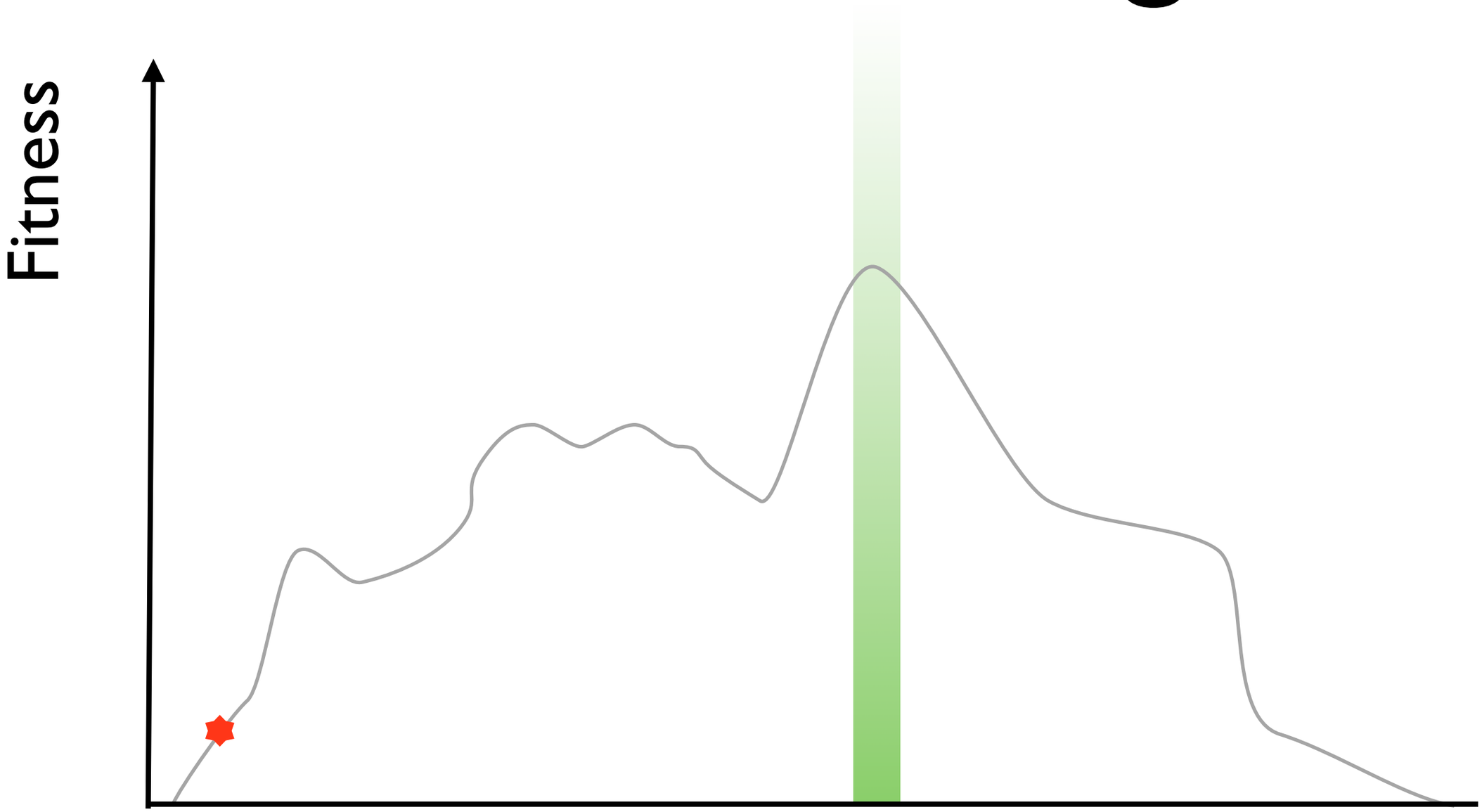
If *f* is zero or less a fault is found

Search Techniques

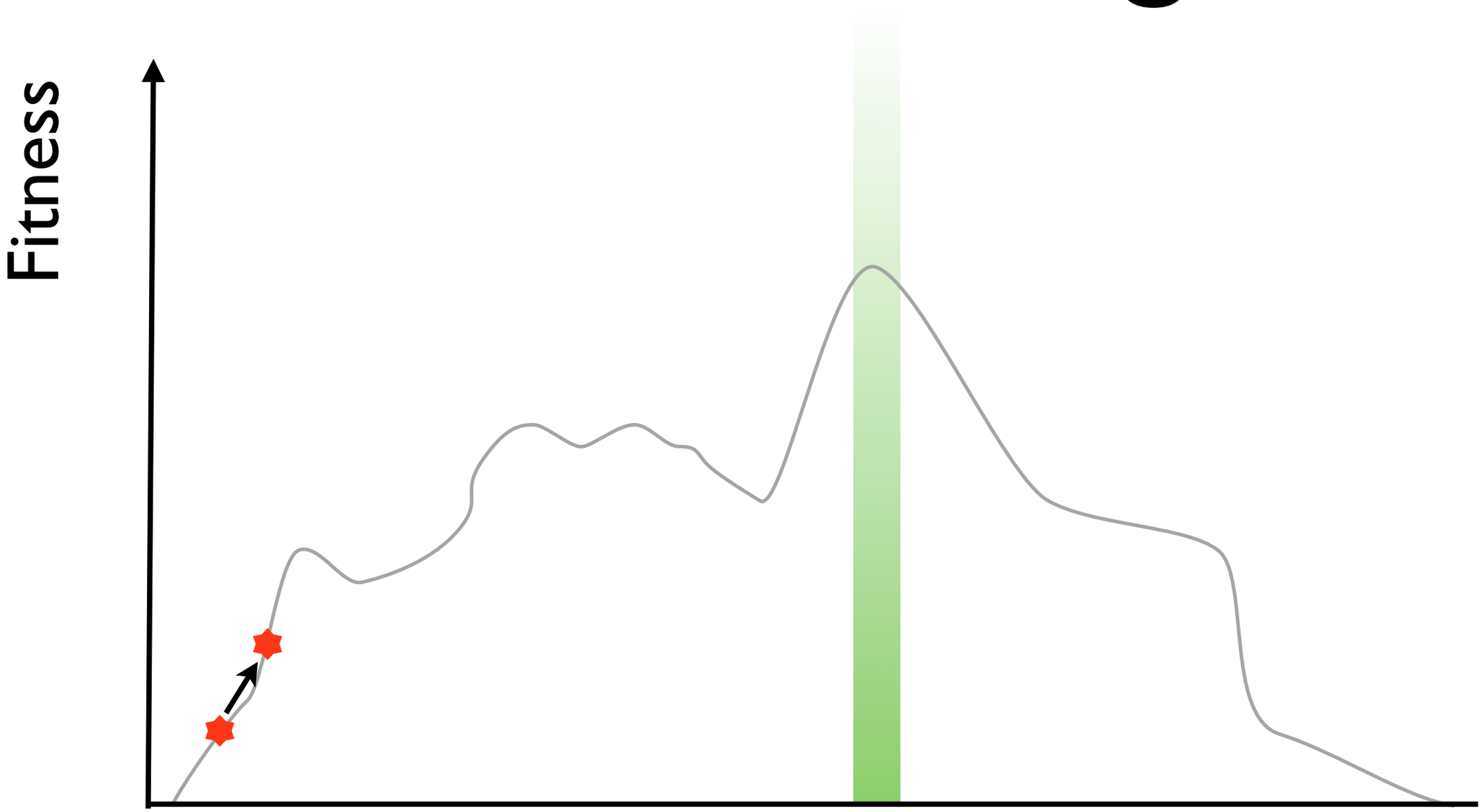
Hill Climbing



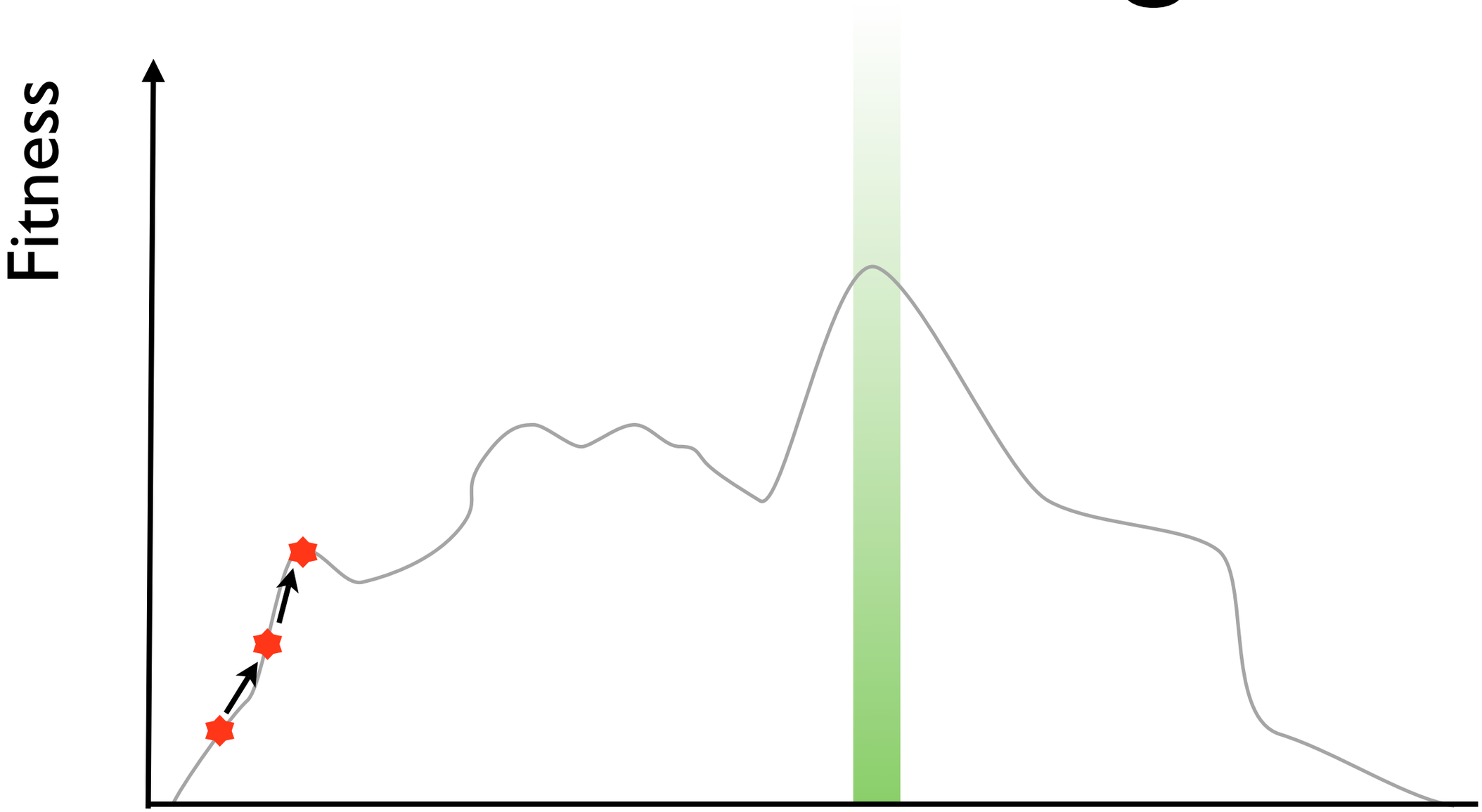
Hill Climbing



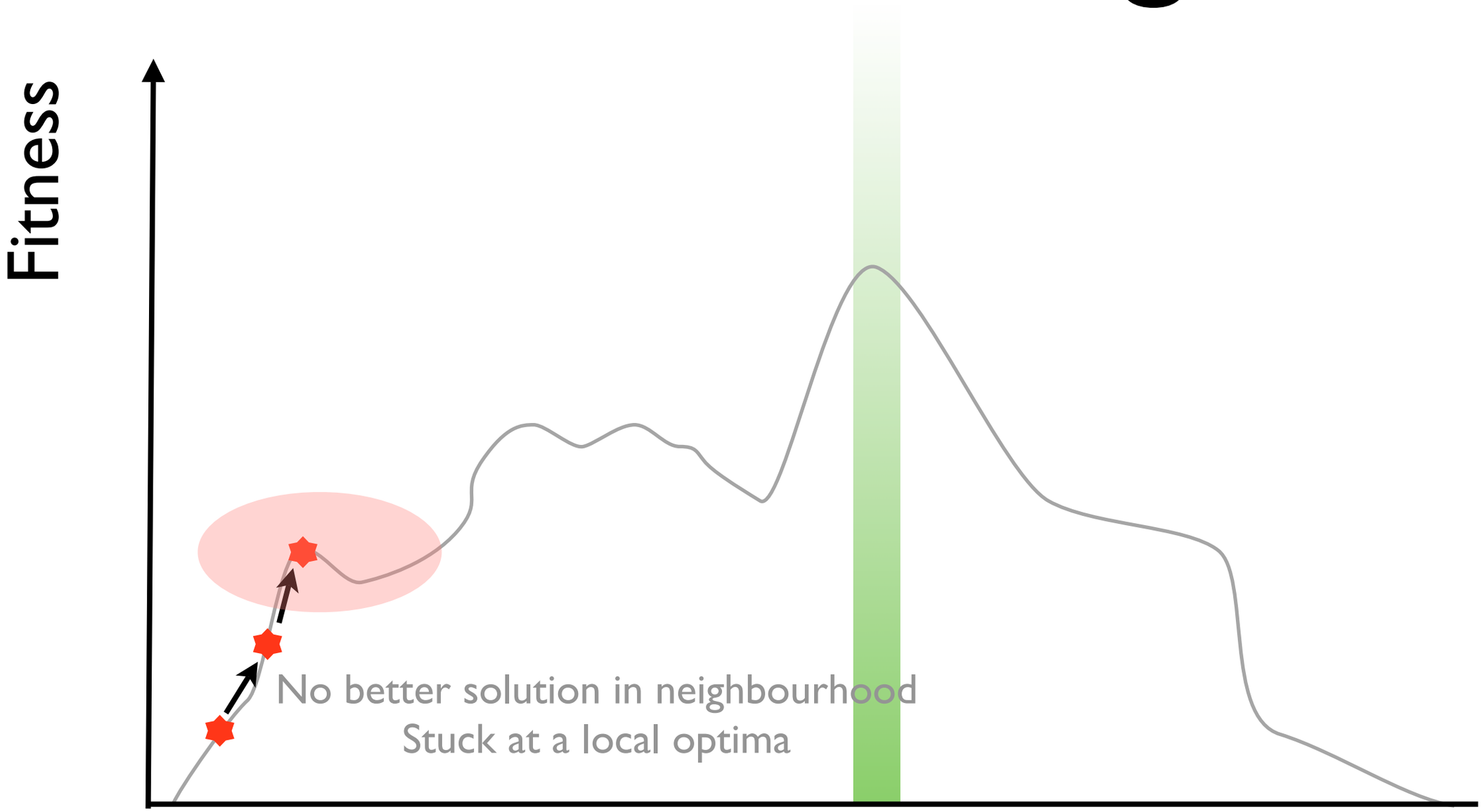
Hill Climbing



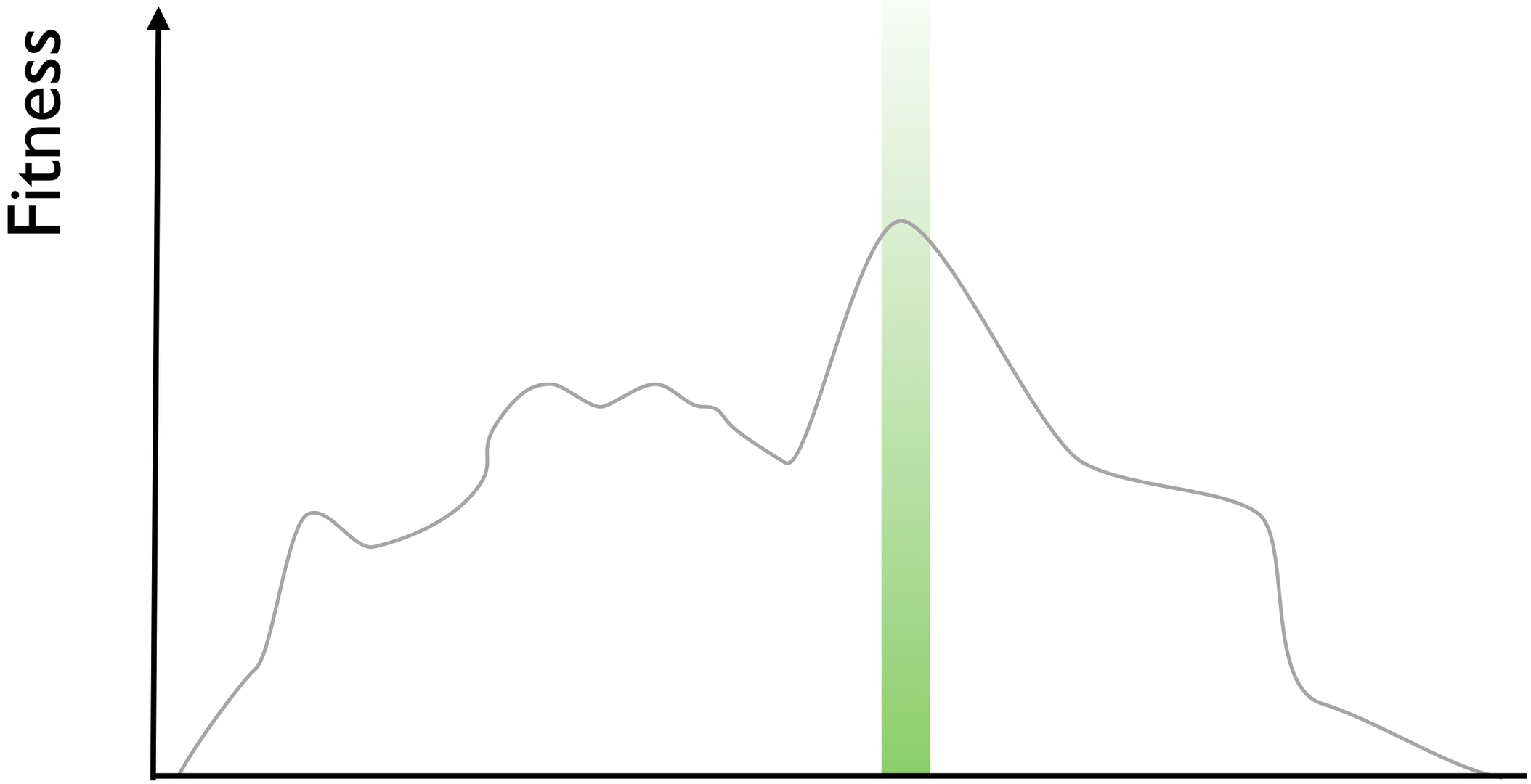
Hill Climbing



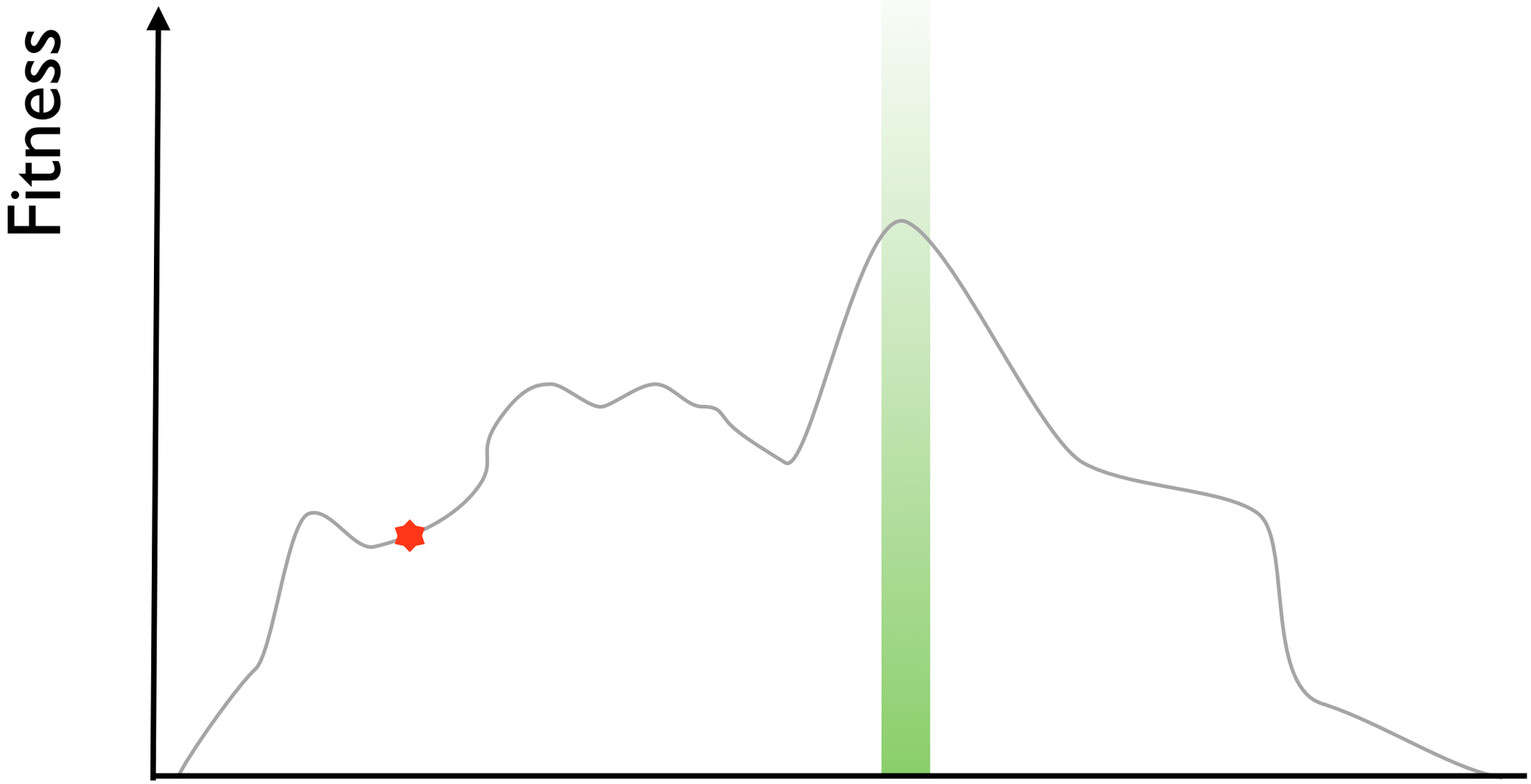
Hill Climbing



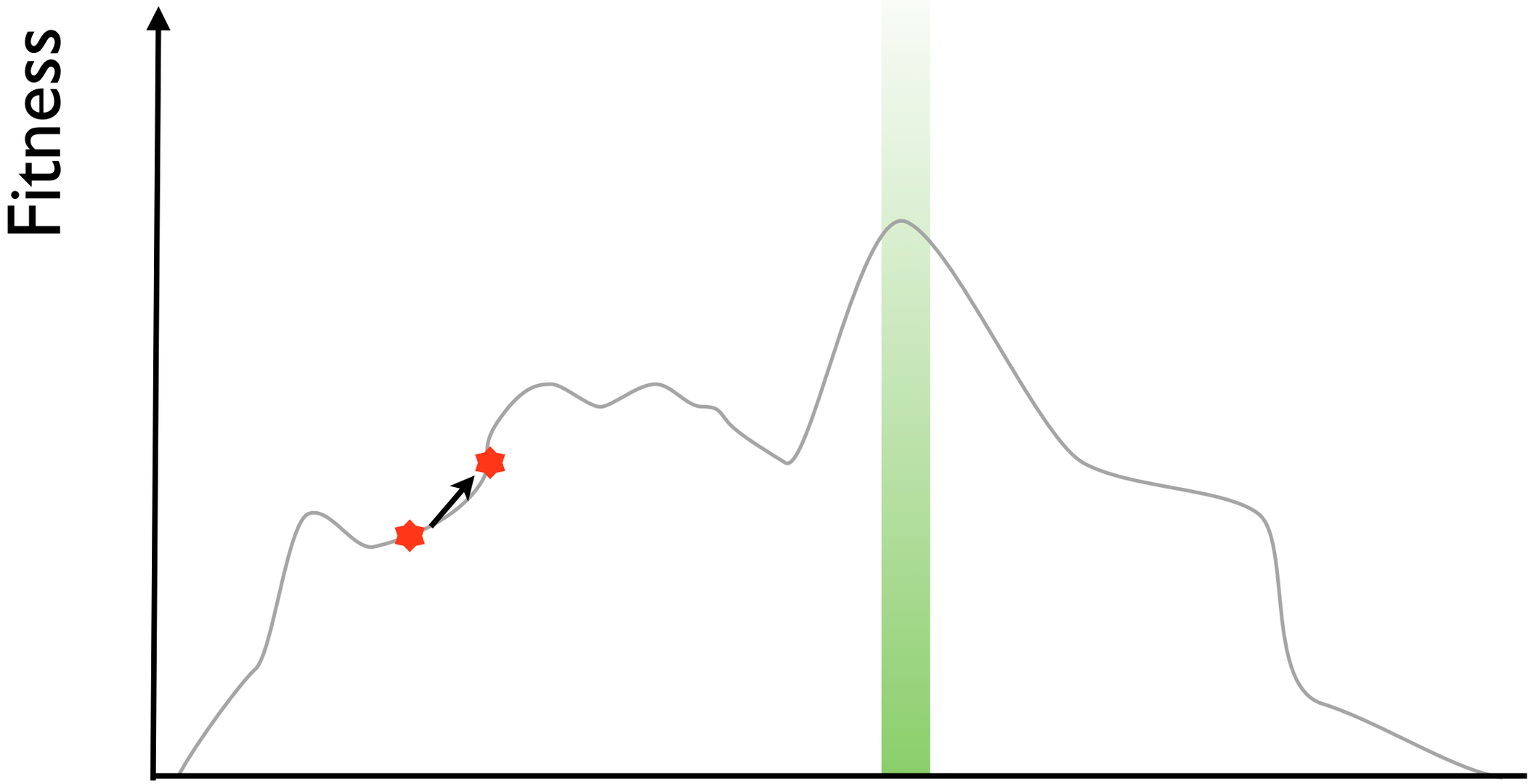
Hill Climbing - Restarts



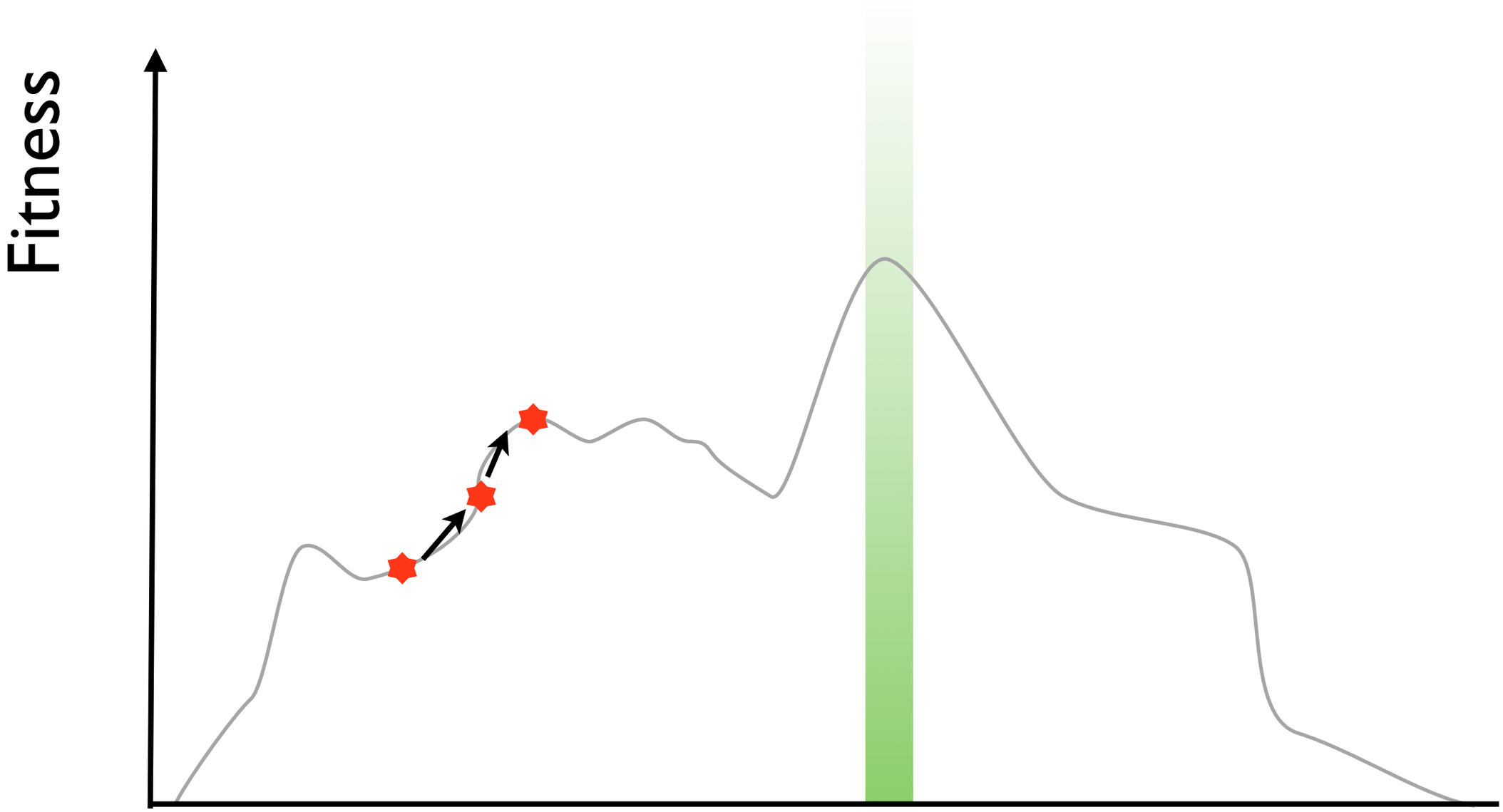
Hill Climbing - Restarts



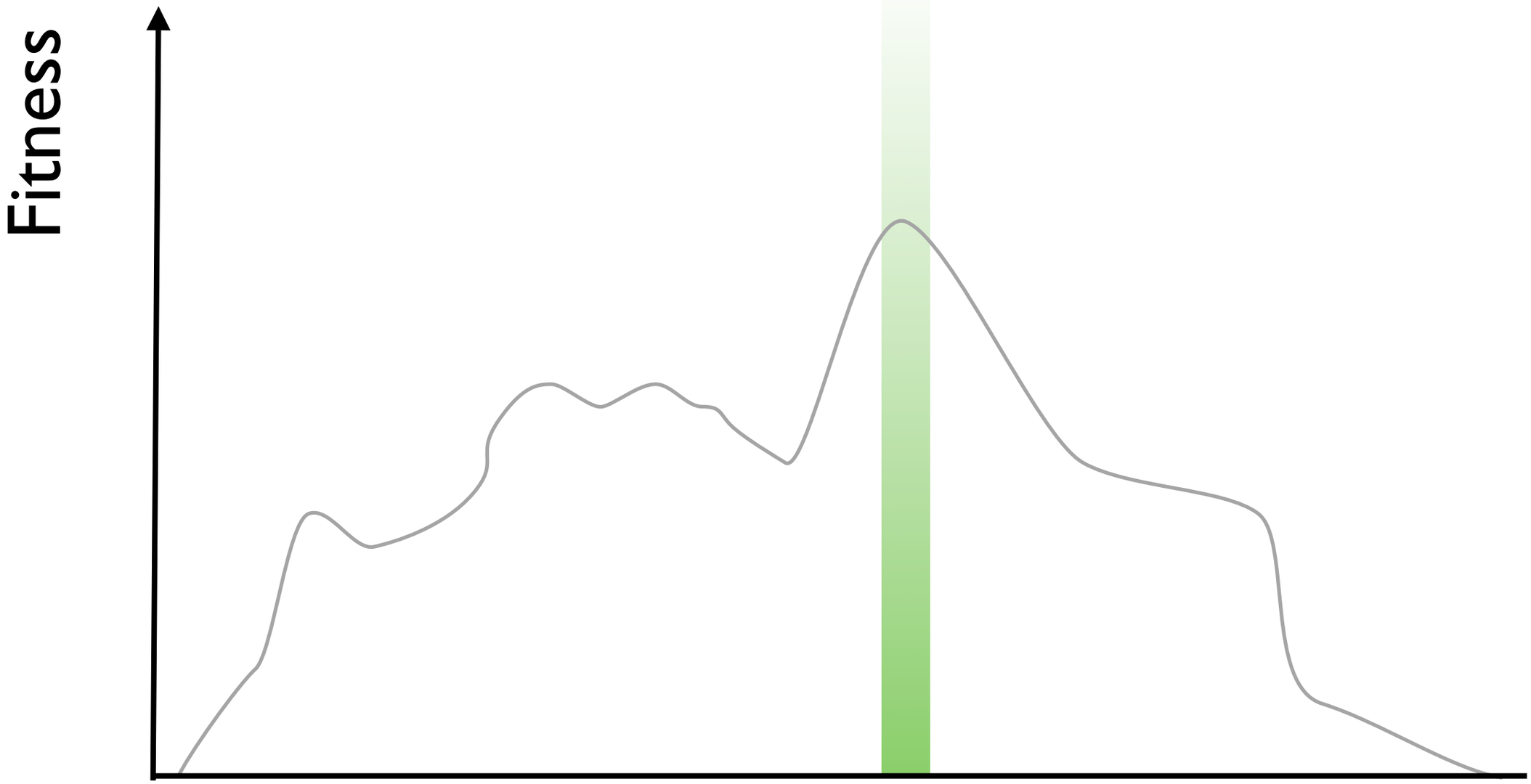
Hill Climbing - Restarts



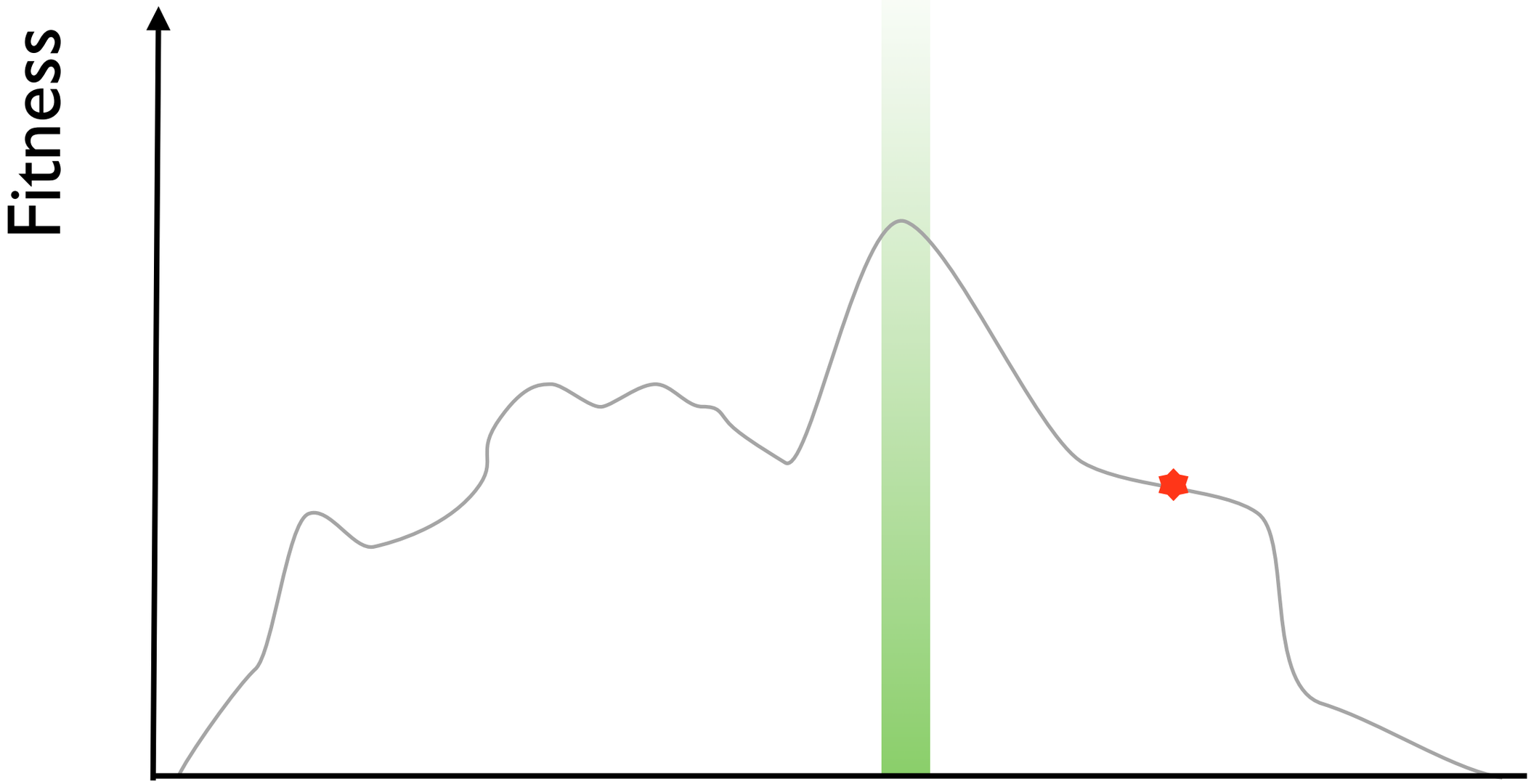
Hill Climbing - Restarts



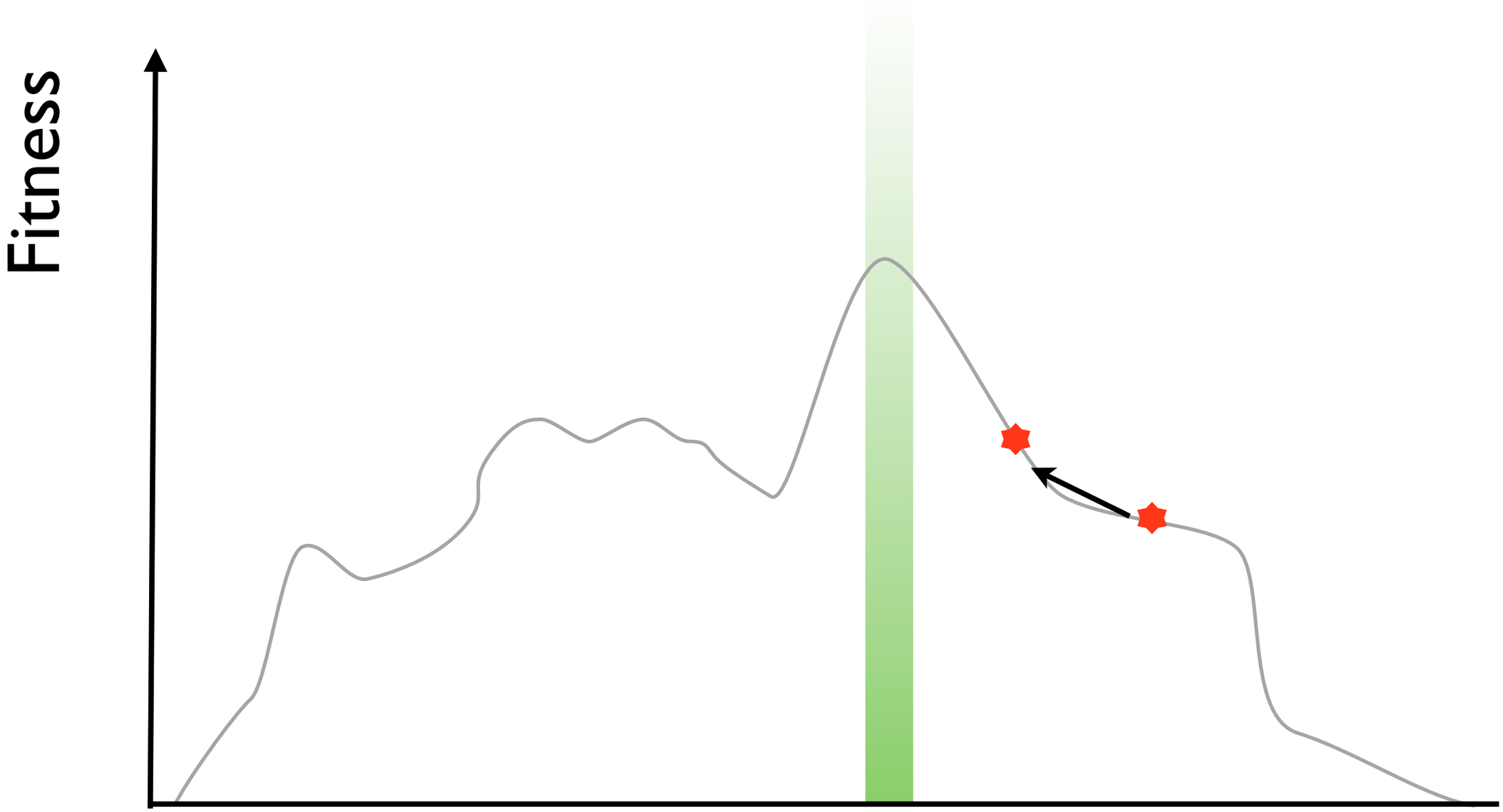
Hill Climbing - Restarts



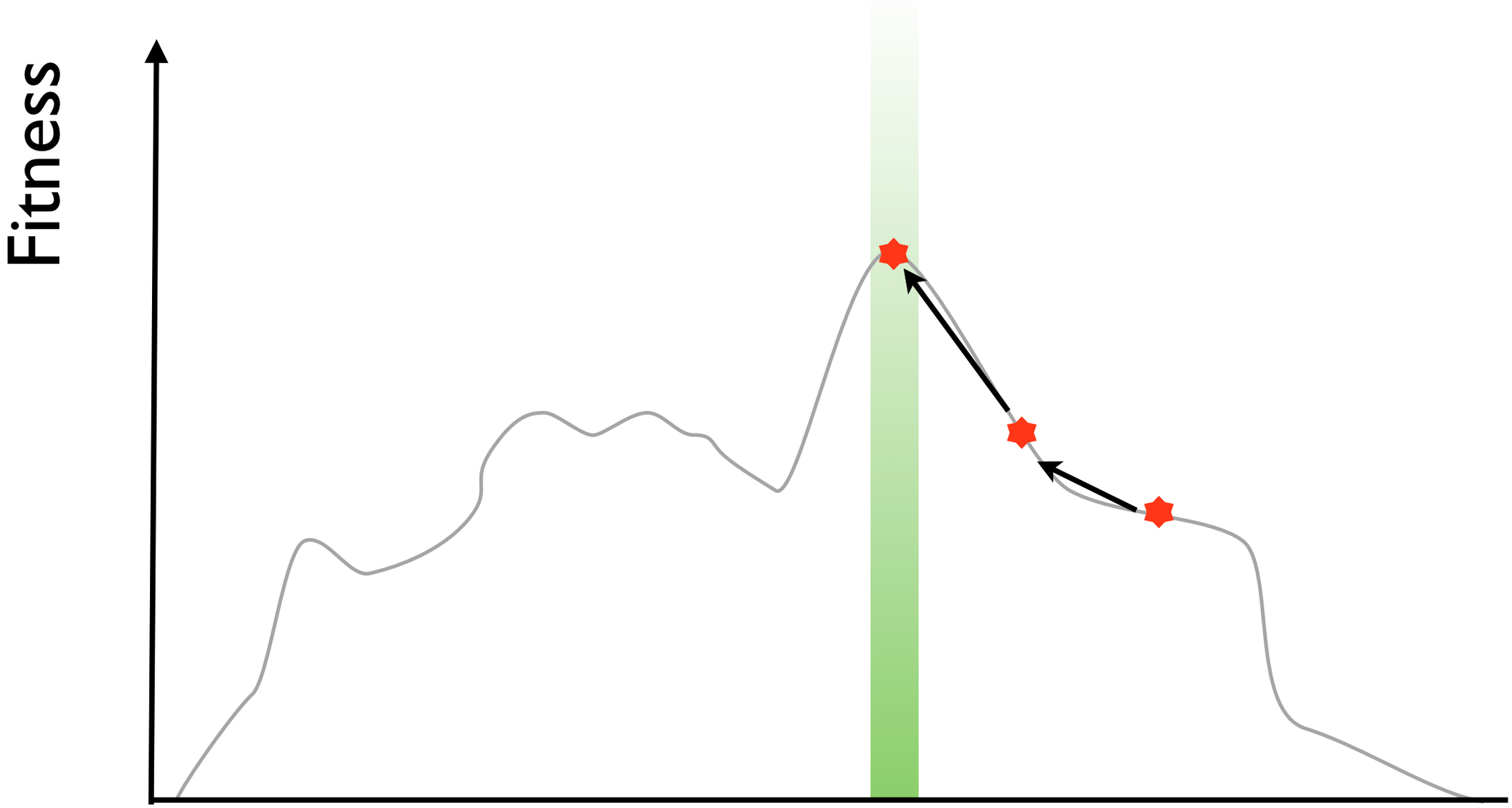
Hill Climbing - Restarts



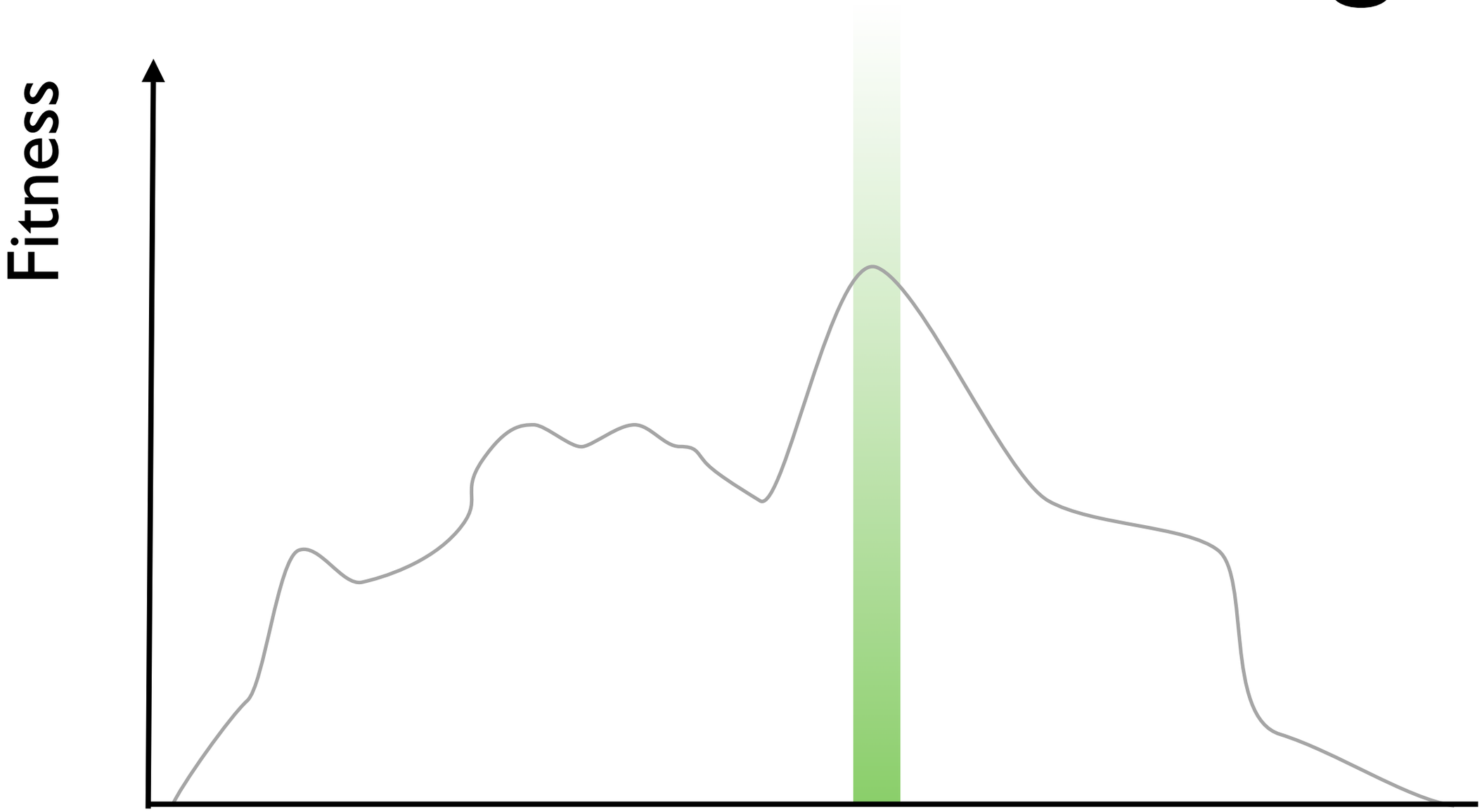
Hill Climbing - Restarts



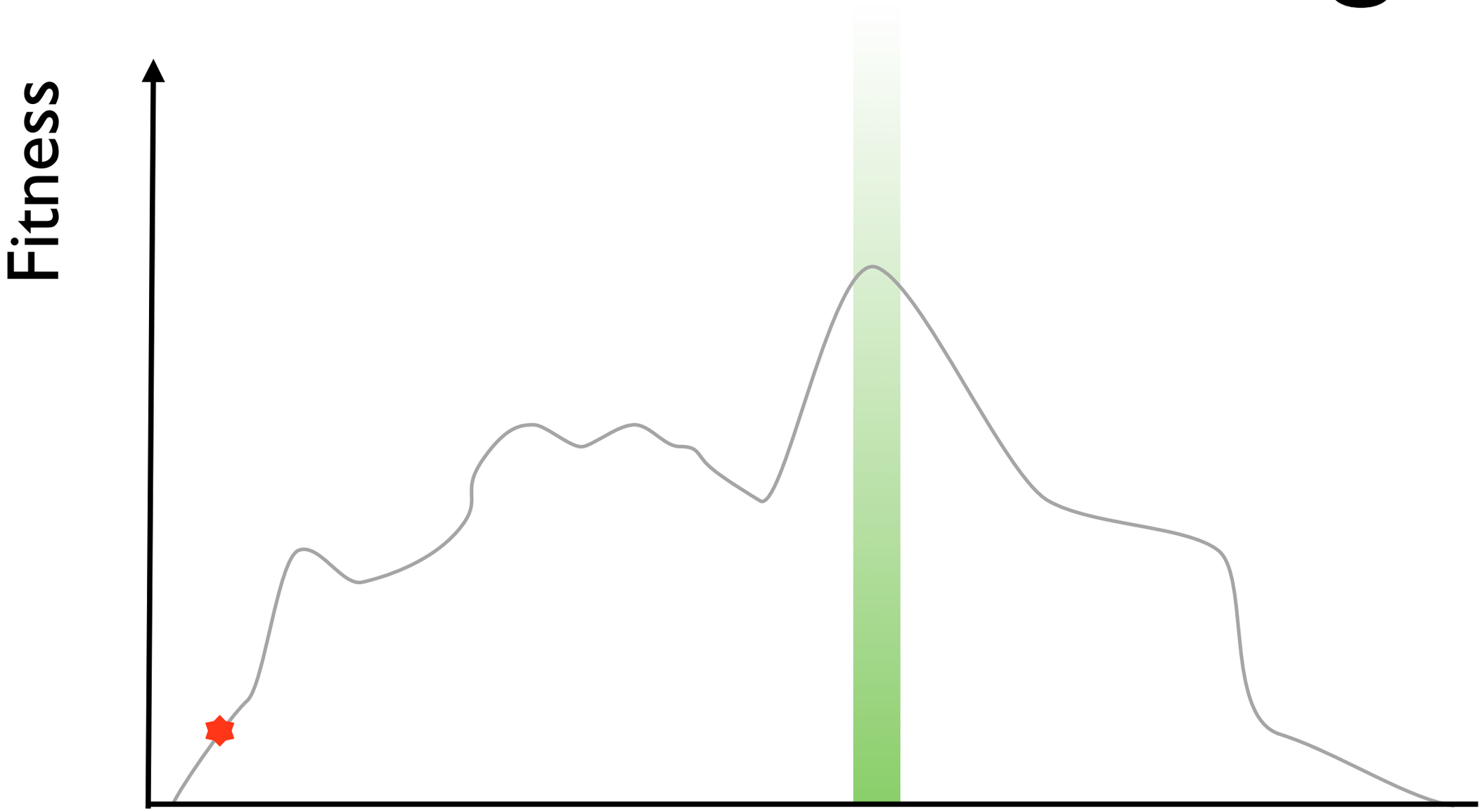
Hill Climbing - Restarts



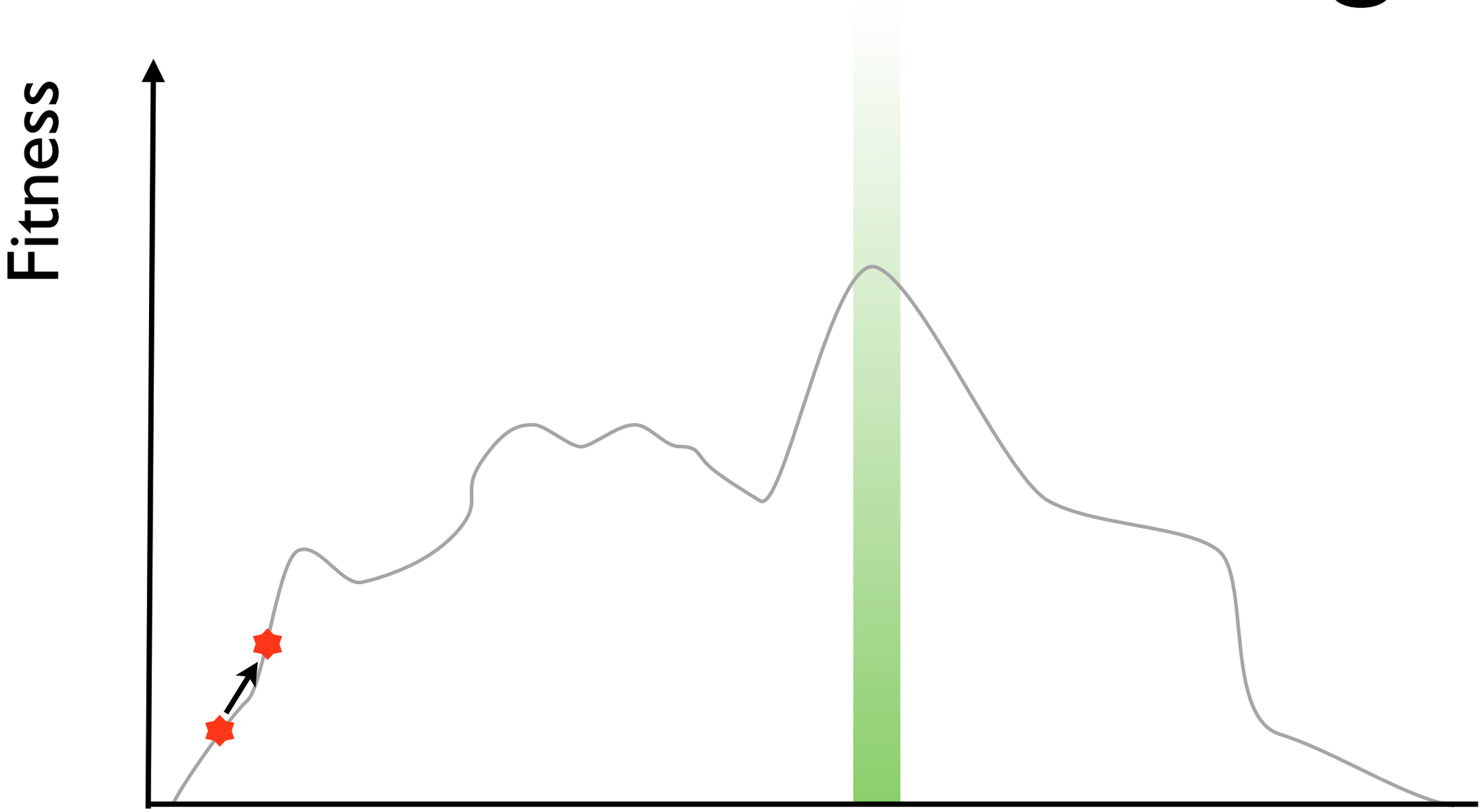
Simulated Annealing



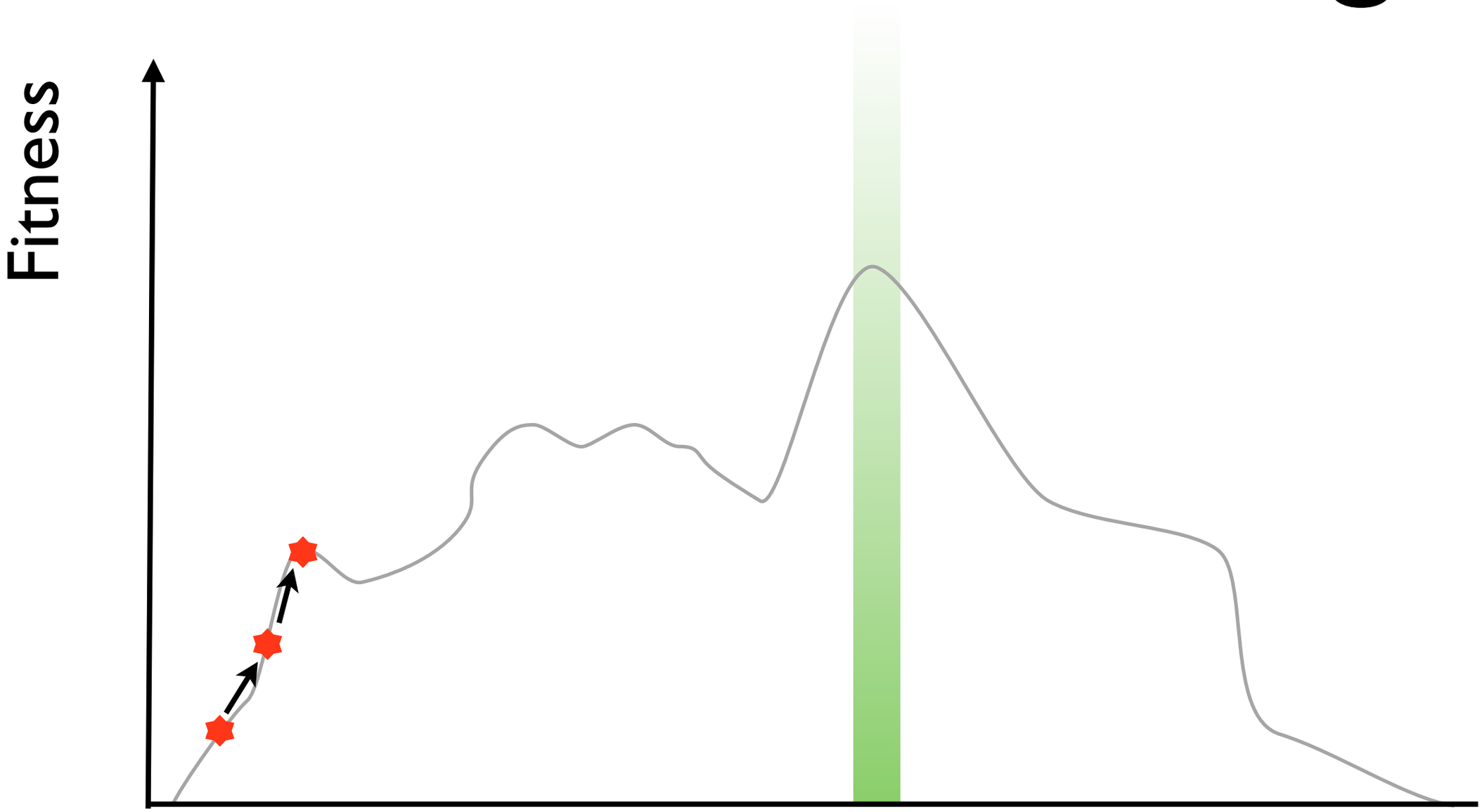
Simulated Annealing



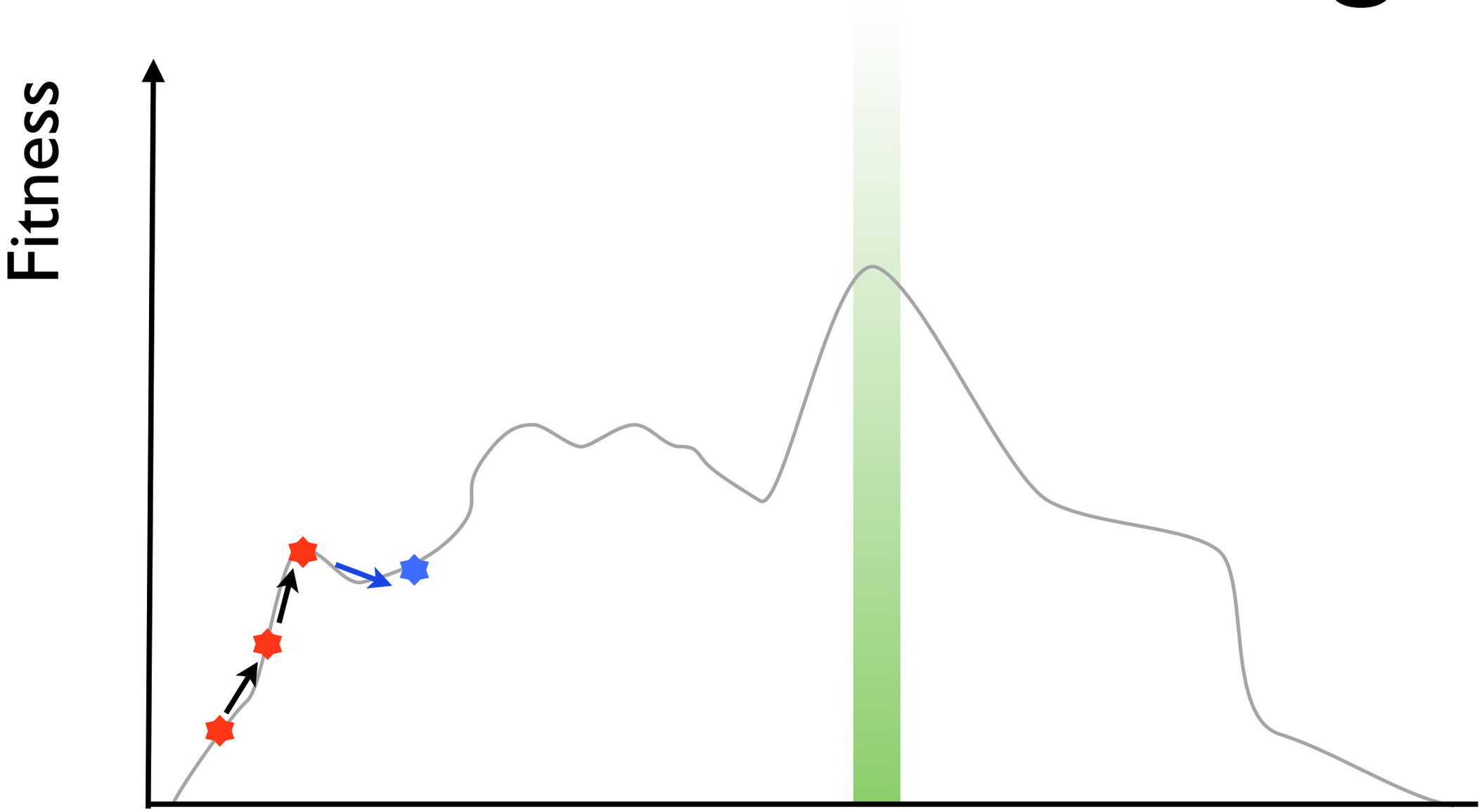
Simulated Annealing



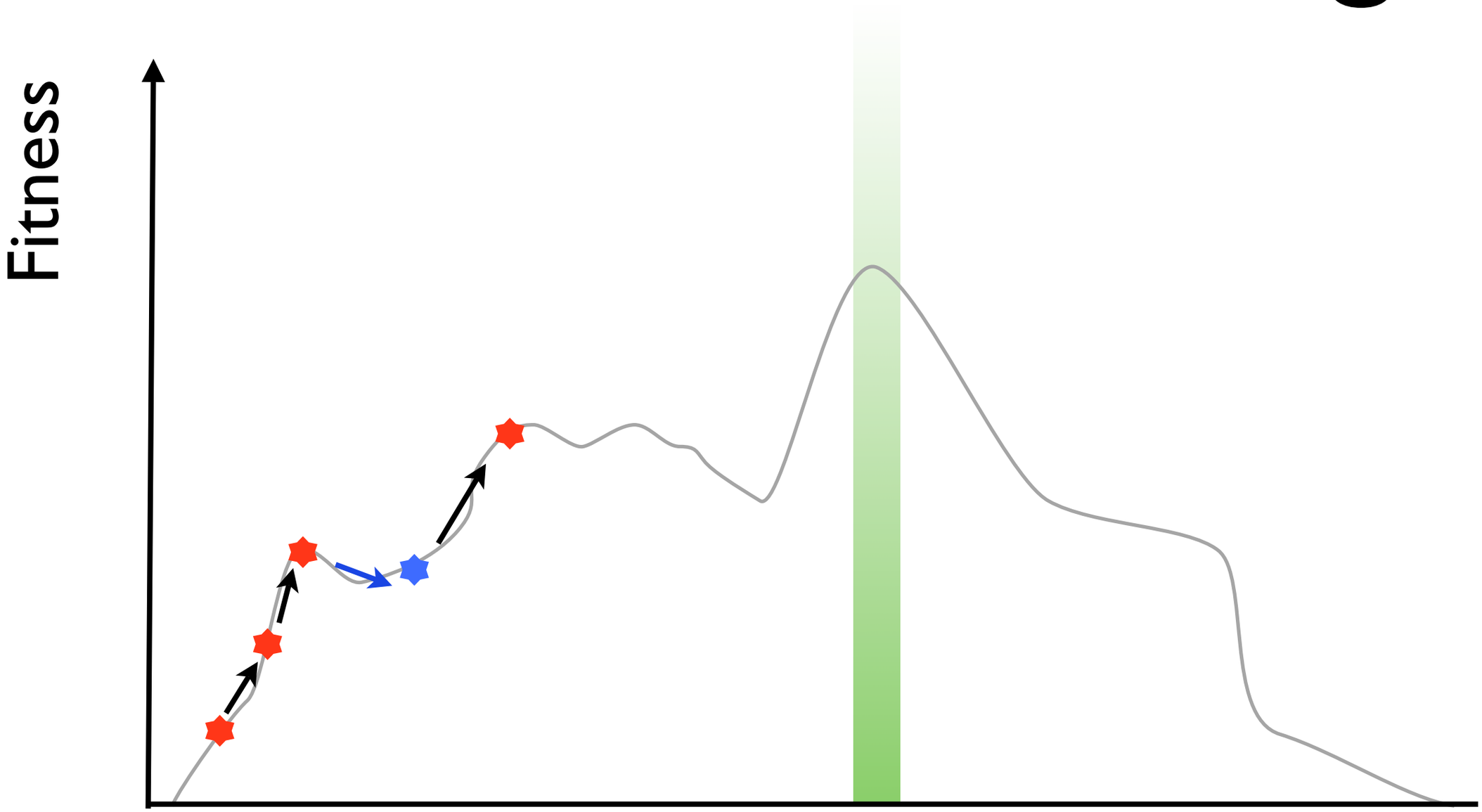
Simulated Annealing



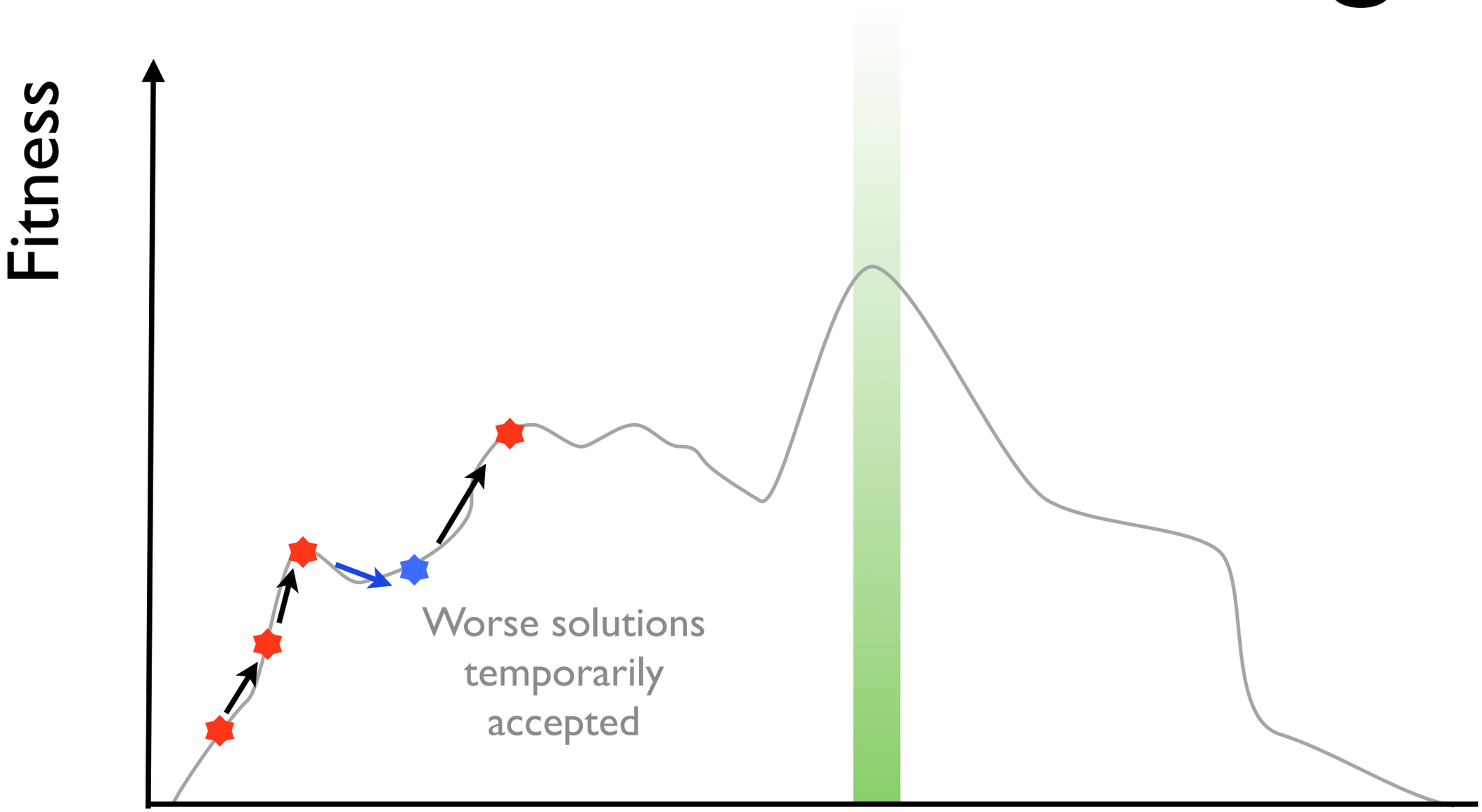
Simulated Annealing



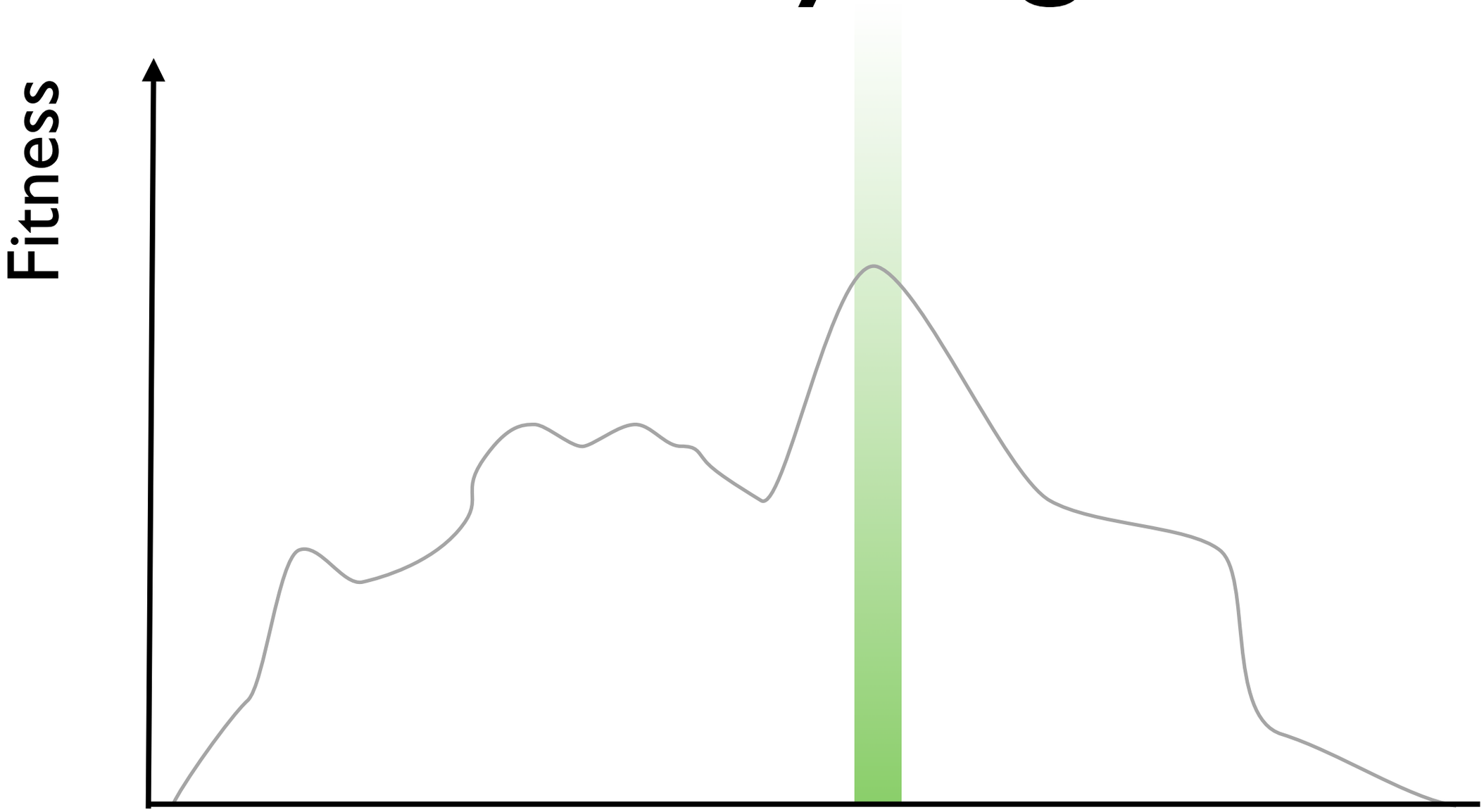
Simulated Annealing



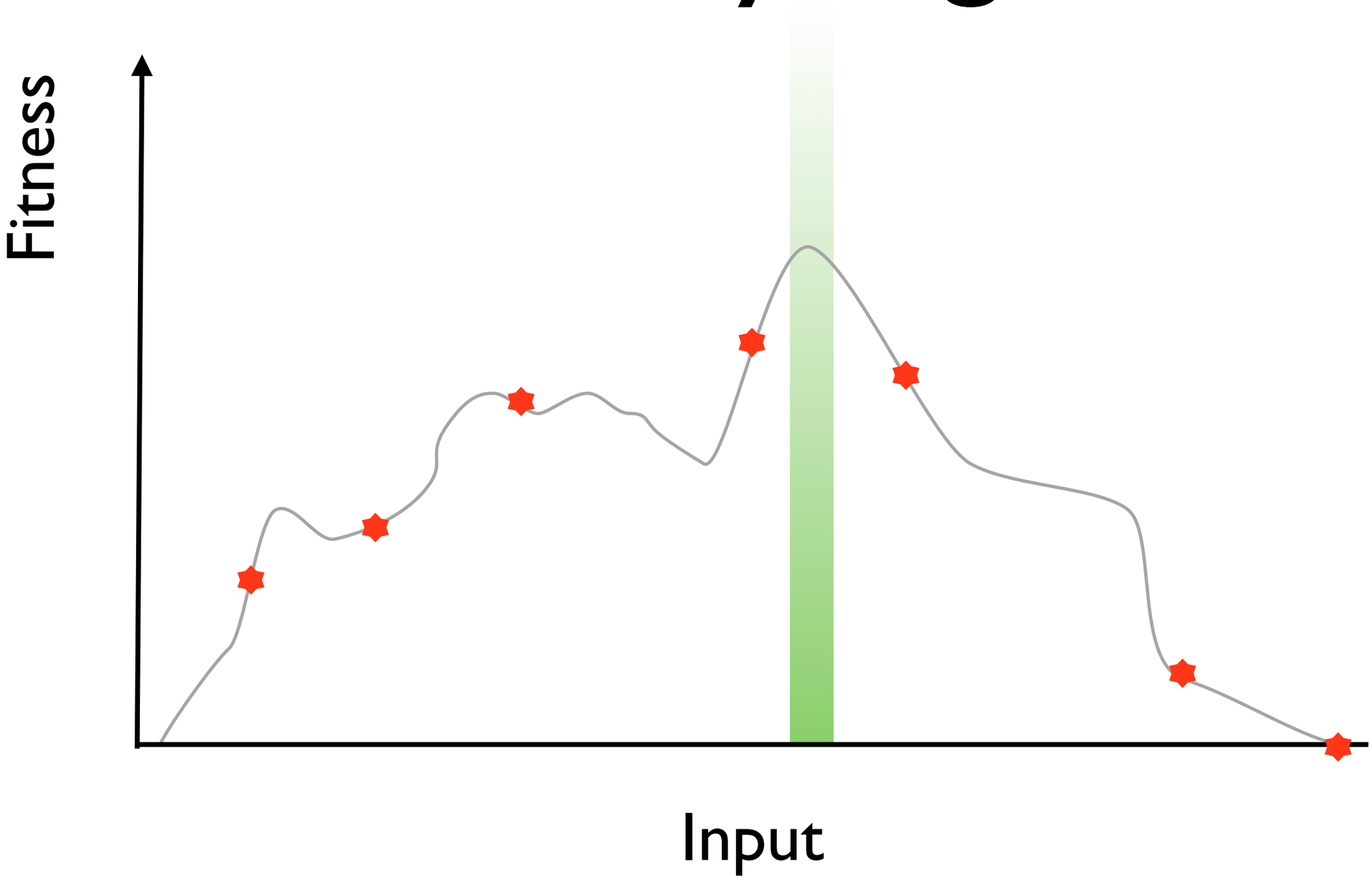
Simulated Annealing



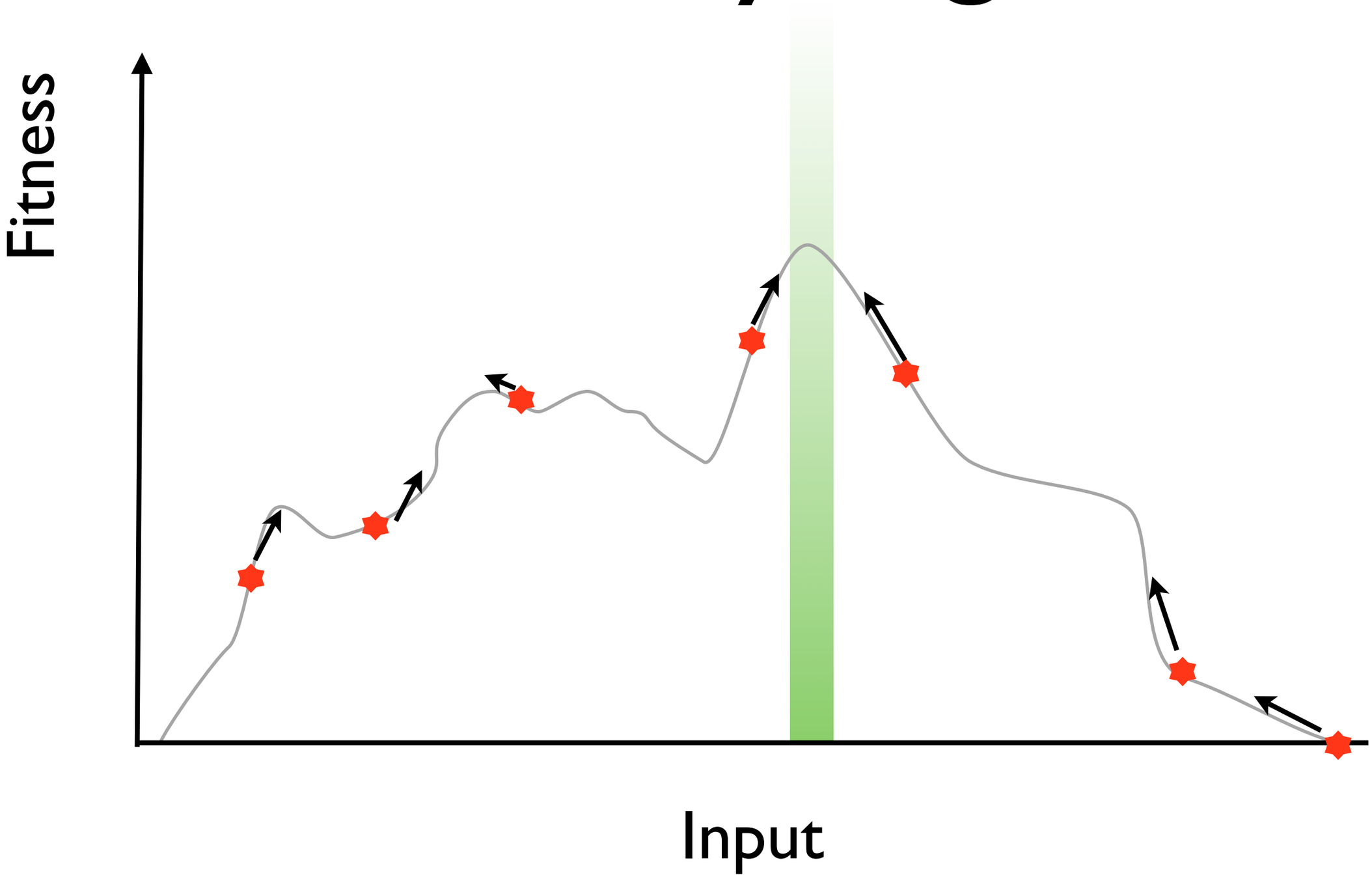
Evolutionary Algorithm



Evolutionary Algorithm



Evolutionary Algorithm



Evolutionary Algorithms

inspired by Darwinian Evolution and concept of survival of the fittest

Crossover

Mutation

Crossover

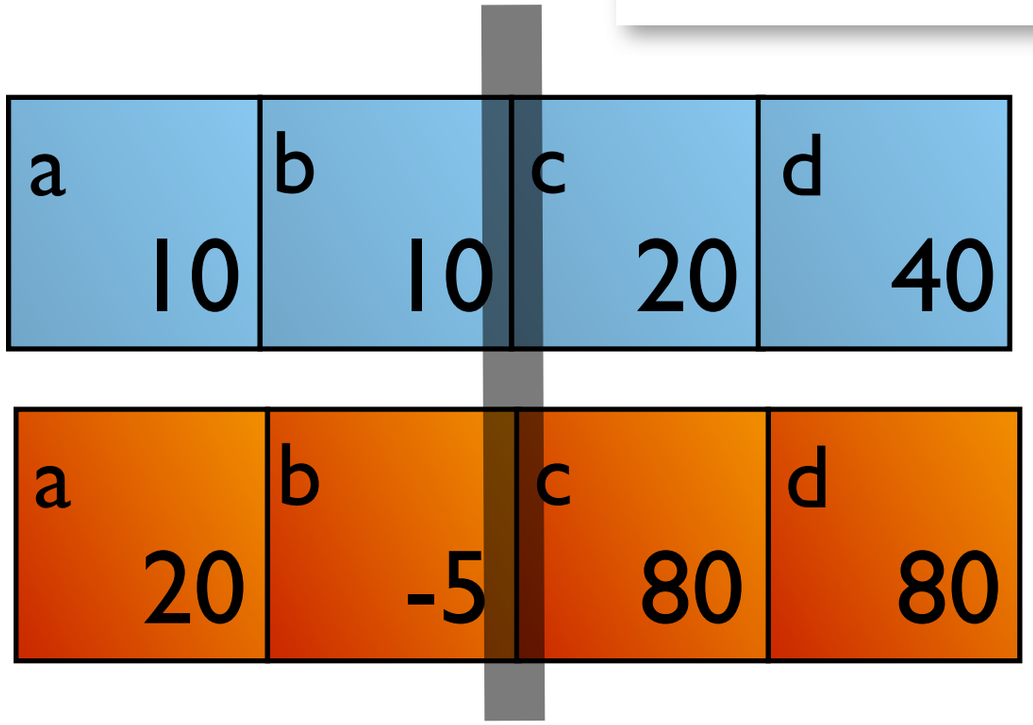
```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```

a	b	c	d
10	10	20	40

a	b	c	d
20	-5	80	80

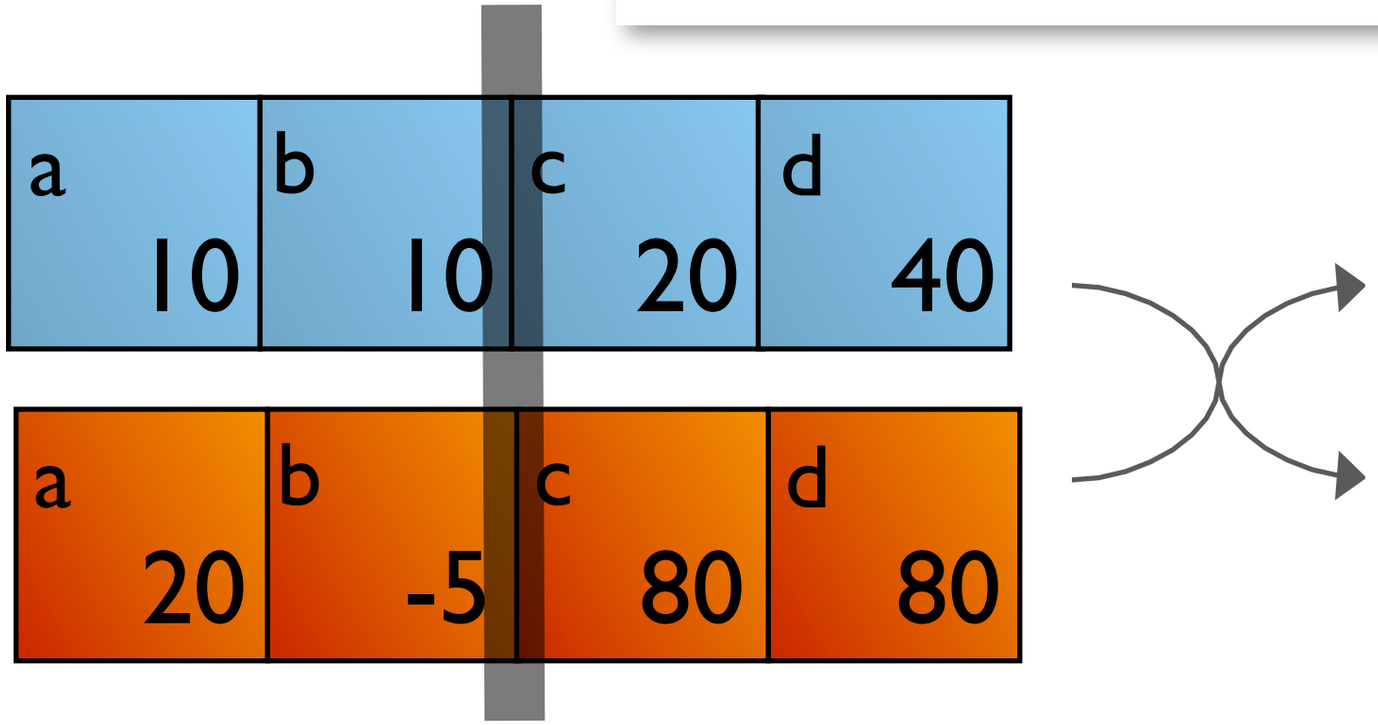
Crossover

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```



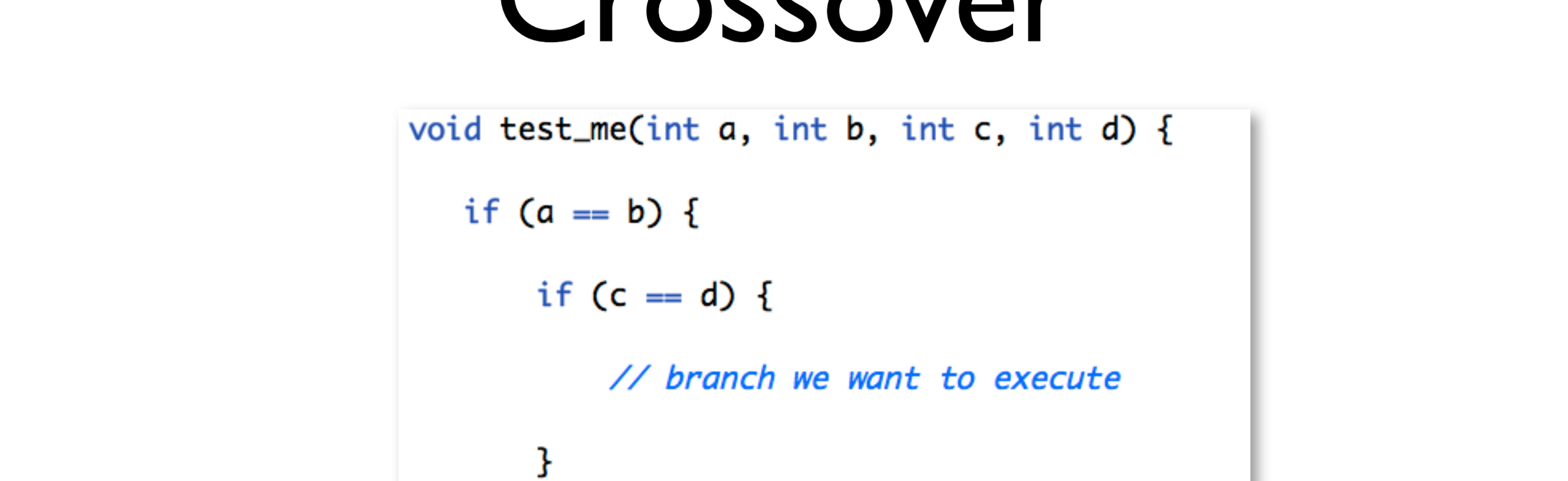
Crossover

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```



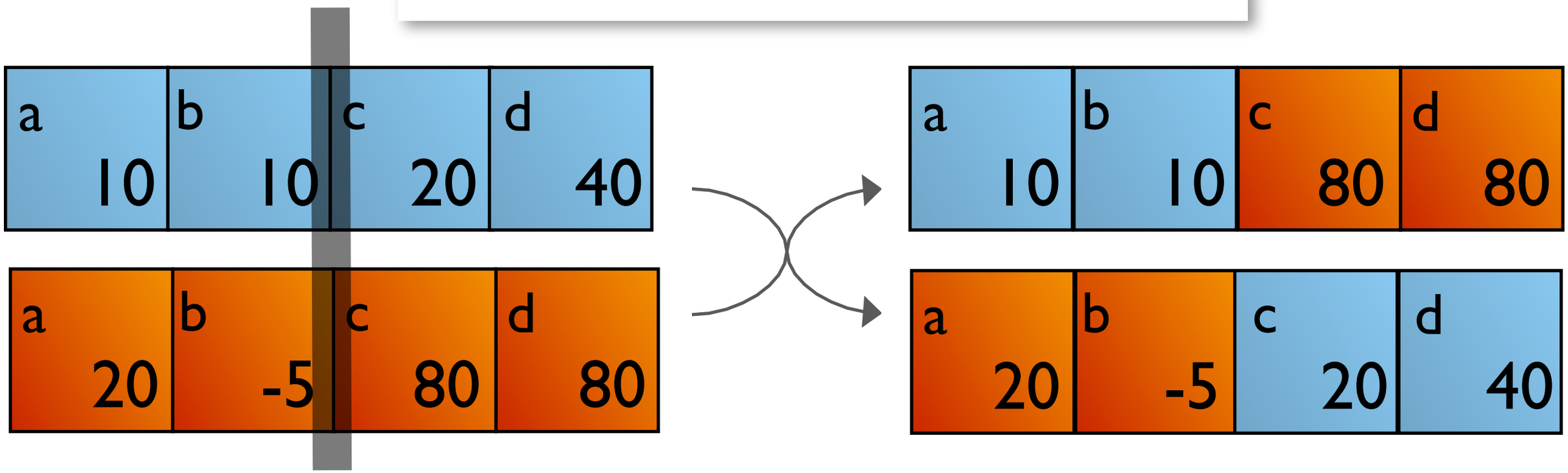
Crossover

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```



Crossover

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```



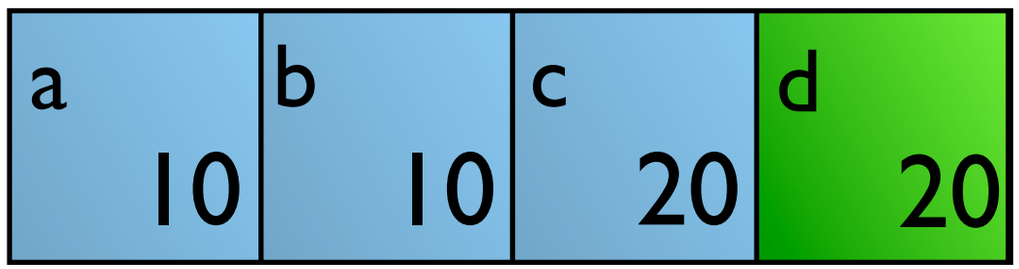
Mutation

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```

a	b	c	d
10	10	20	40

Mutation

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```



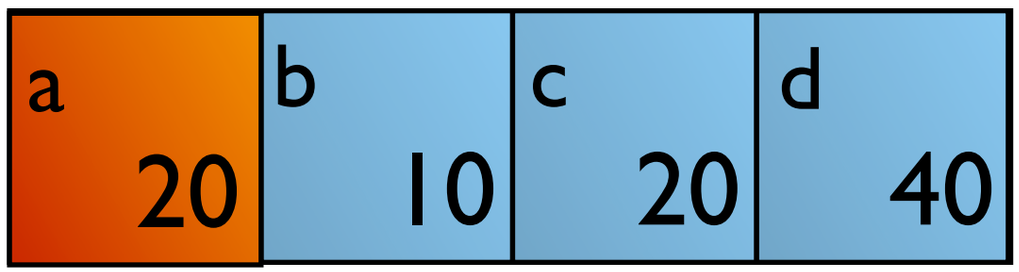
Mutation

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```

a	b	c	d
10	10	20	40

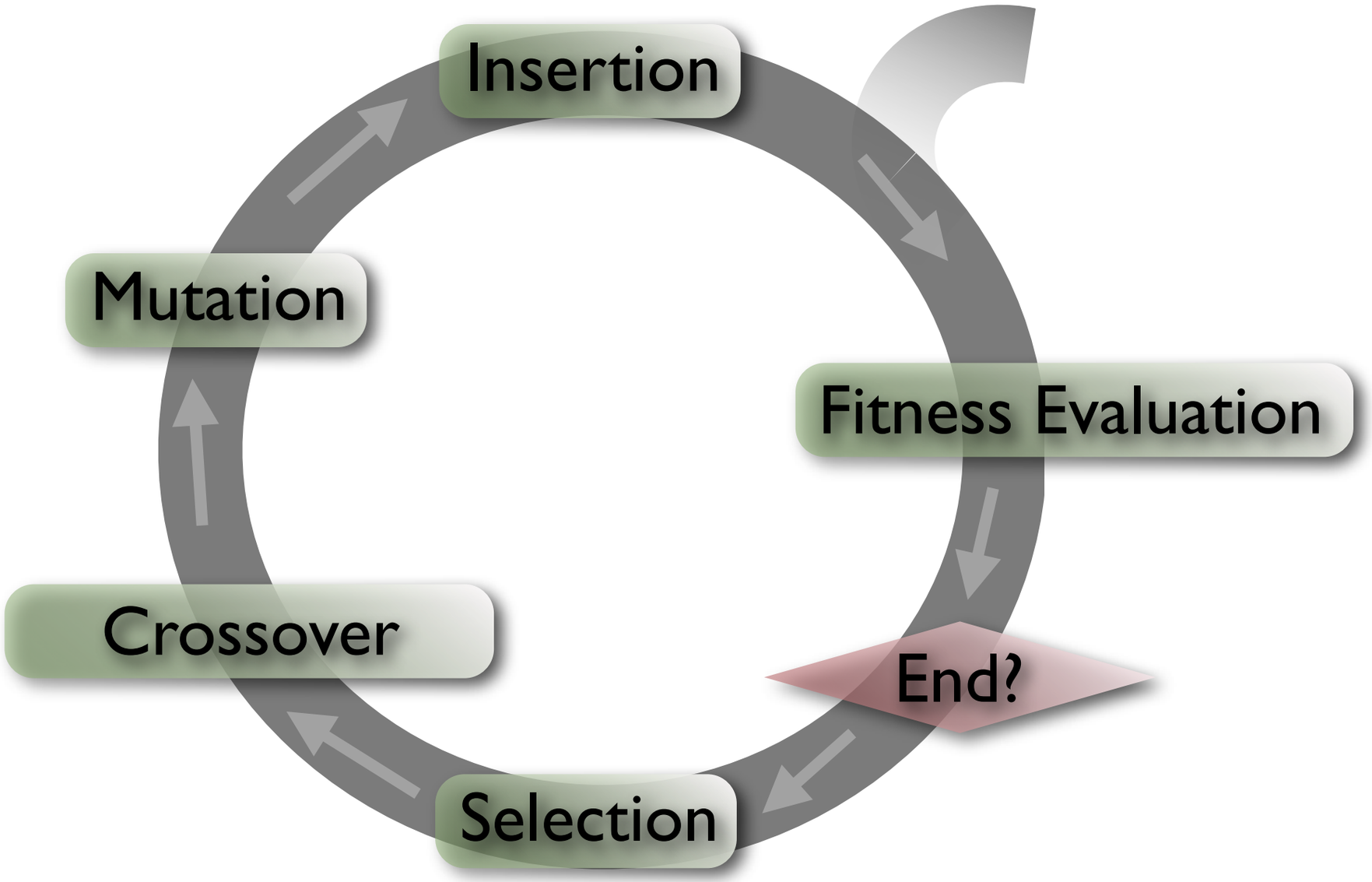
Mutation

```
void test_me(int a, int b, int c, int d) {

   if (a == b) {

       if (c == d) {

           // branch we want to execute

       }
   }

   ...
```

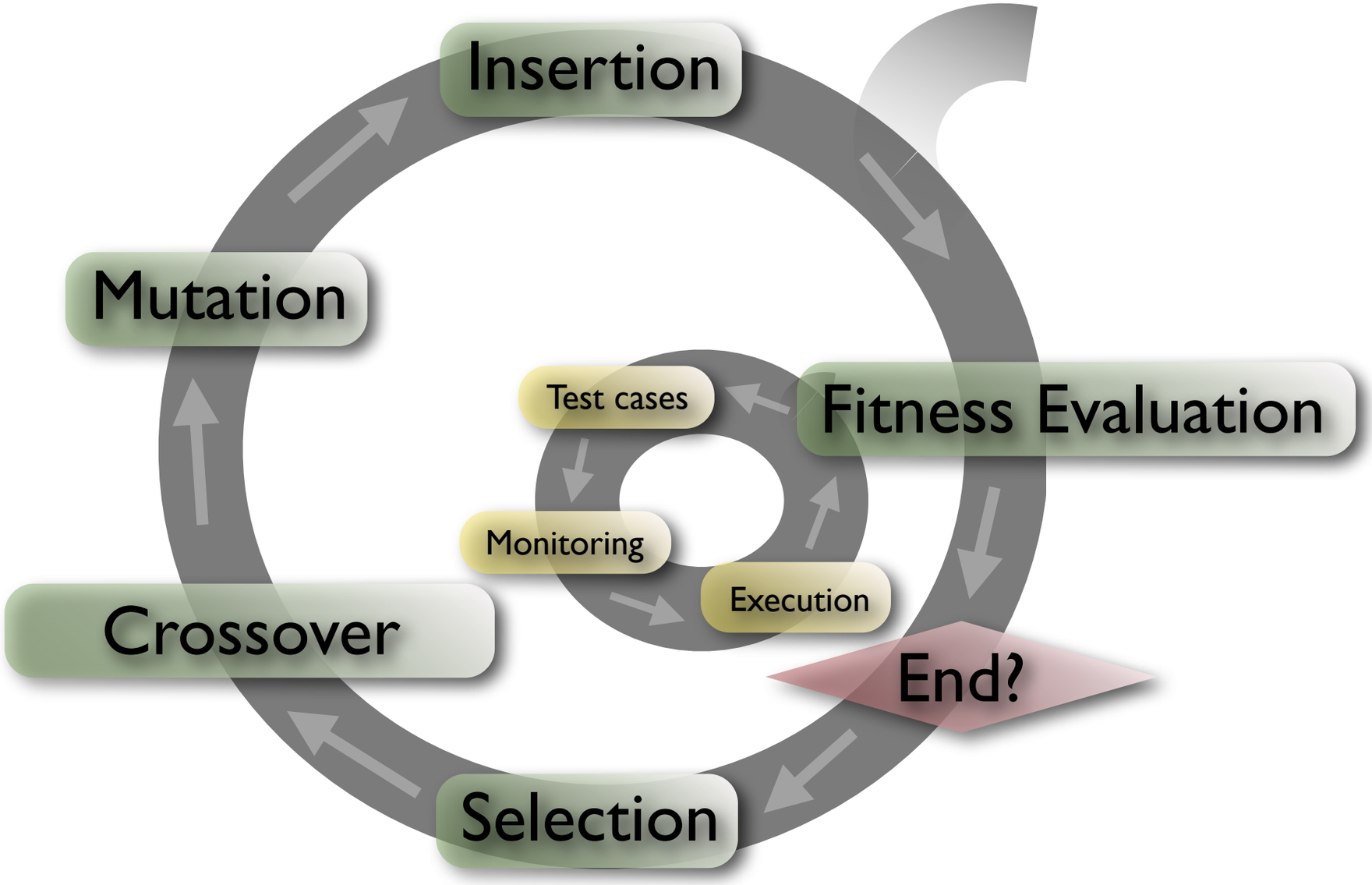


Evolutionary Testing

Evolutionary Testing



Evolutionary Testing

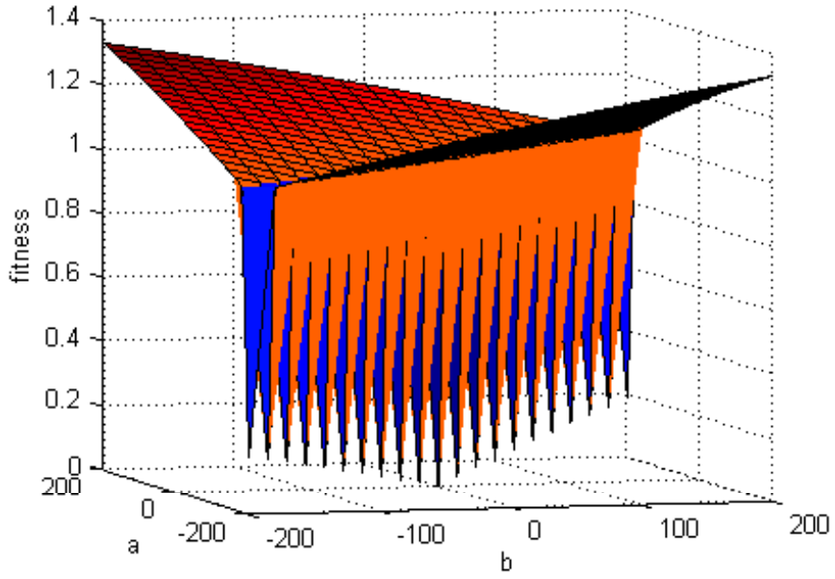


Which search method?

Depends on characteristics of the search landscape

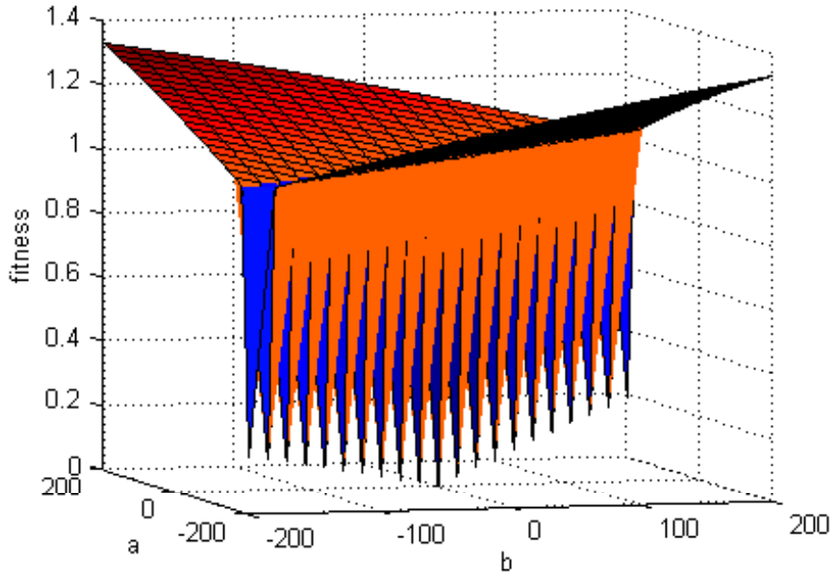
Which search method?

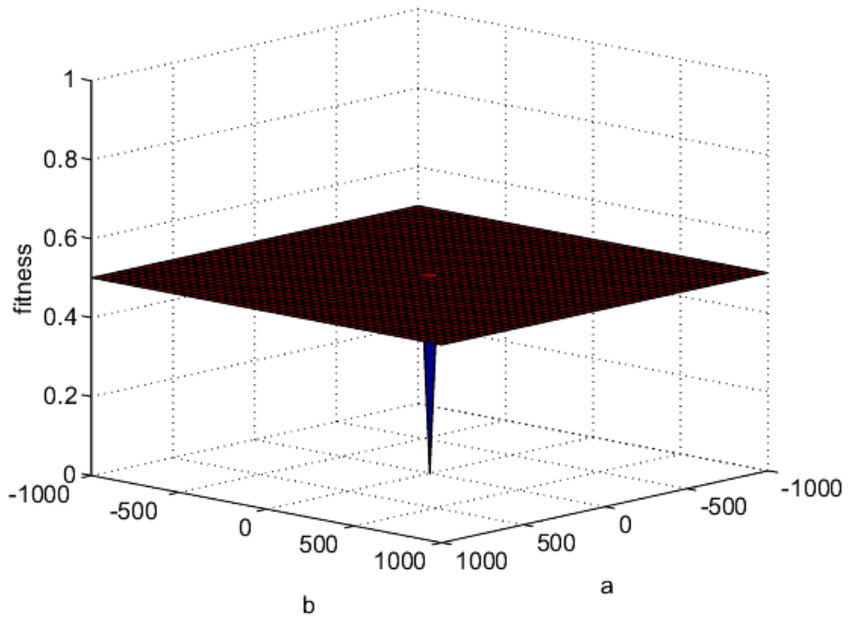
Depends on characteristics of the search landscape



Which search method?

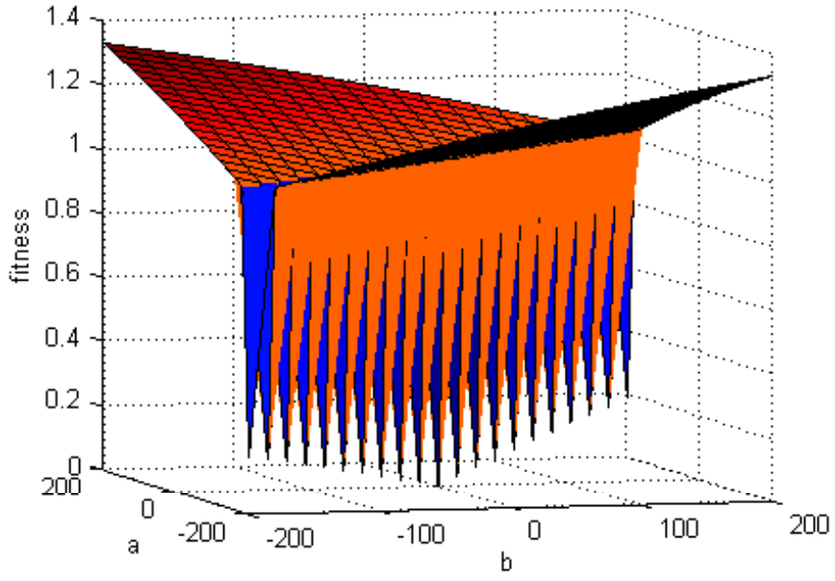
Depends on characteristics of the search landscape

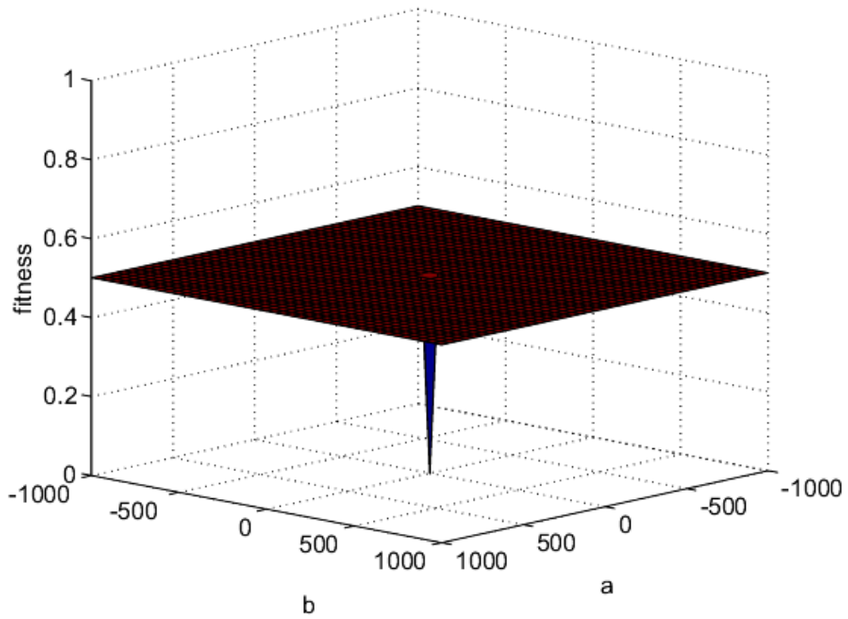


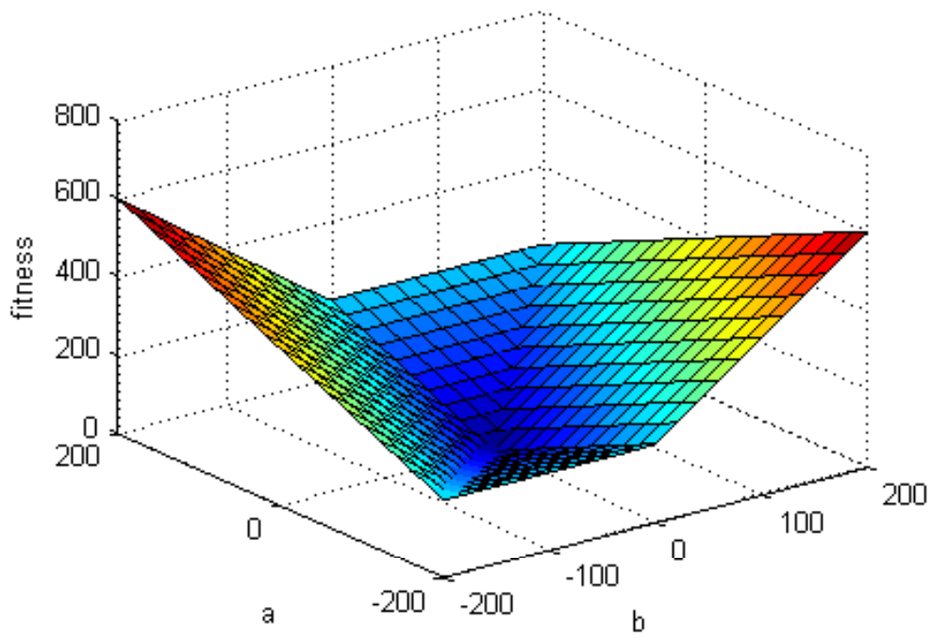


Which search method?

Depends on characteristics of the search landscape

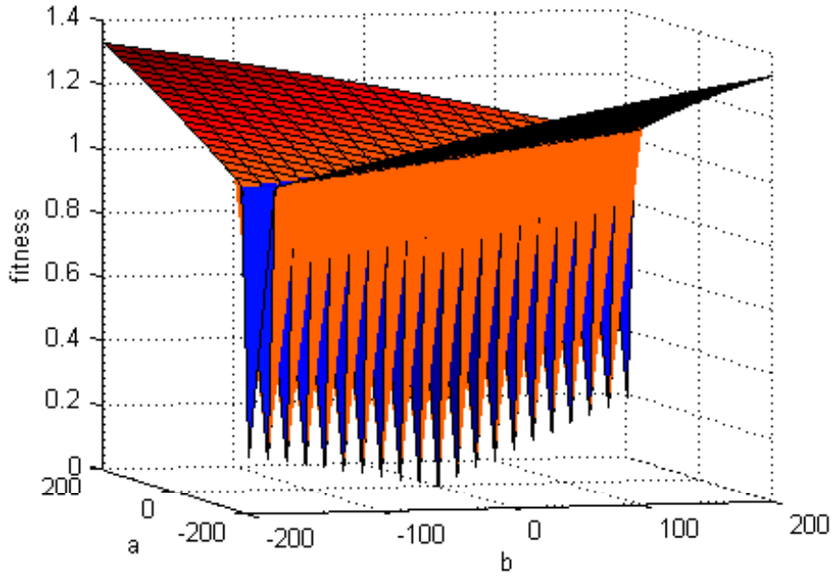


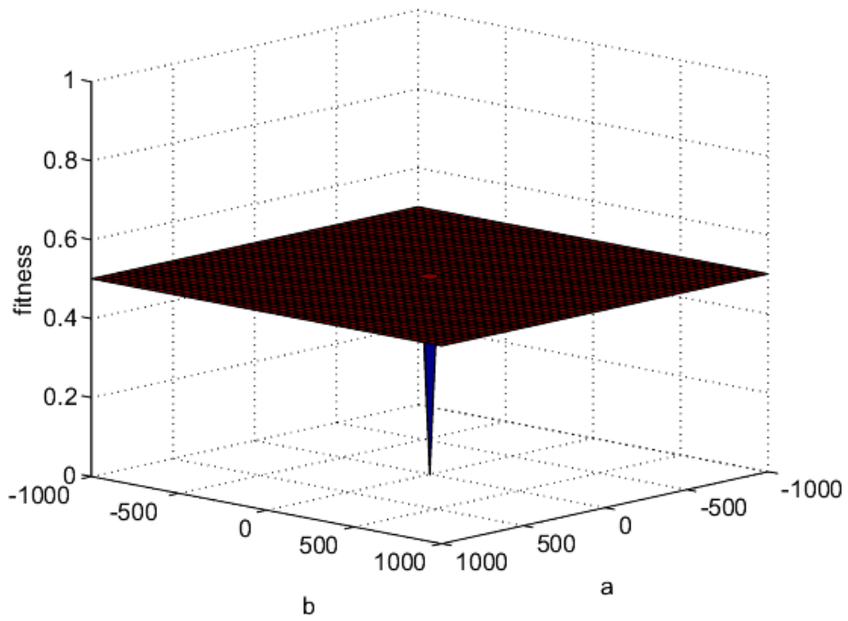


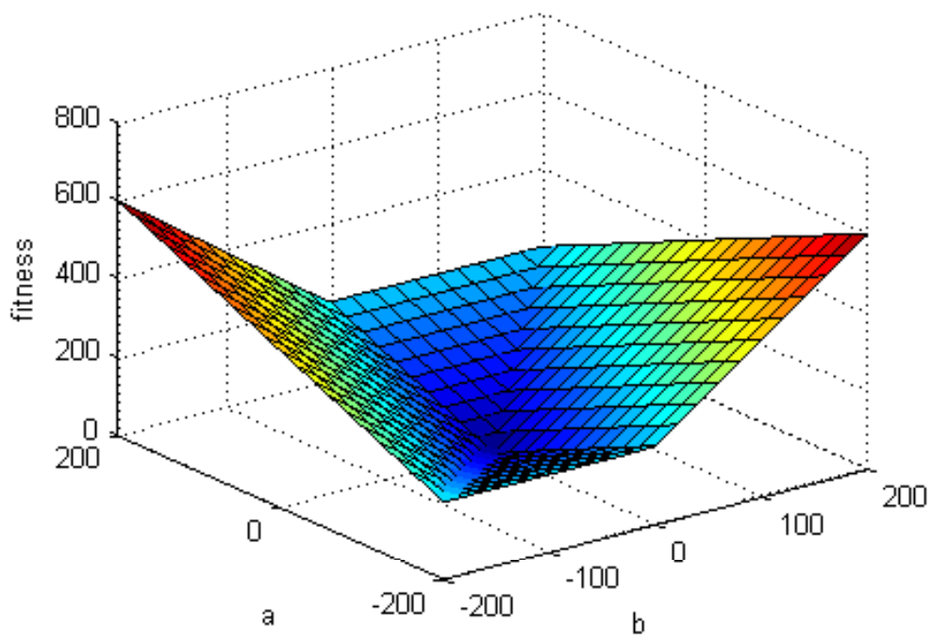


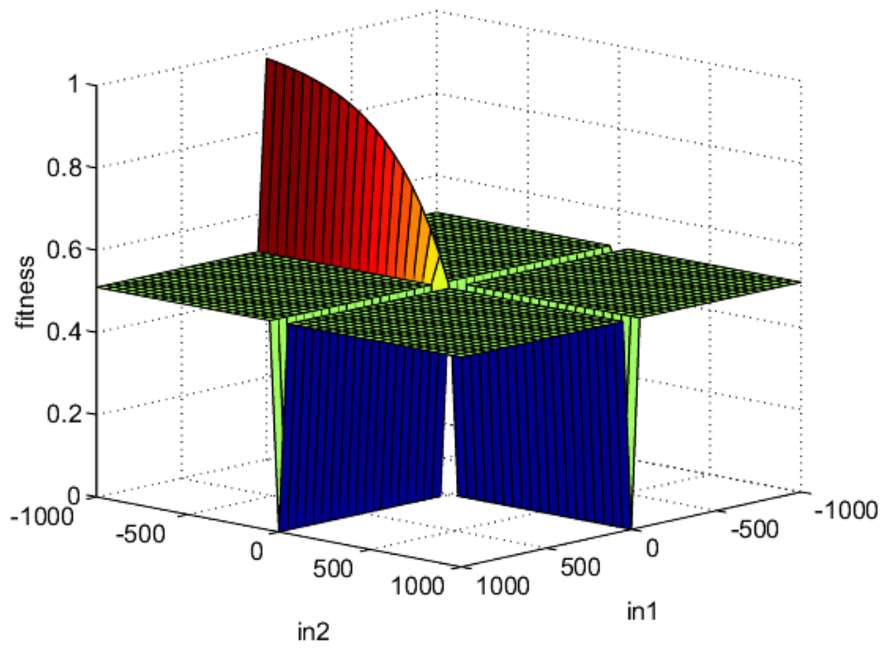
Which search method?

Depends on characteristics of the search landscape







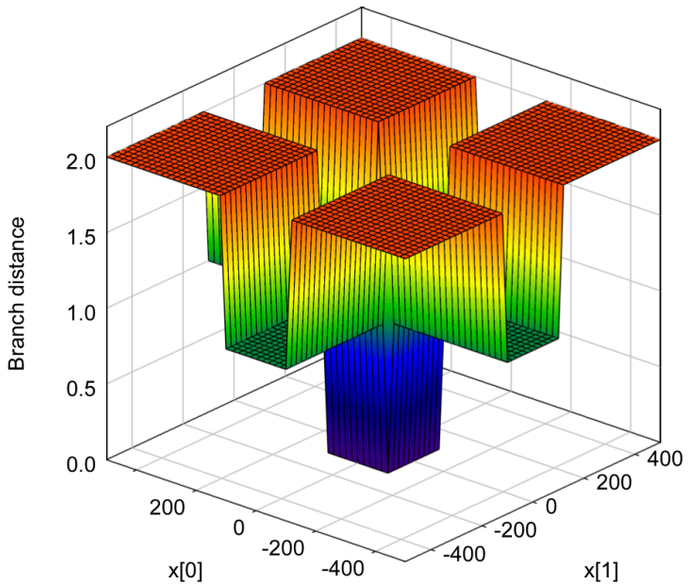


Which search method?

Some landscapes are hard for some searches but easy for others

...and vice versa...

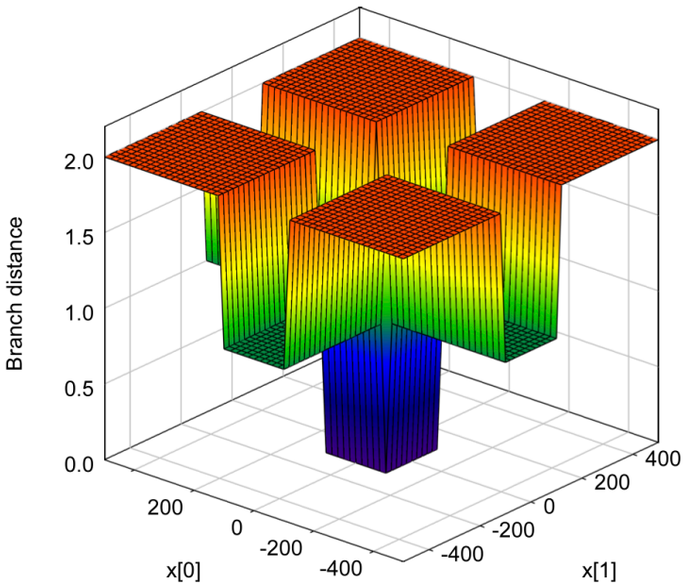
Which search method?



Some landscapes are hard for some searches but easy for others

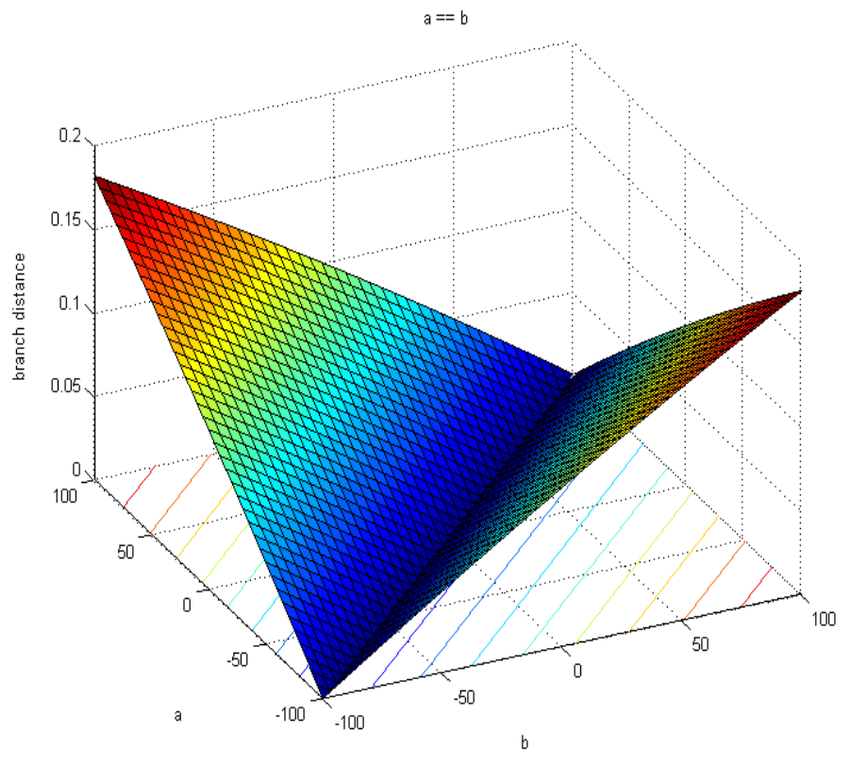
...and vice versa...

Which search method?

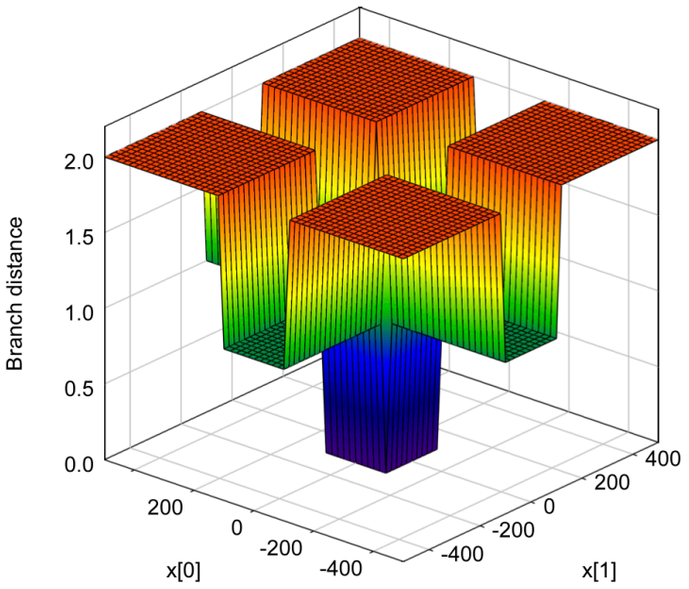


Some landscapes are hard for some searches but easy for others

...and vice versa...

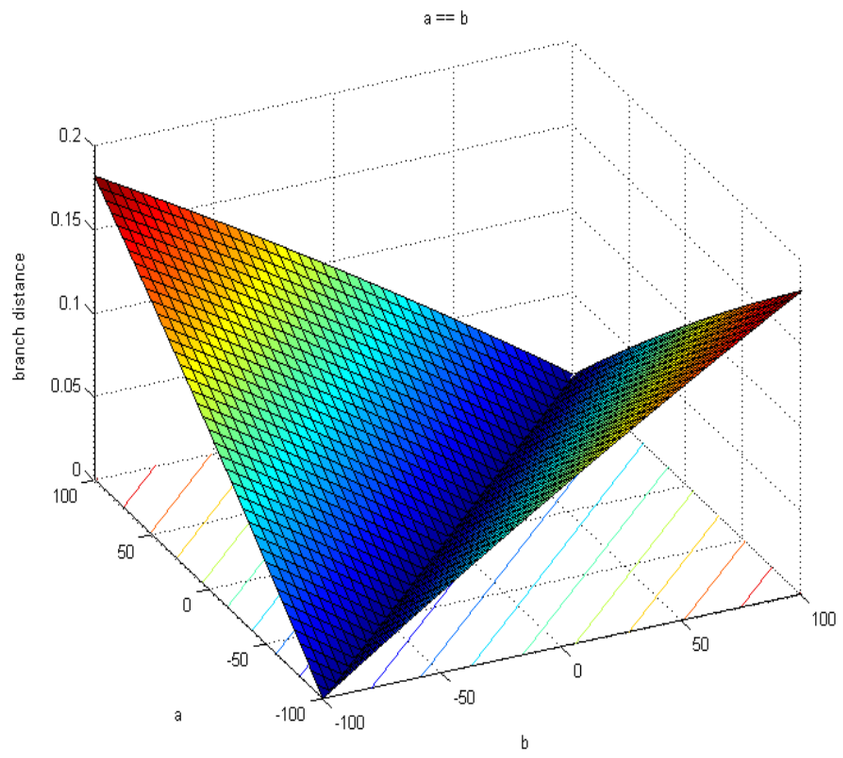


Which search method?



Some landscapes are hard for some searches but easy for others

...and vice versa...



more on this later...

Ingredients for an optimising search algorithm

Representation

Neighbourhood

Fitness Function

Ingredients for Search-Based Testing

Representation

Neighbourhood

Fitness Function

Ingredients for Search-Based Testing

Representation	A method of encoding all possible inputs
Neighbourhood	Usually straightforward
Fitness Function	Inputs are already in data structures

Ingredients for Search-Based Testing

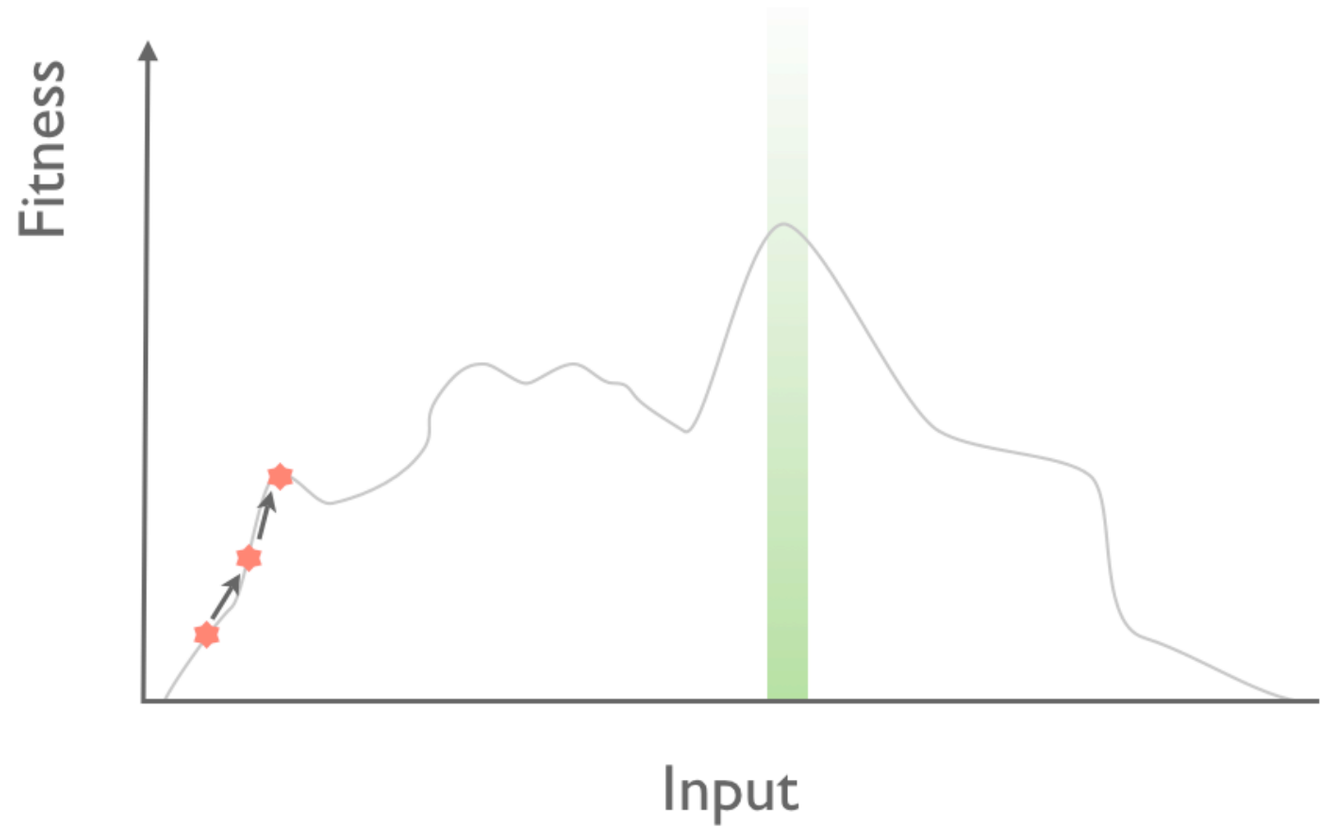
Representation

Part of our understanding of the problem

Neighbourhood

We need to know our near neighbours

Fitness Function



Ingredients for Search-Based Testing

Representation

Transformation of the test goal to a numerical function

Neighbourhood

Fitness Function

Numerical values indicate how ‘good’ an input is

More search algorithms

More search algorithms

Tabu Search

Particle Swarm Optimisation

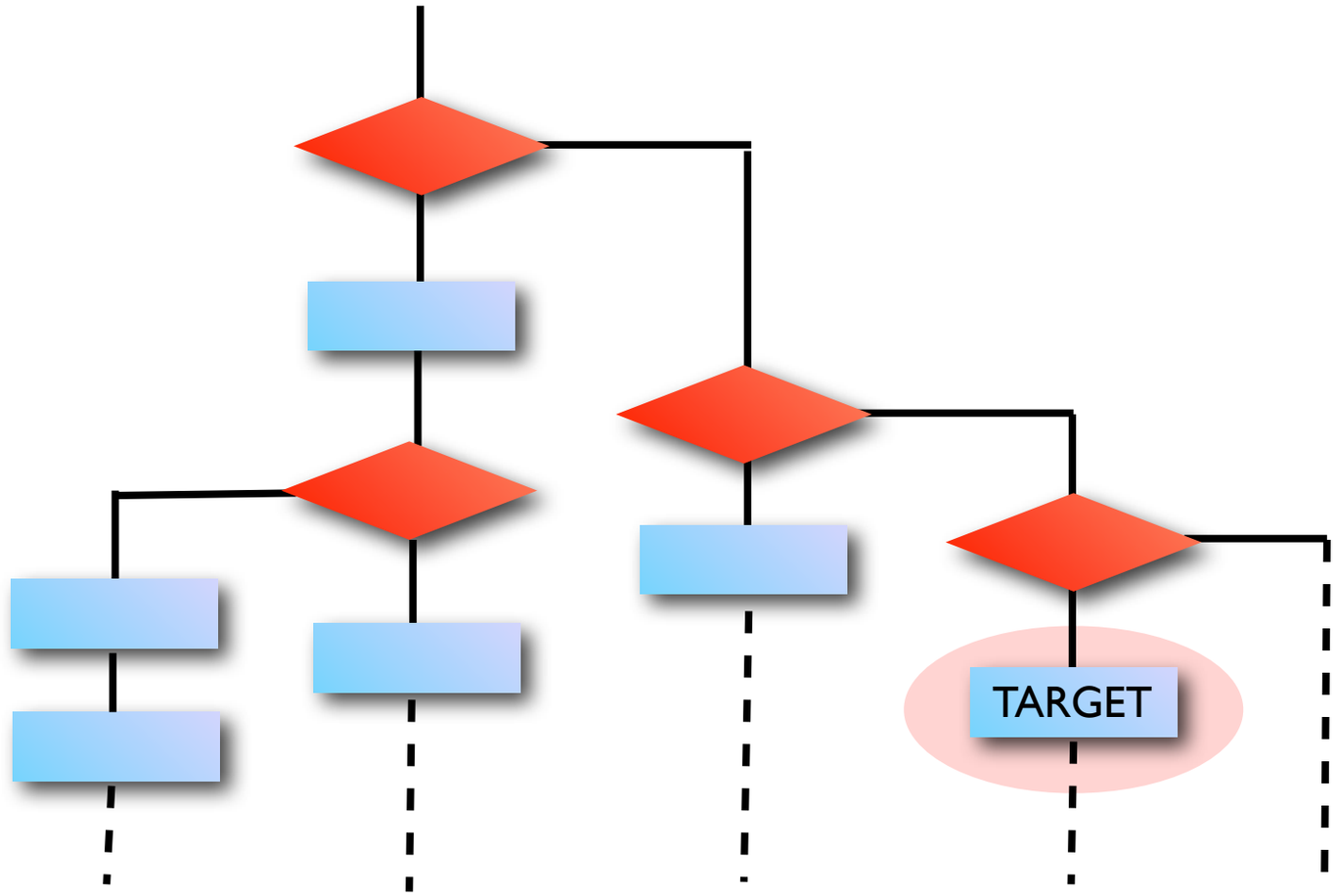
Ant Colony Optimisation

Genetic Programming

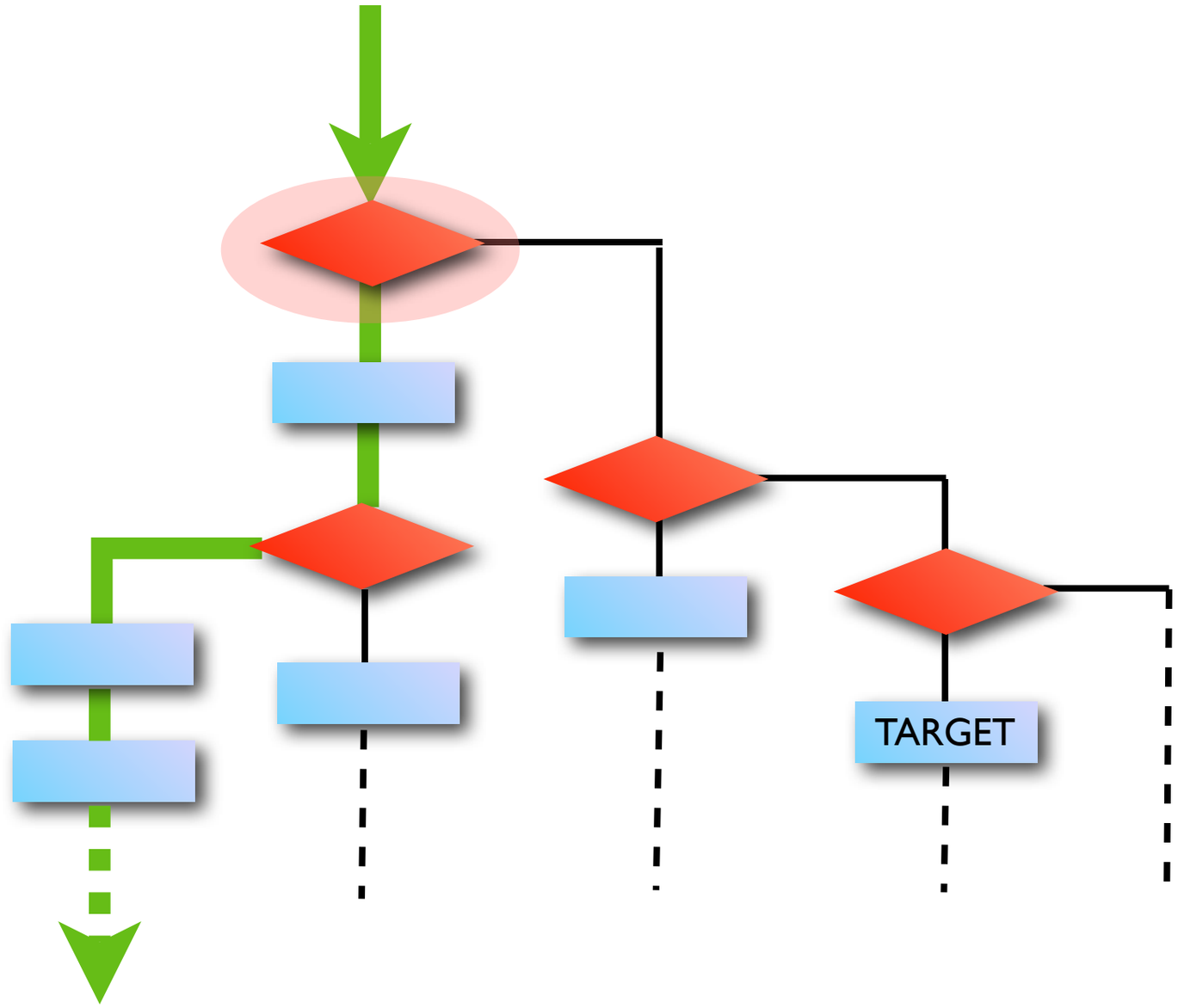
Estimation of Distribution Algorithms

Search-Based Structural Test Data Generation

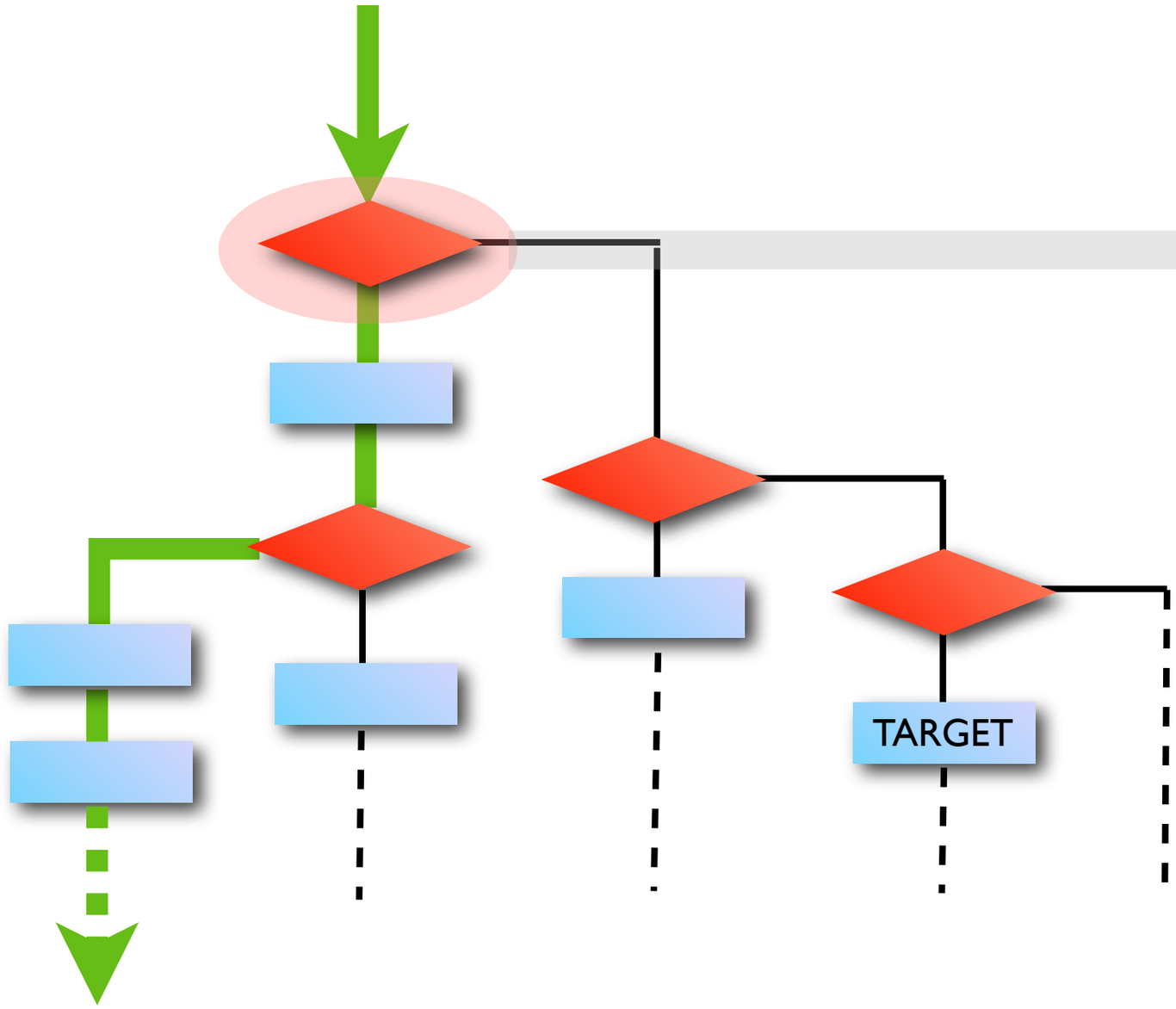
Covering a structure



Fitness evaluation

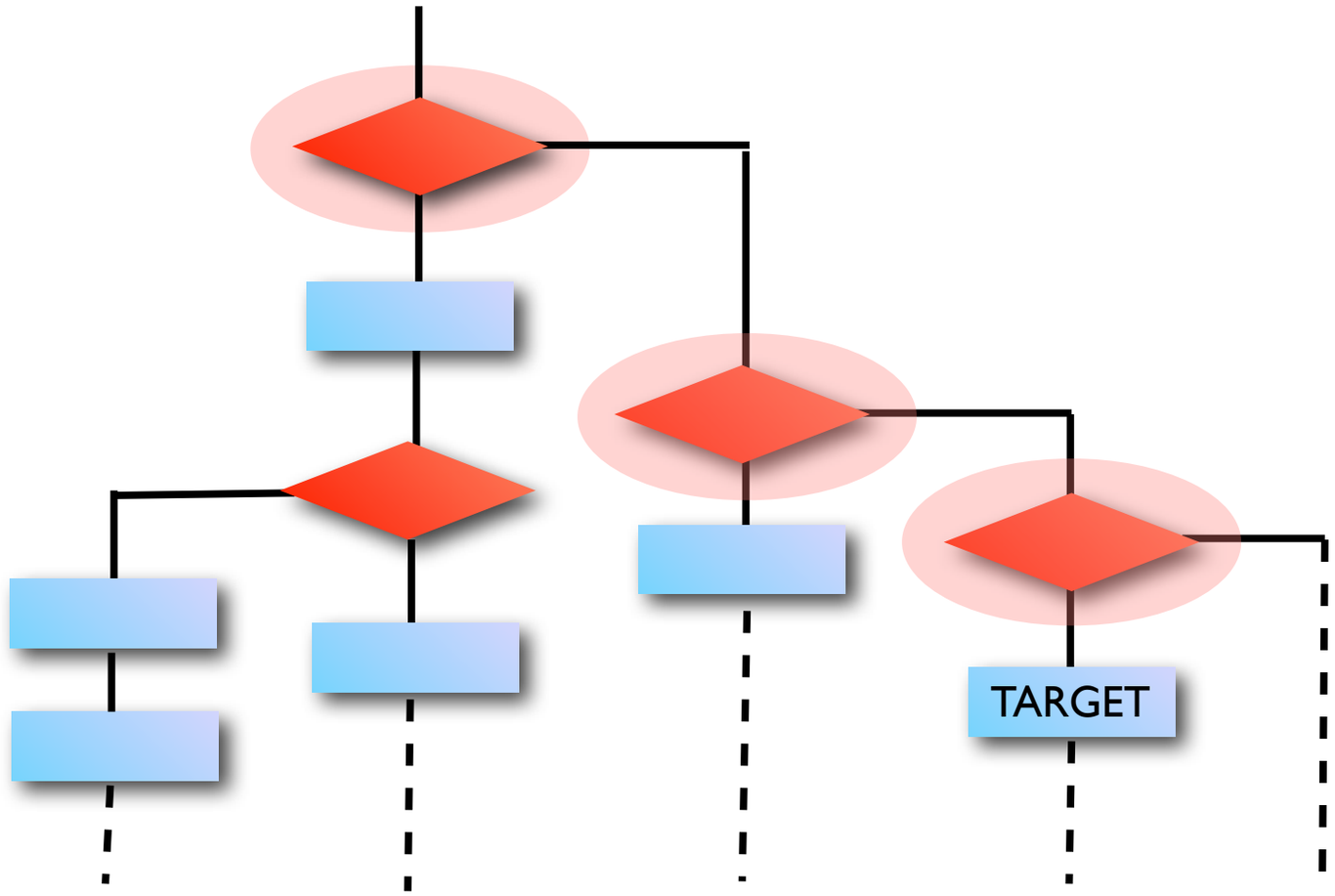


Fitness evaluation

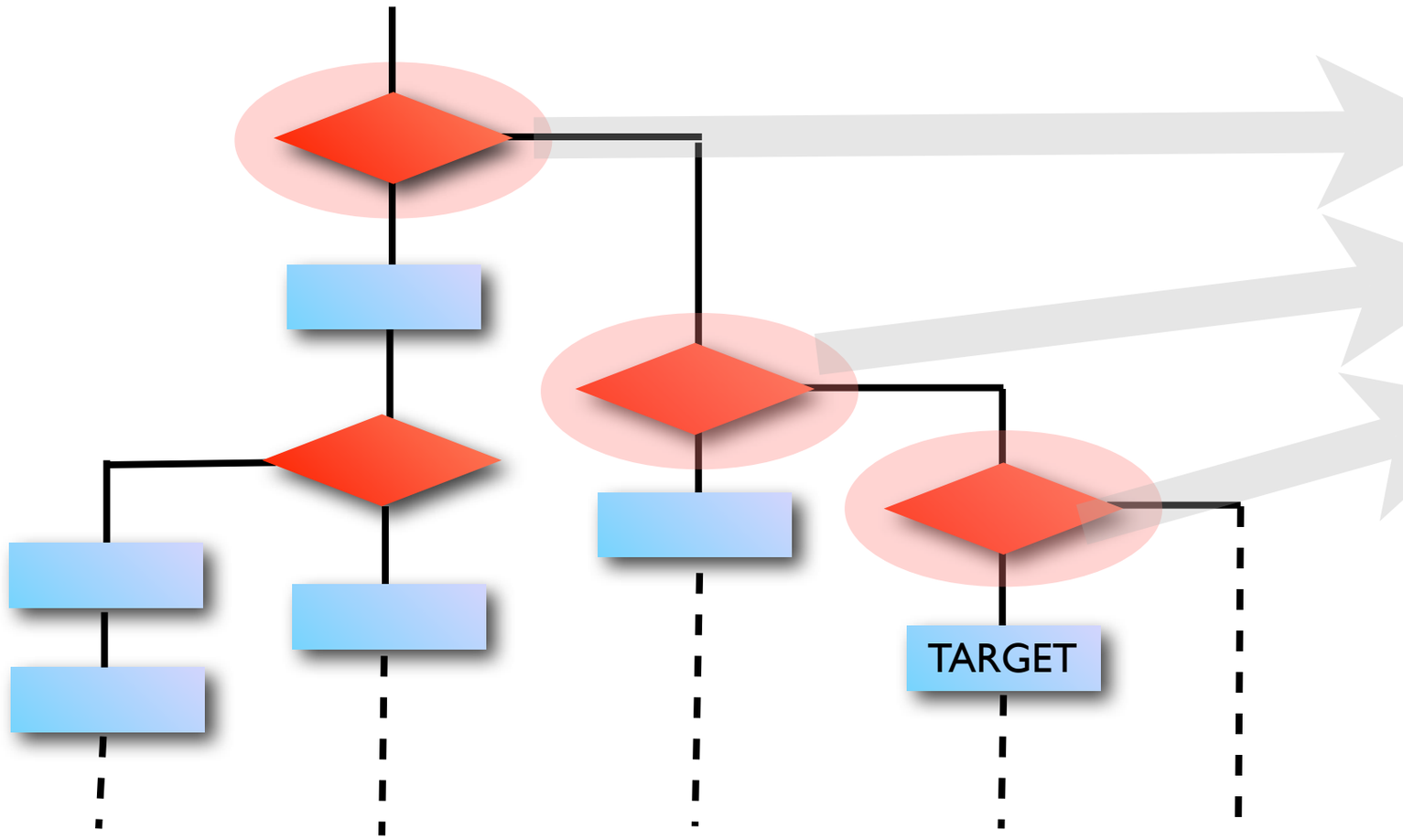


The test data executes the ‘wrong’ path

Analysing control flow



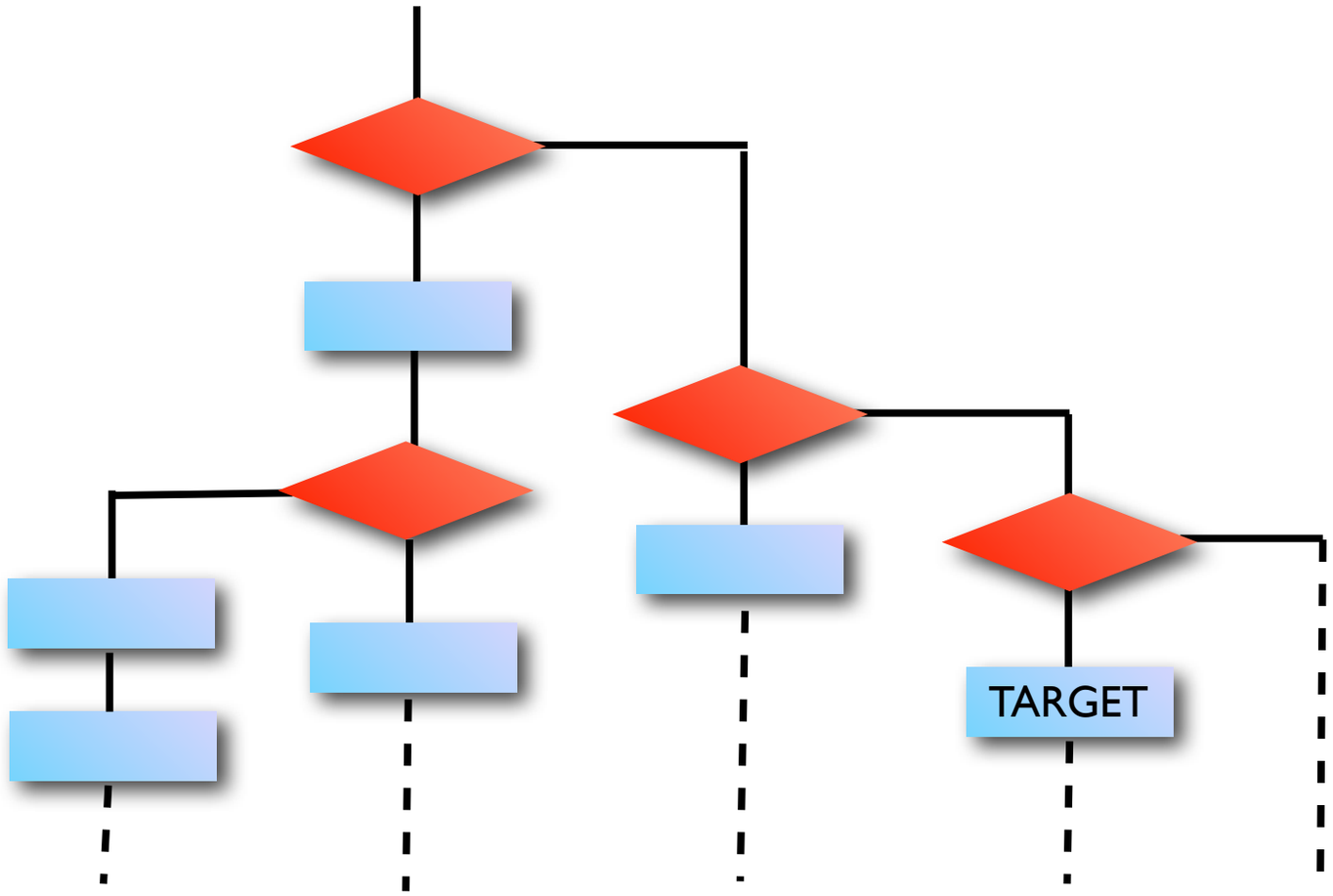
Analysing control flow



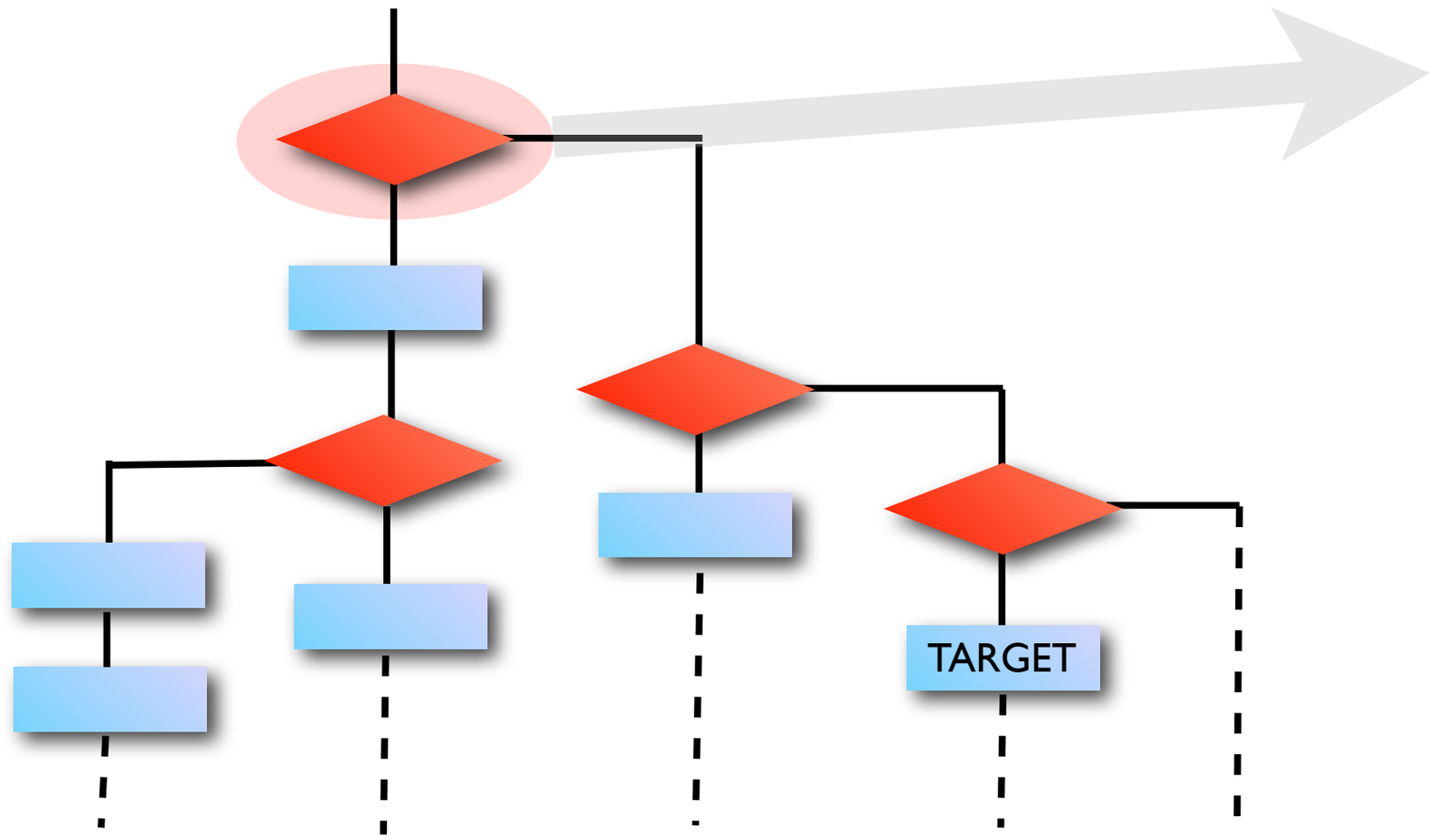
The outcomes at key decision statements matter.

These are the decisions on which the target is control dependent

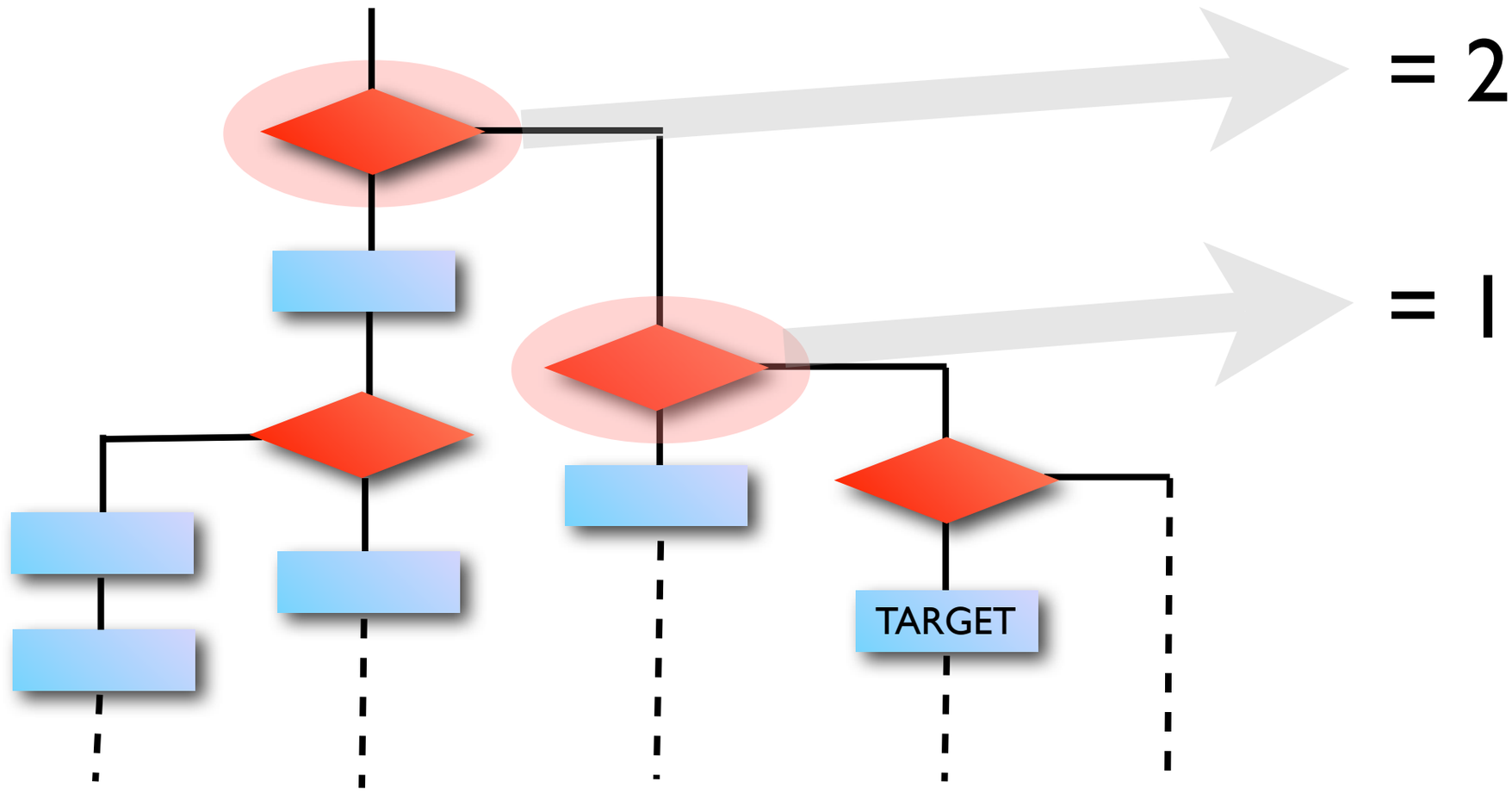
Approach Level



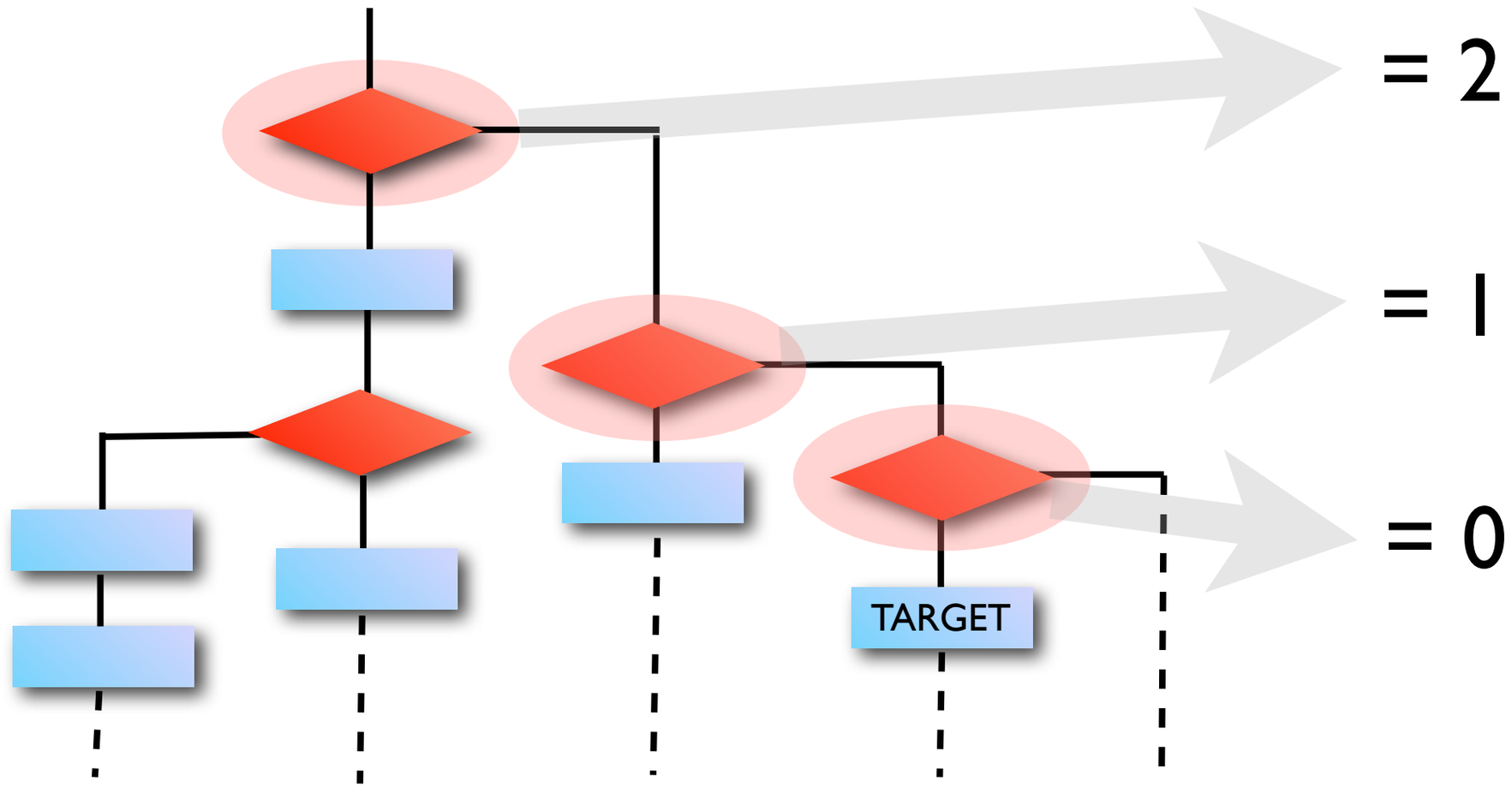
Approach Level



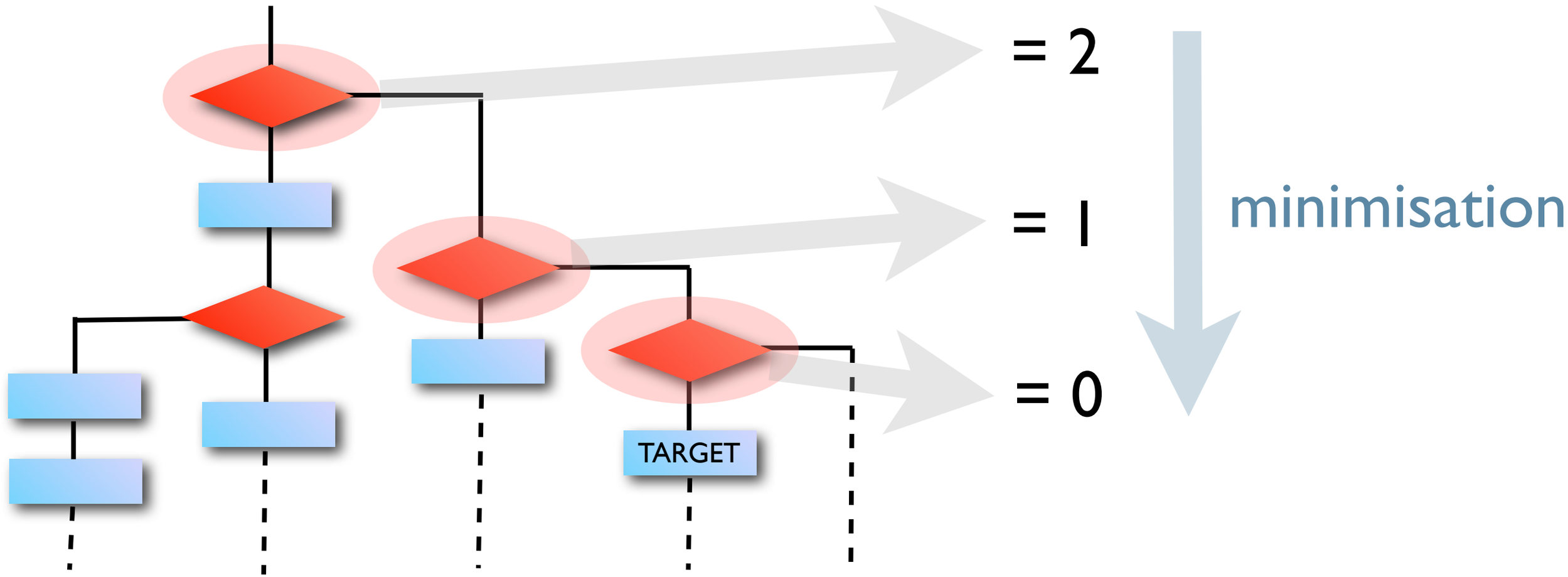
Approach Level



Approach Level



Approach Level



Analysing predicates

Approach level alone gives us coarse values

```
if (a == b) {
    // ....
}
```

a = 50, b = 0
a = 45, b = 5
a = 40, b = 10
a = 35, b = 15
a = 30, b = 20
a = 25, b = 25

Analysing predicates

Approach level alone gives us coarse values

```
if (a == b) {
    // ....
}
```

a = 50, b = 0
a = 45, b = 5
a = 40, b = 10
a = 35, b = 15
a = 30, b = 20
a = 25, b = 25

getting ‘closer’ to being true

Branch distance

Associate a distance formula with different relational predicates

```
if (a == b) {
    // ....
}
```

abs(a-b)

a = 50, b = 0	branch distance = 50
a = 45, b = 5	branch distance = 40
a = 40, b = 10	branch distance = 30
a = 35, b = 15	branch distance = 20
a = 30, b = 20	branch distance = 10
a = 25, b = 25	branch distance = 0

getting ‘closer’ to being true

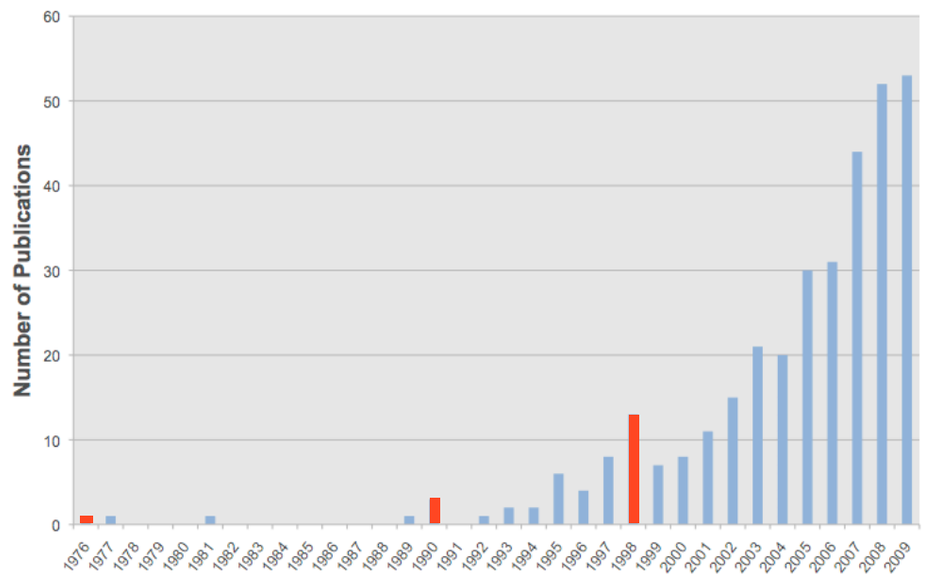
Branch distances for relational predicates

Relational predicate	Objective function *obj*
Boolean	if *TRUE* then 0 else K
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$\neg a$	Negation is moved inwards and propagated over a

Webb Miller & D. Spooner. Automatic Generation of Floating-Point Test Data. IEEE Trans. Software Eng. 1976

Bogdan Korel. Automated Software Test Data Generation. IEEE Trans. Software Eng. 1990

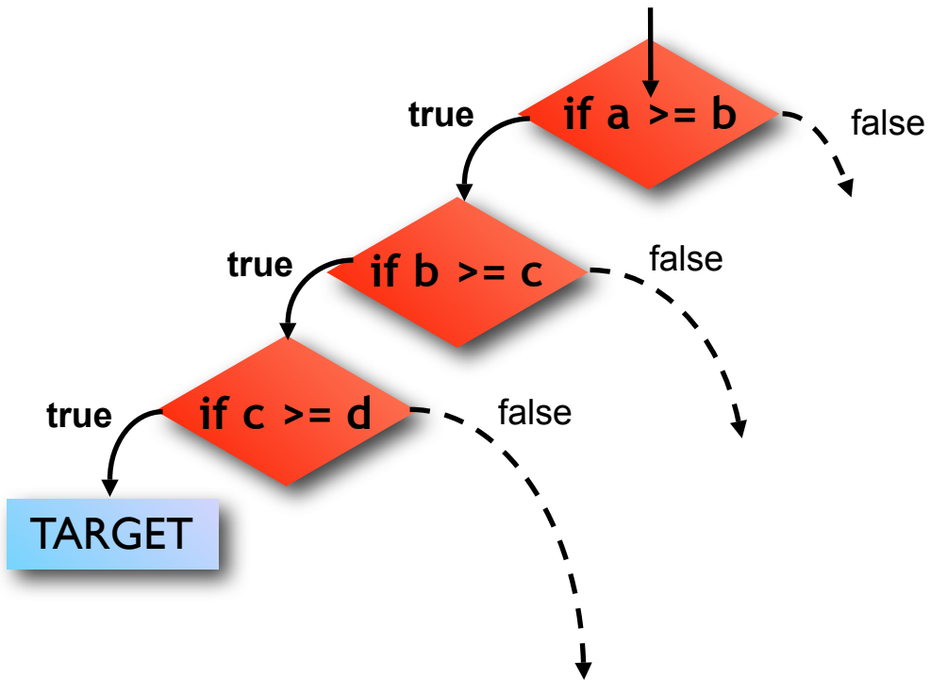
Nigel Tracey, John Clark & Keith Mander. The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach. 1998



Putting it all together

Fitness = approach Level + *normalised* branch distance

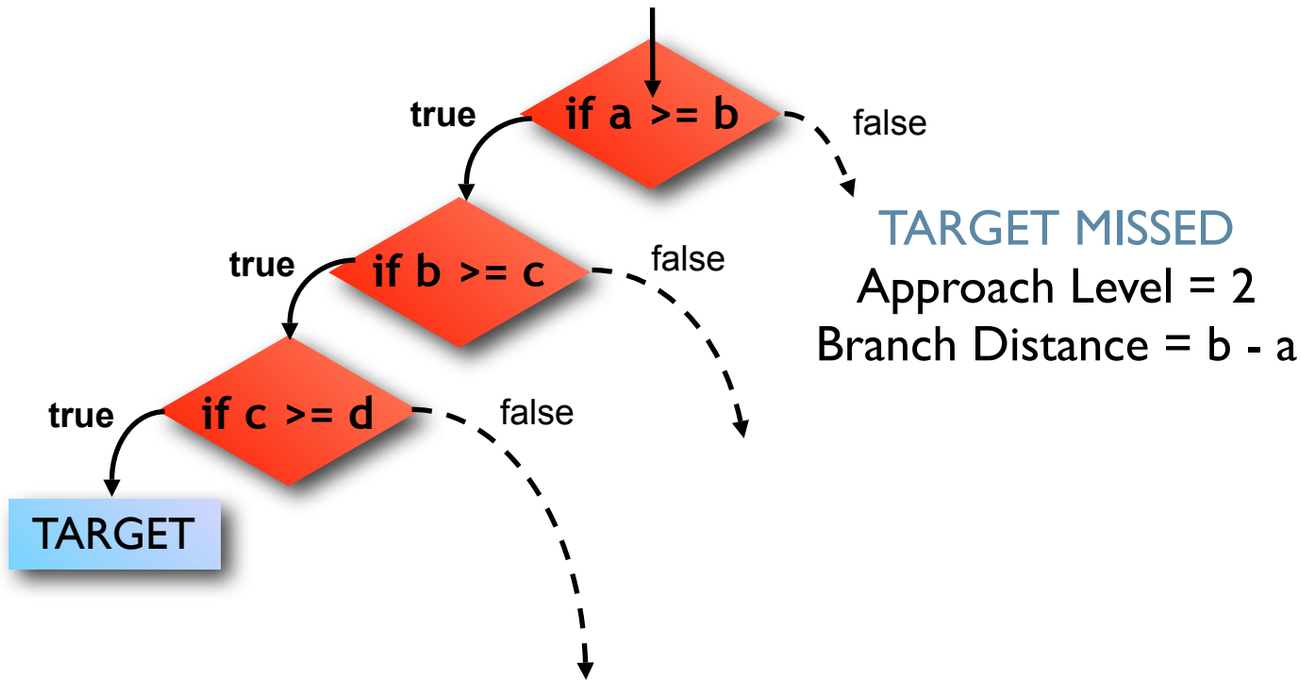
```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
. . .
```



Putting it all together

Fitness = approach Level + *normalised* branch distance

```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
. . .
```



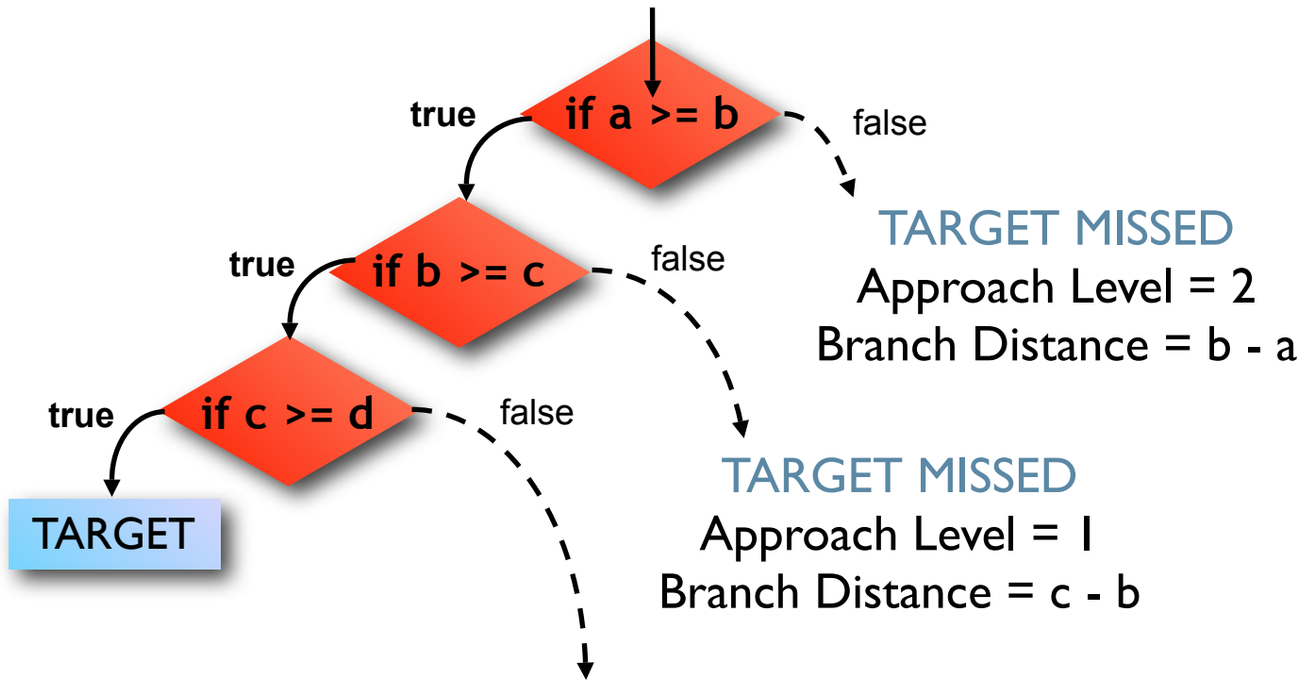
normalised branch distance between 0 and 1

indicates how close approach level is to being penetrated

Putting it all together

Fitness = approach Level + *normalised* branch distance

```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
. . .
```



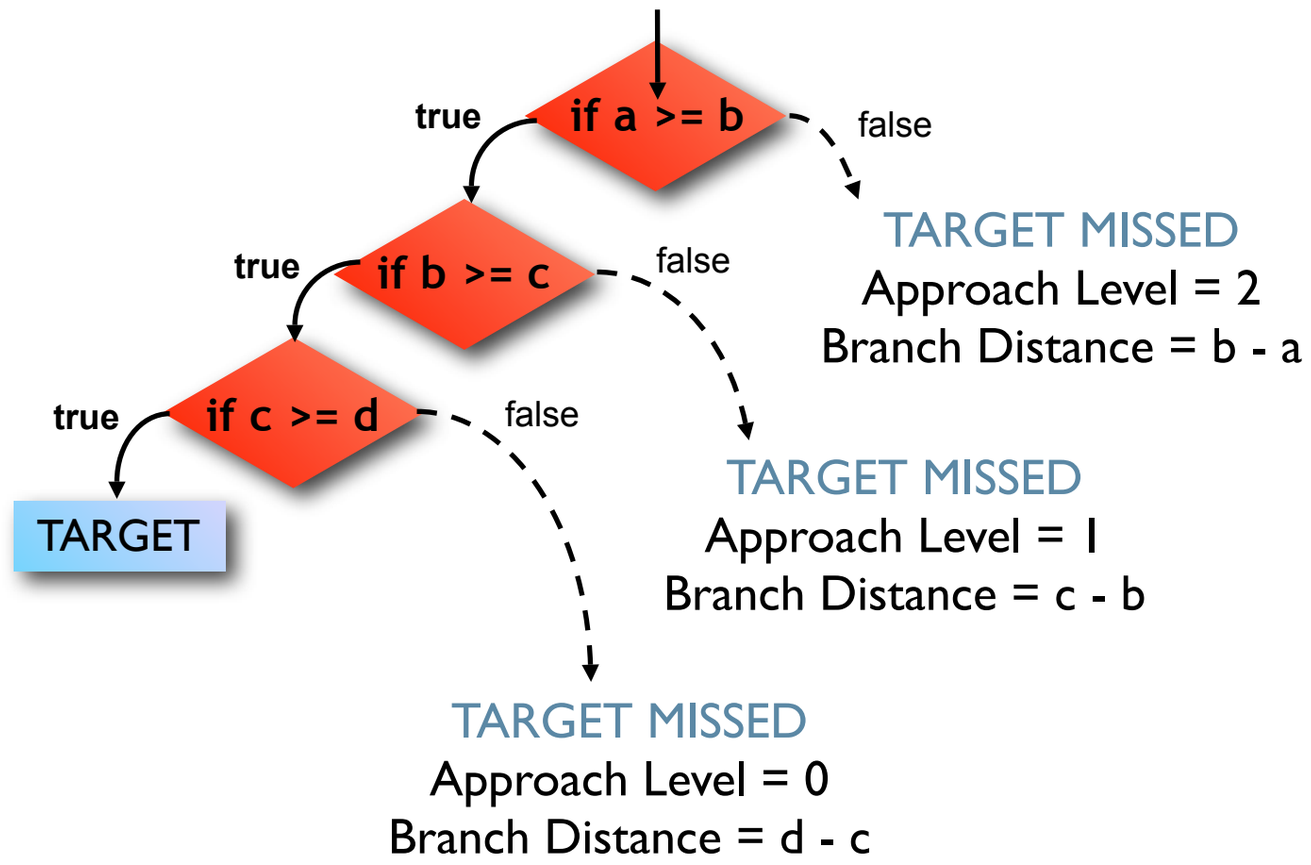
normalised branch distance between 0 and 1

indicates how close approach level is to being penetrated

Putting it all together

Fitness = approach Level + *normalised* branch distance

```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
. . .
```



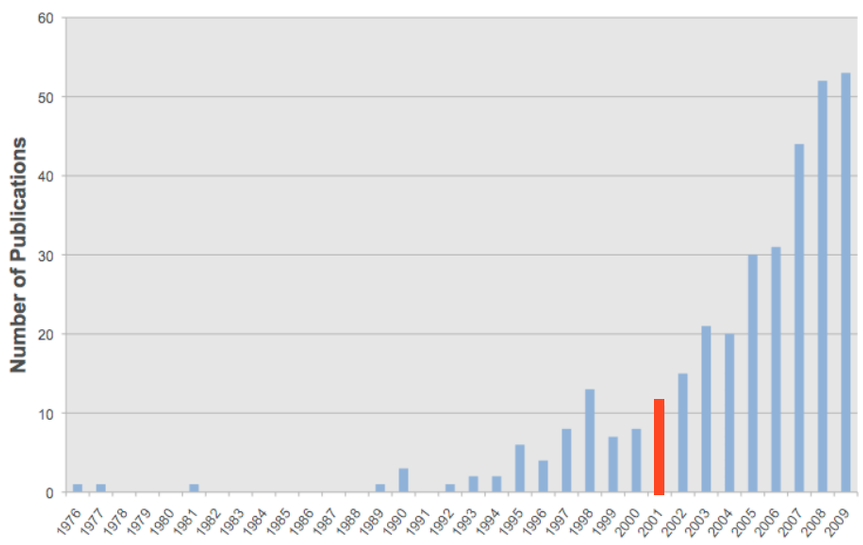
normalised branch distance between 0 and 1
indicates how close approach level is to being penetrated

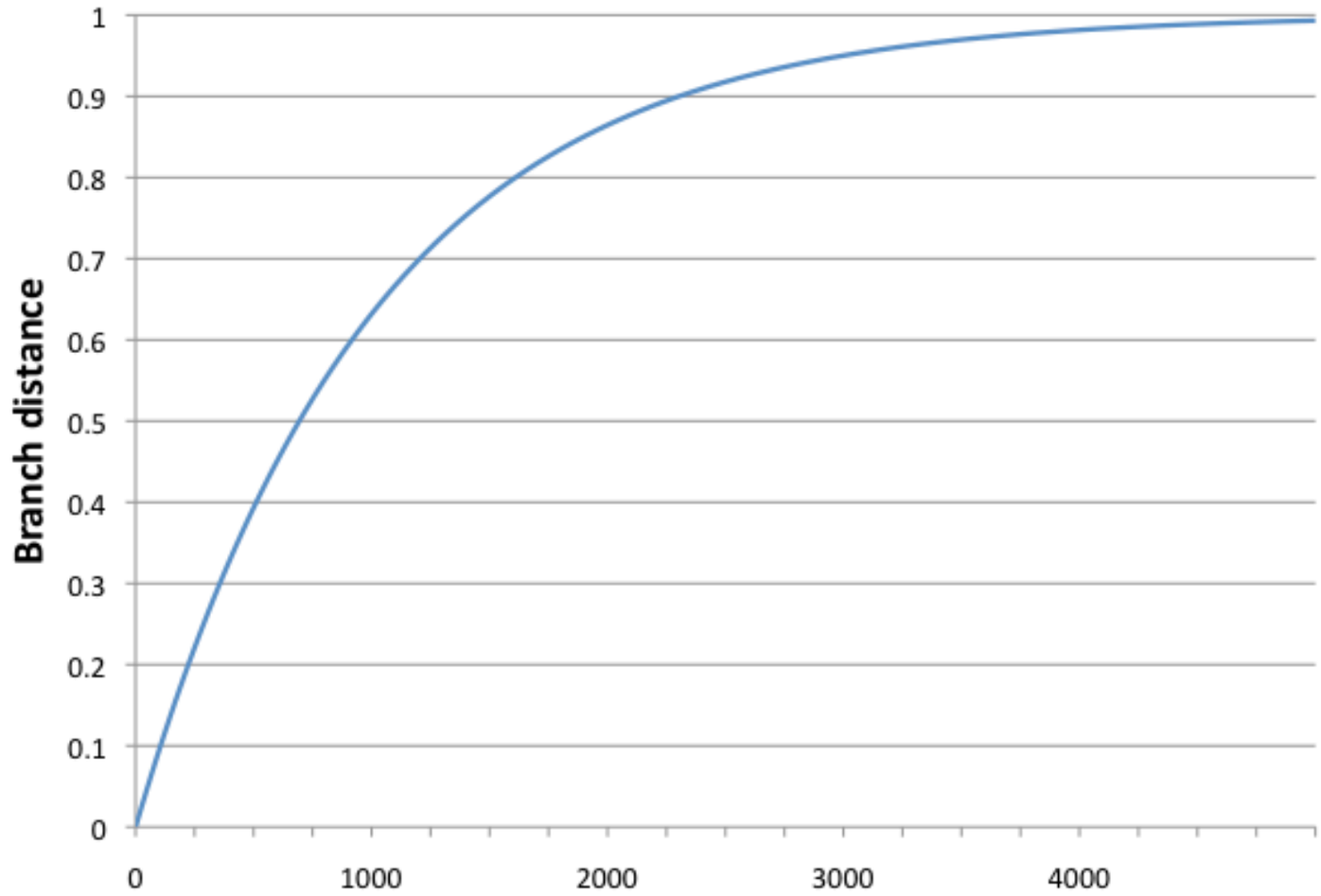
Normalisation Functions

Since the ‘maximum’ branch distance is generally unknown we need a non-standard normalisation function

$$1 - \alpha^{-x}$$

Baresel (2000), alpha = 1.001



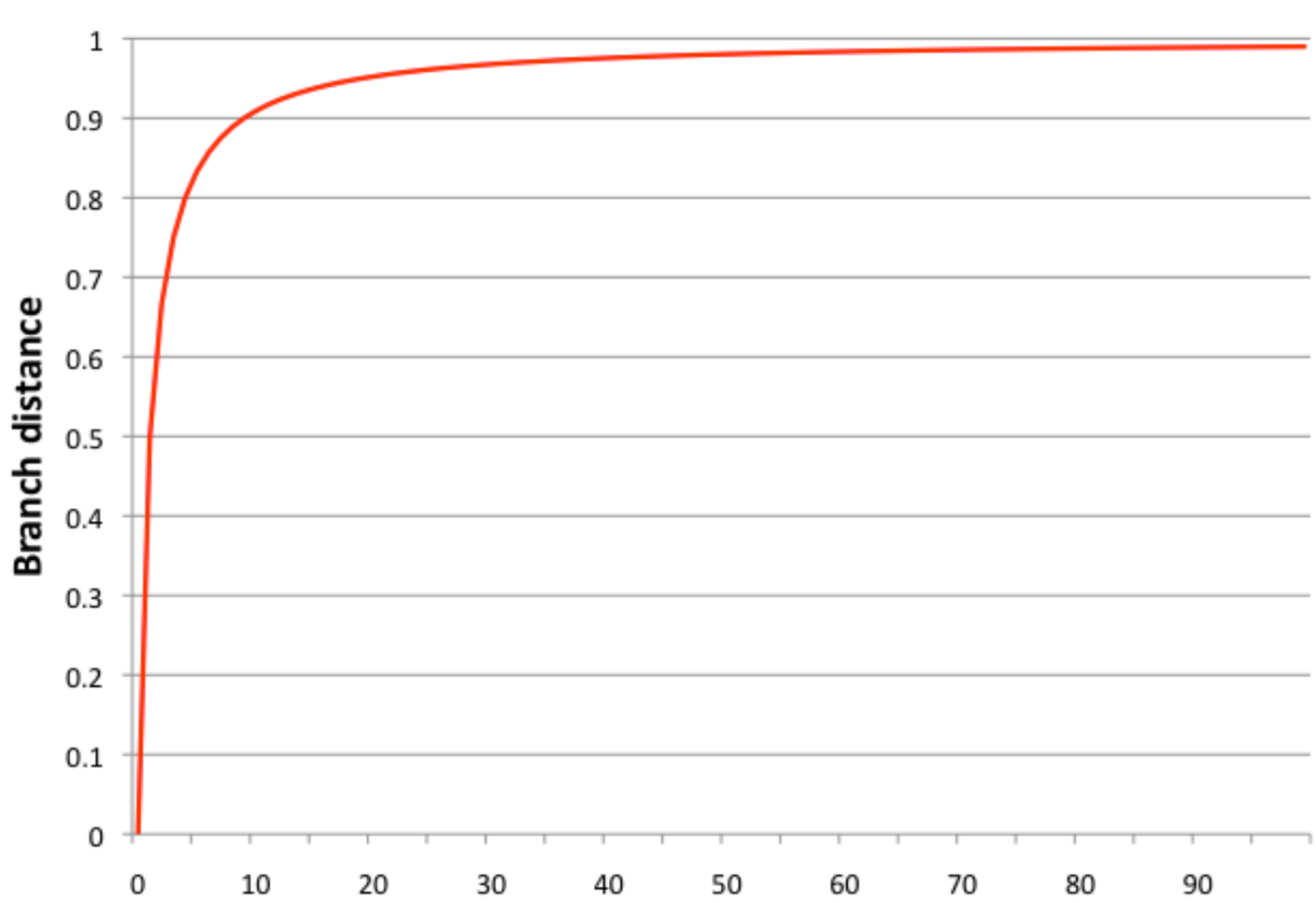


Normalisation Functions

Since the ‘maximum’ branch distance is generally unknown we need a non-standard normalisation function

$$\frac{x}{x+\beta}$$

Arcuri (2010), beta = 1



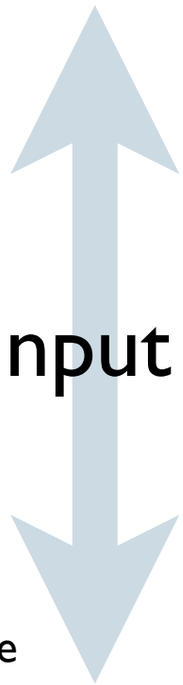
Alternating Variable Method

‘Probe’ moves

void fn(input1, input2, input3 ....)

Alternating Variable Method

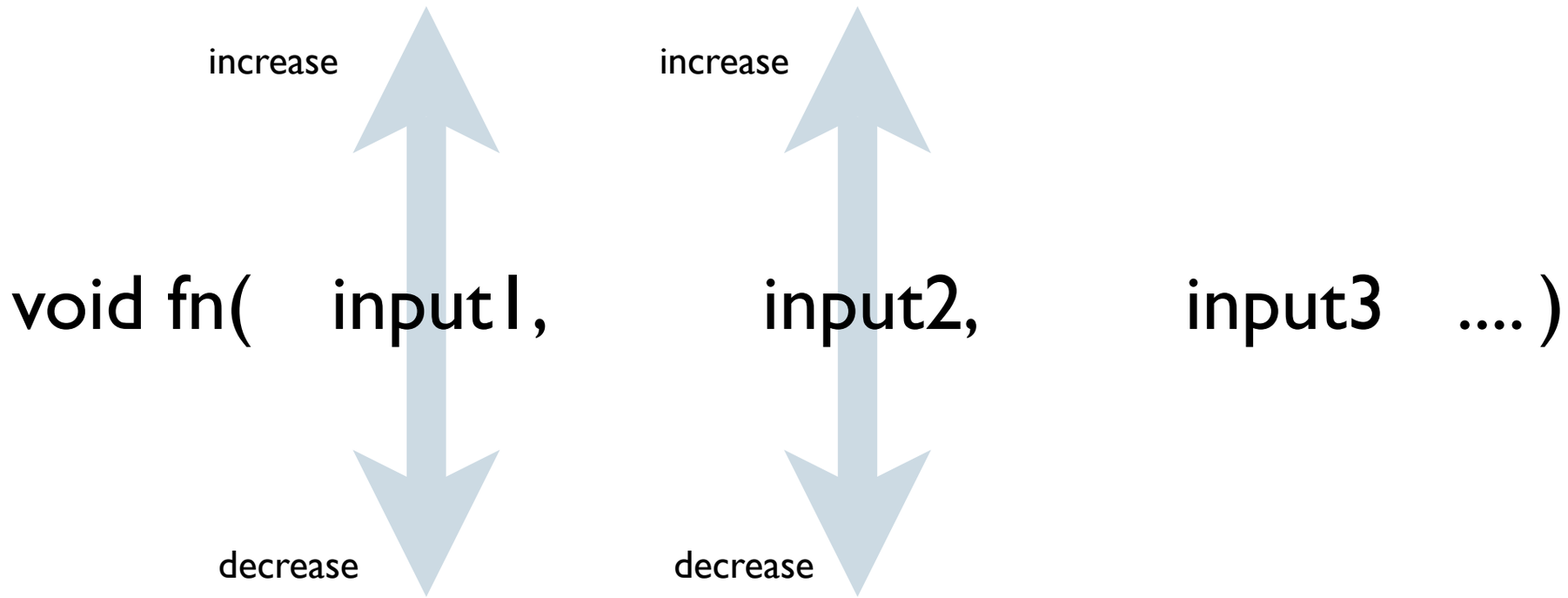
‘Probe’ moves



void fn(input1, input2, input3 ....)

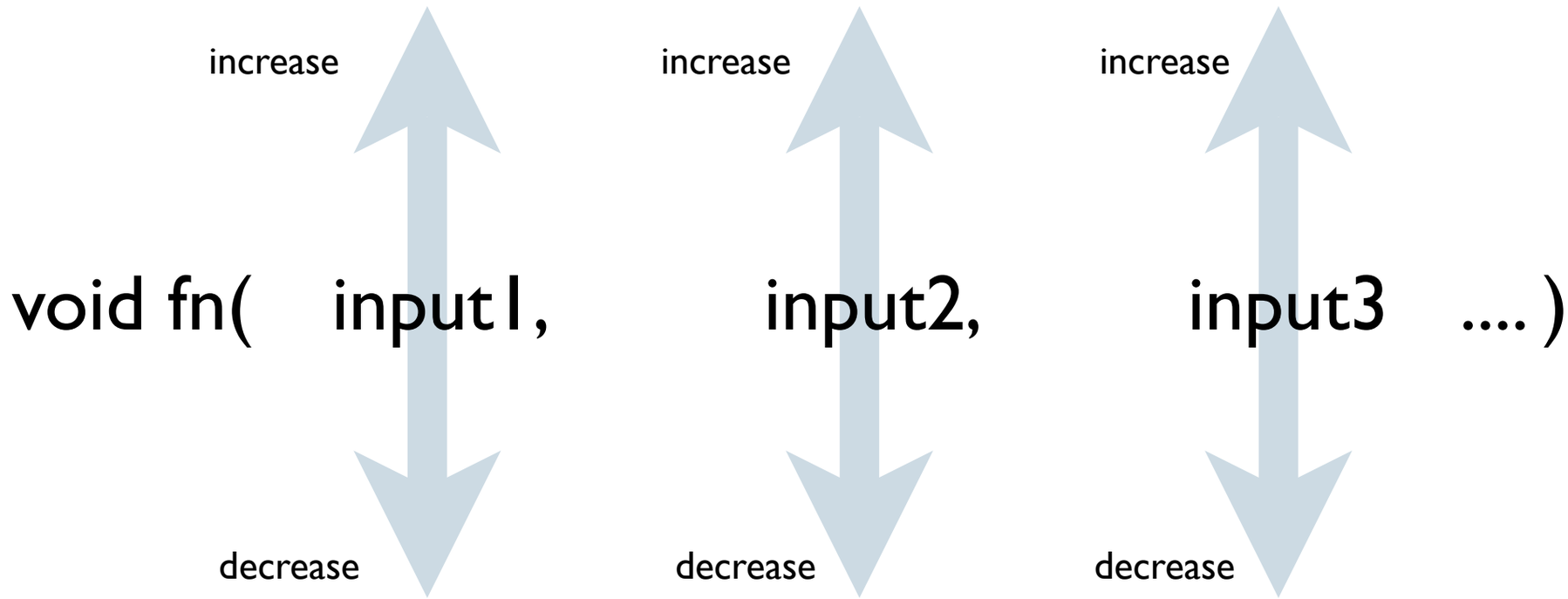
Alternating Variable Method

‘Probe’ moves



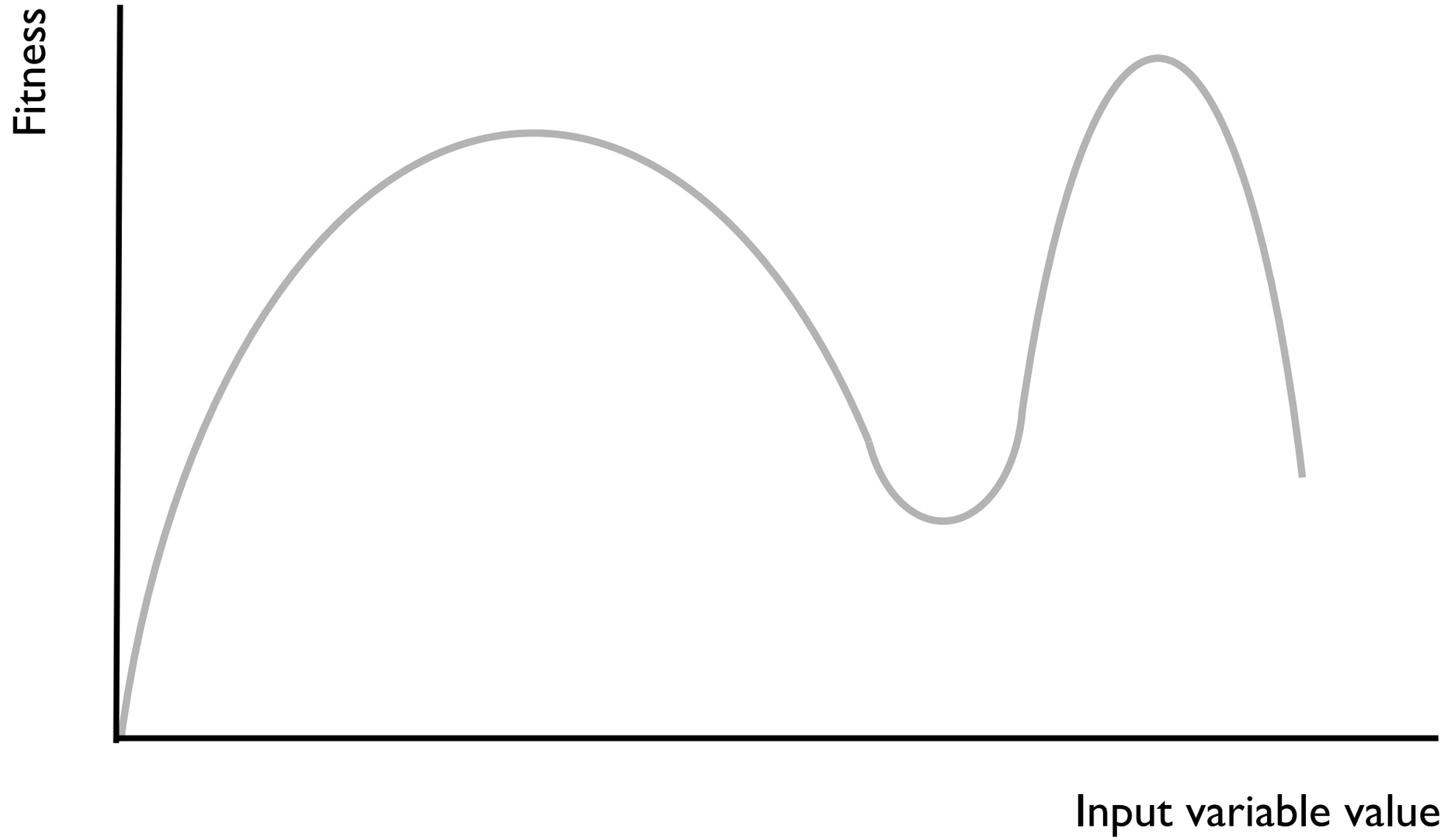
Alternating Variable Method

‘Probe’ moves



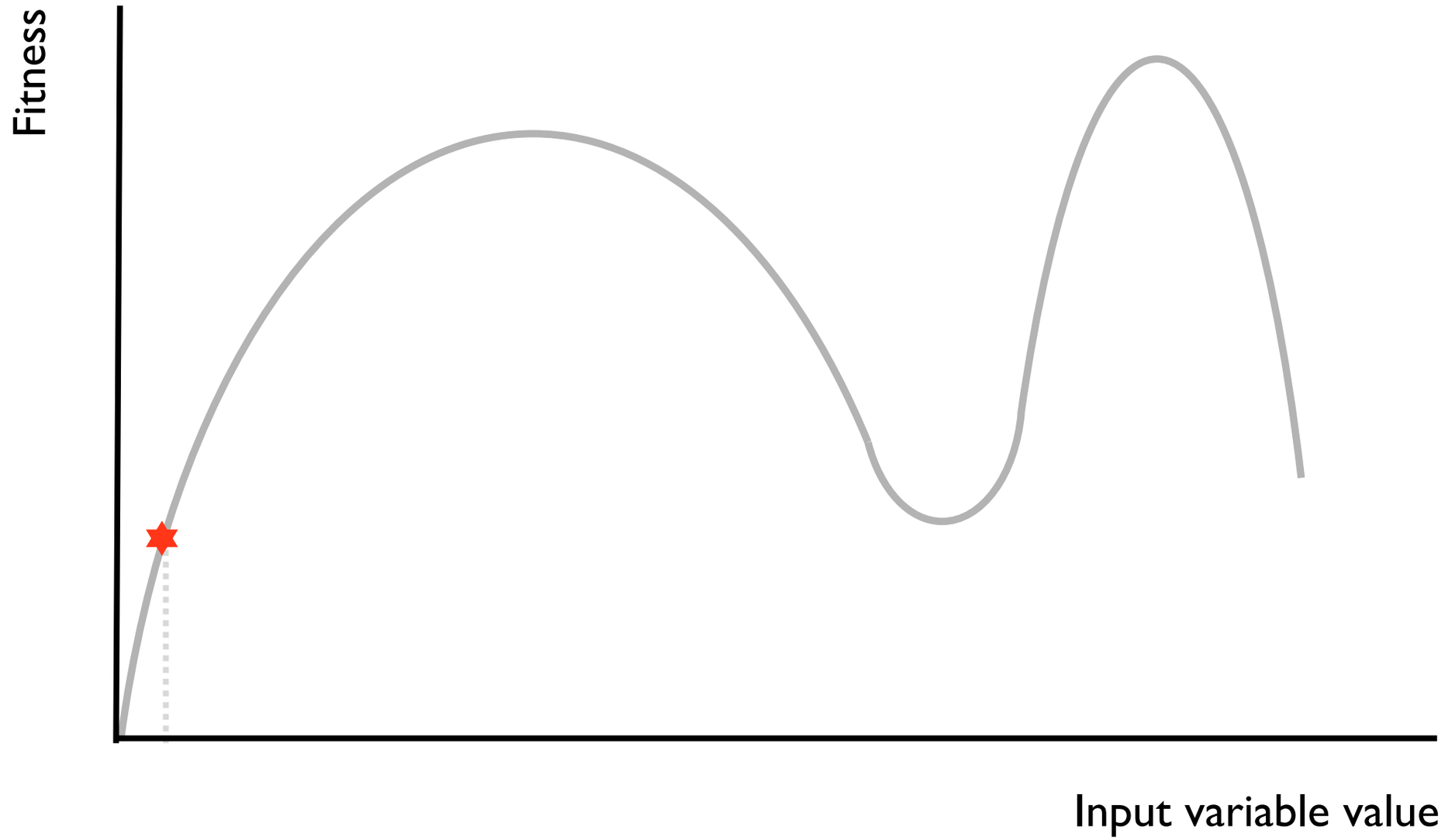
Alternating Variable Method

Accelerated hill climb



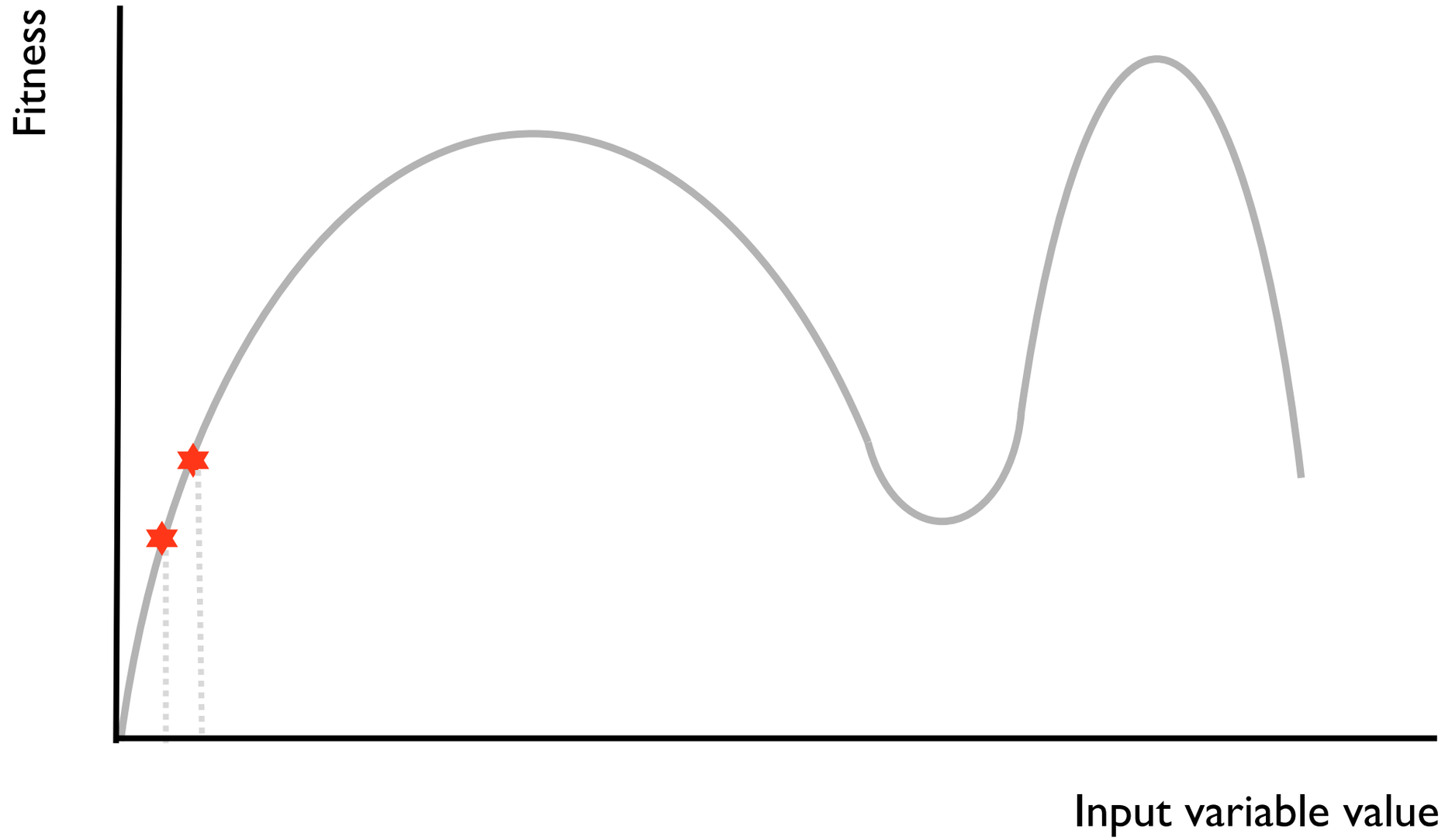
Alternating Variable Method

Accelerated hill climb



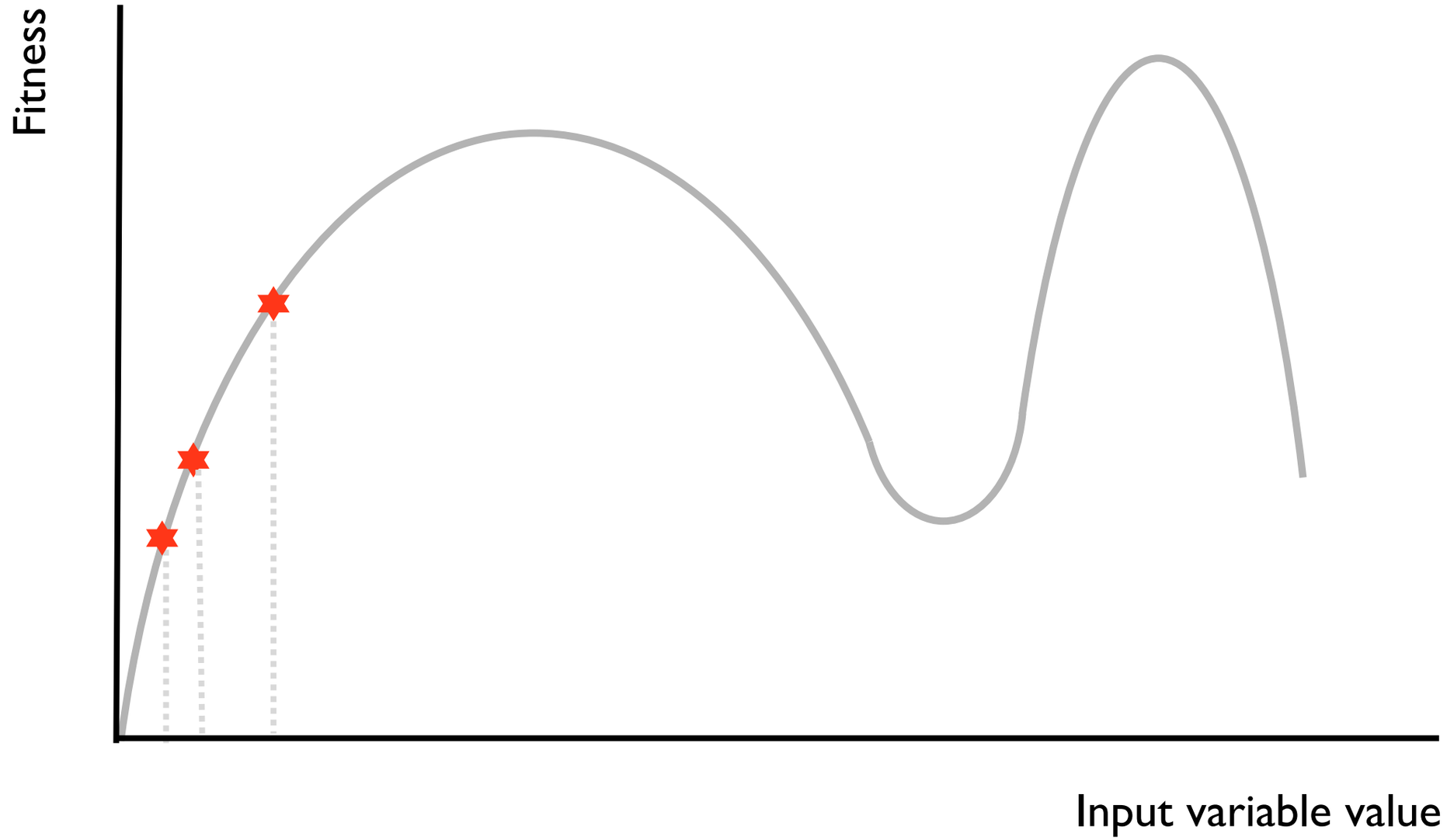
Alternating Variable Method

Accelerated hill climb



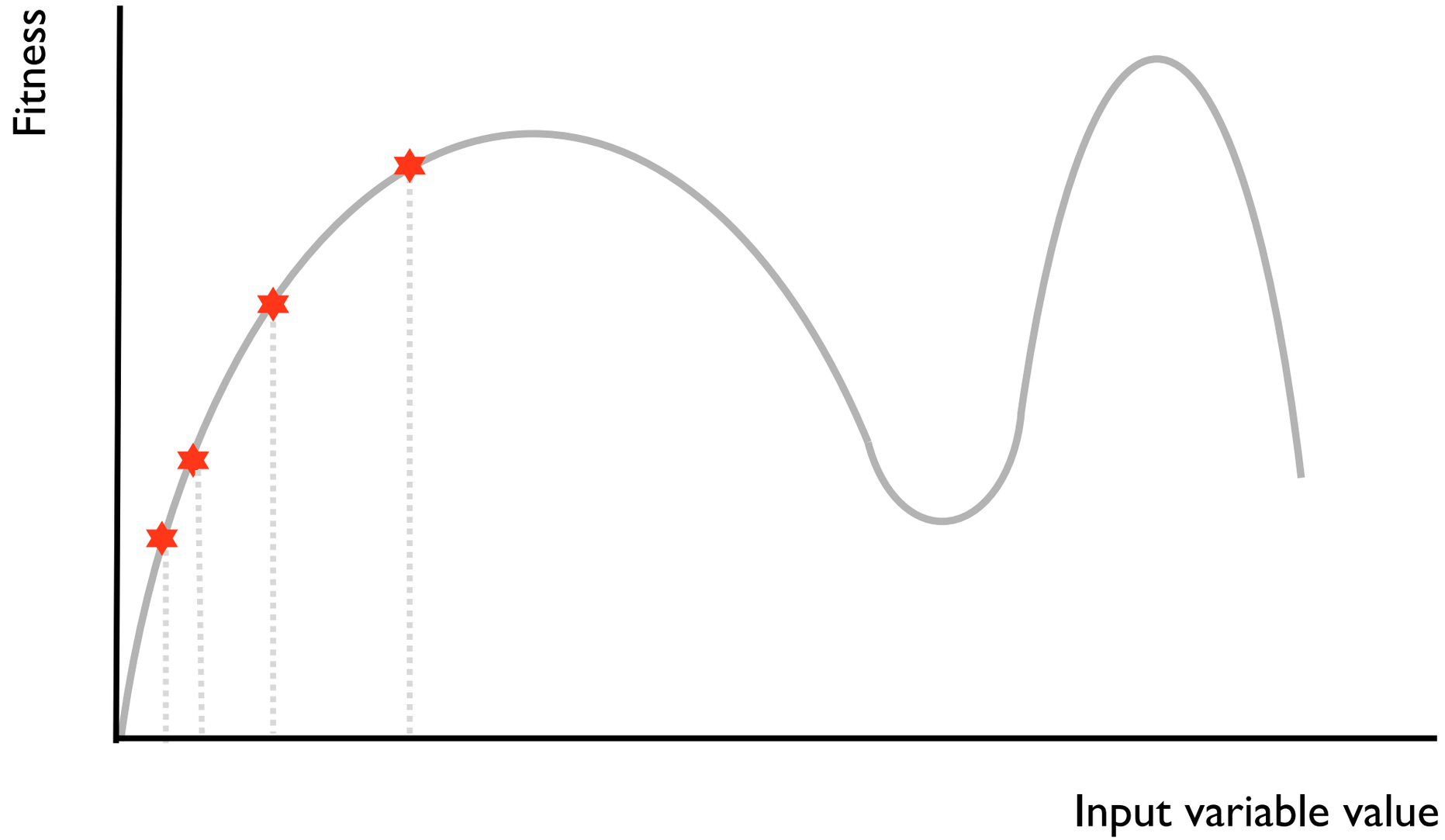
Alternating Variable Method

Accelerated hill climb



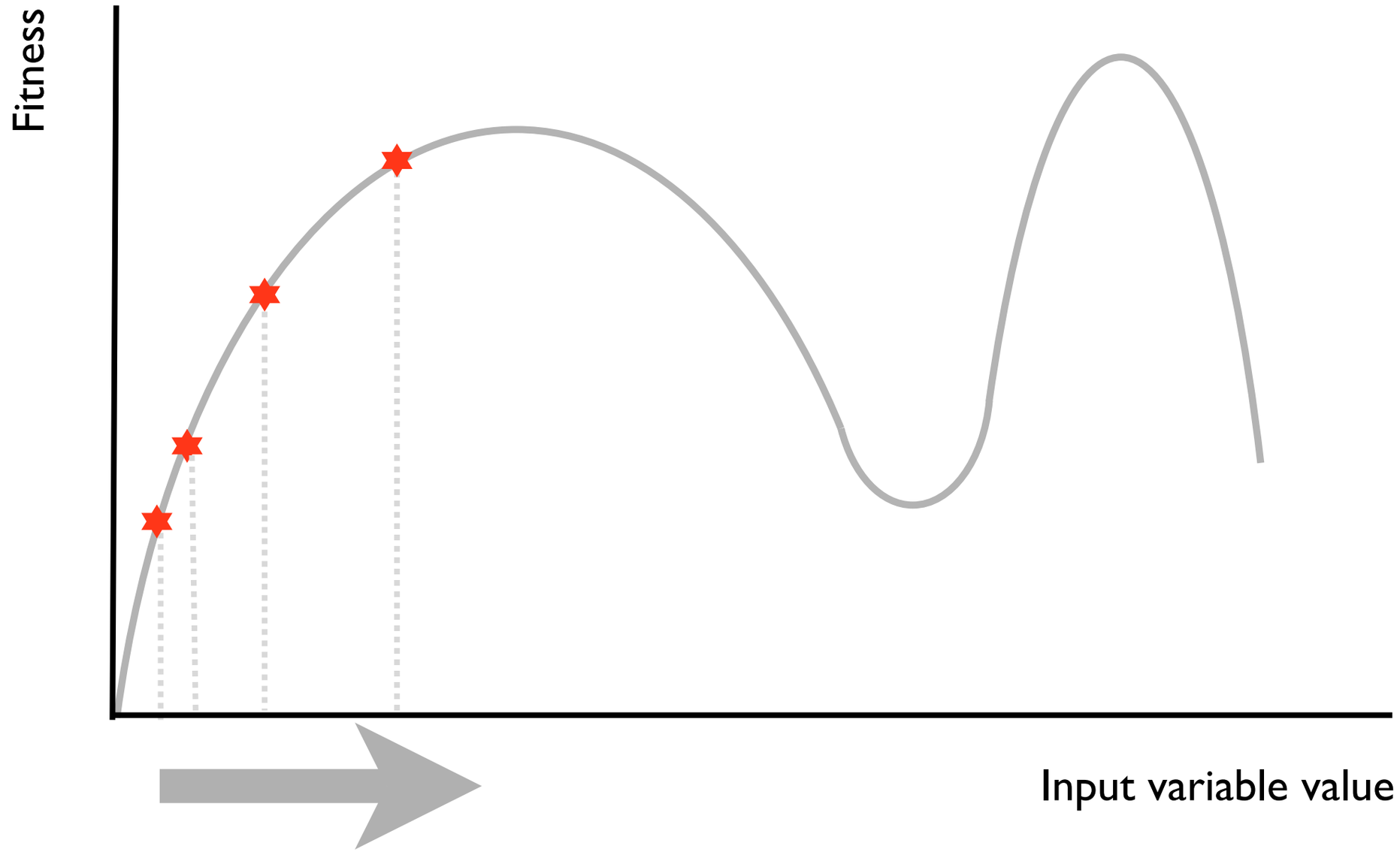
Alternating Variable Method

Accelerated hill climb



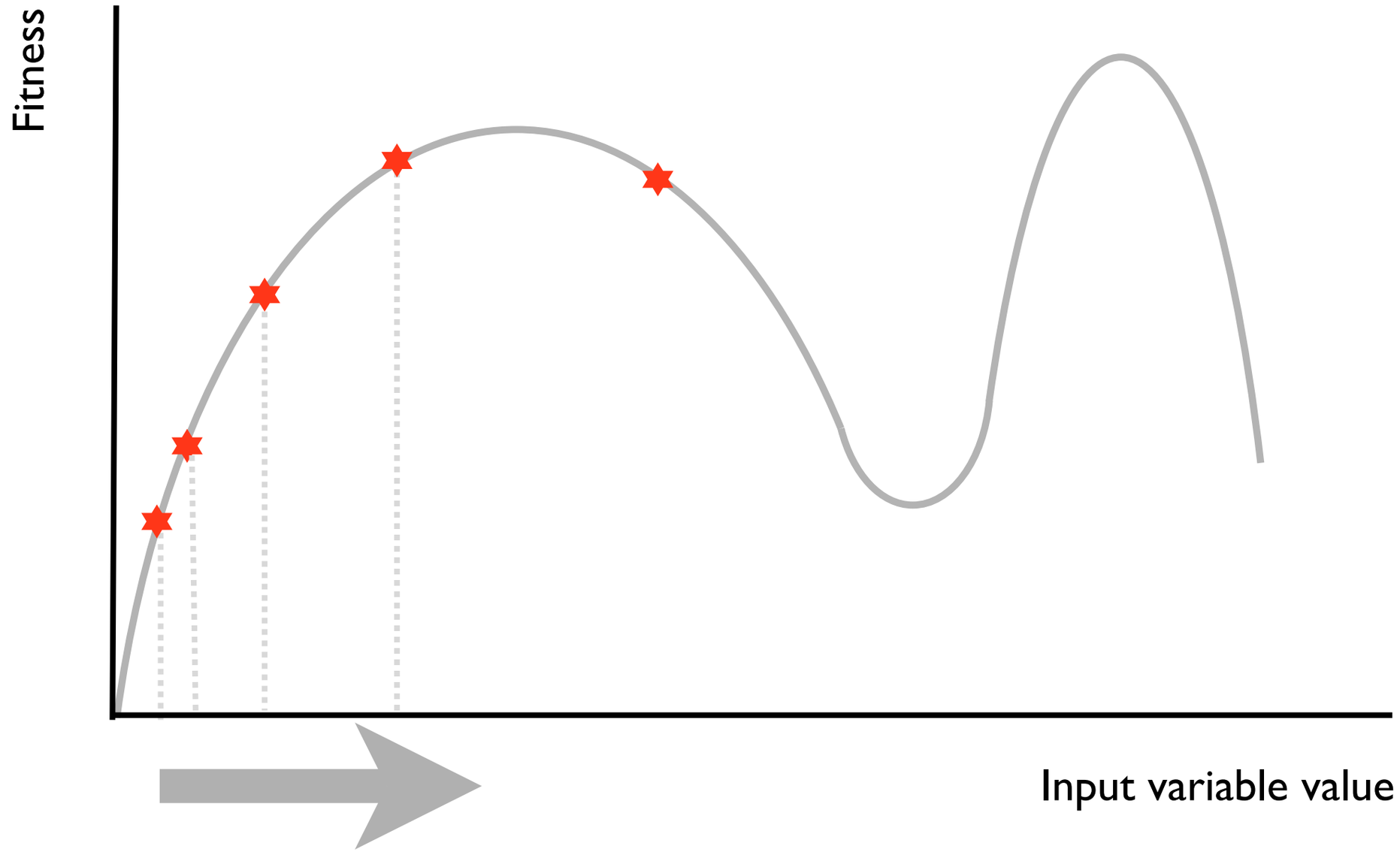
Alternating Variable Method

Accelerated hill climb



Alternating Variable Method

Accelerated hill climb



Alternating Variable Method

1. Randomly generate start point

a=10, b=20, c=30

```
void example(int a, int b, ...) {

    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```

Alternating Variable Method

1. Randomly generate start point

   a=10, b=20, c=30

2. 'Probe' moves on a

   a=9, b=20, c=30

   a=11, b=20, c=30


```
void example(int a, int b, ...) {

    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```

Alternating Variable Method

1. Randomly generate start point

   a=10, b=20, c=30

2. 'Probe' moves on a

   a=9, b=20, c=30

   a=11, b=20, c=30

   no effect

3. 'Probe' moves on b

   a=10, b=19, c=30

   improved branch distance

```
void example(int a, int b, ...) {

    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```

Alternating Variable Method

1. Randomly generate start point

   a=10, b=20, c=30

2. 'Probe' moves on a

   a=9, b=20, c=30

   a=11, b=20, c=30

   no effect

3. 'Probe' moves on b

   a=10, b=19, c=30

   improved branch distance

4. Accelerated moves in direction of improvement

```
void example(int a, int b, ...) {

    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```

A search-based test data generator tool

IGUANA

Input Generation Using Automated Novel Algorithms

IGUANA
(Java)

Test object

(C code compiled to a DLL)

IGUANA
(Java)

inputs

Test object

(C code compiled to a DLL)

IGUANA
(Java)

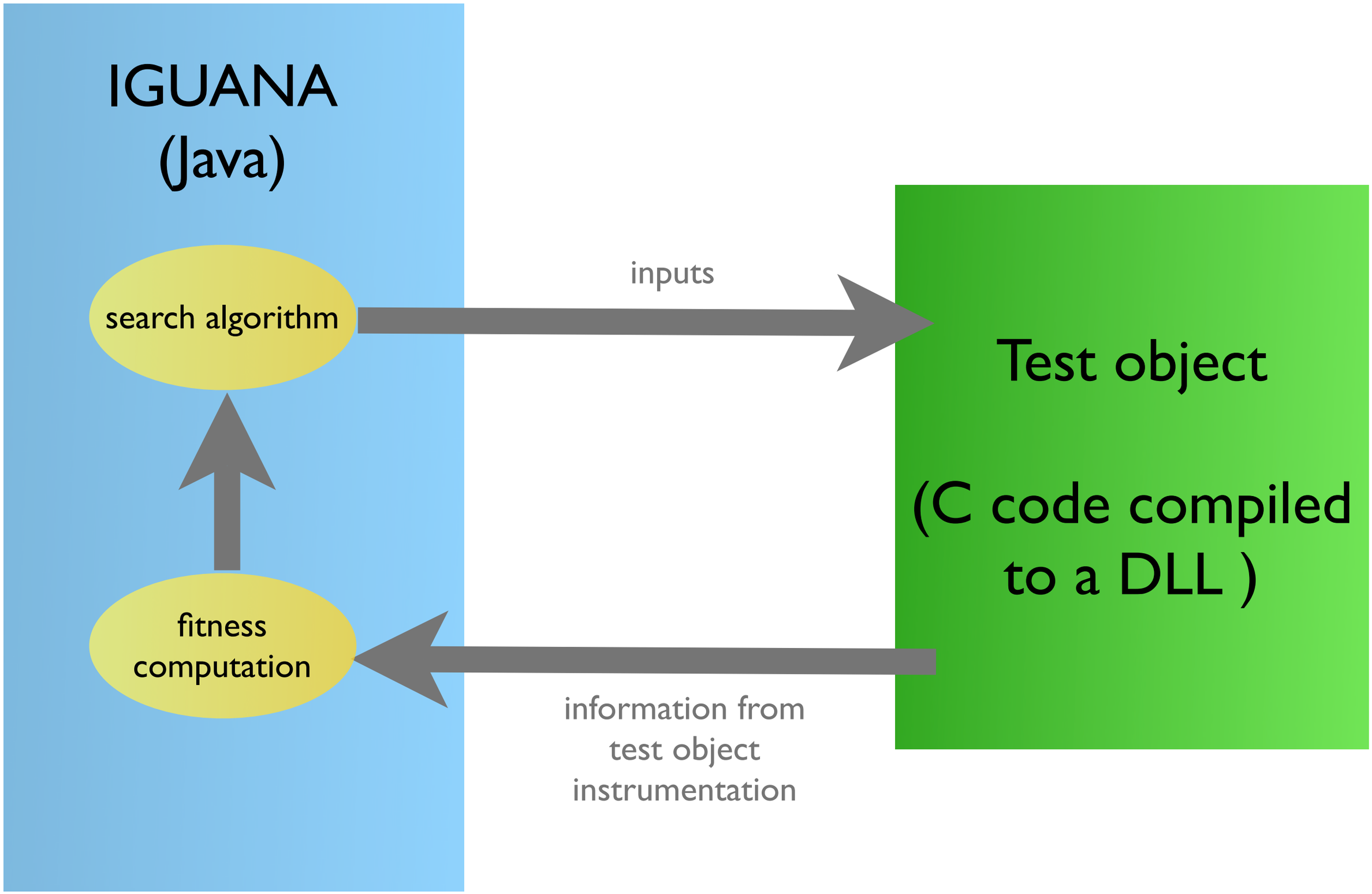
inputs

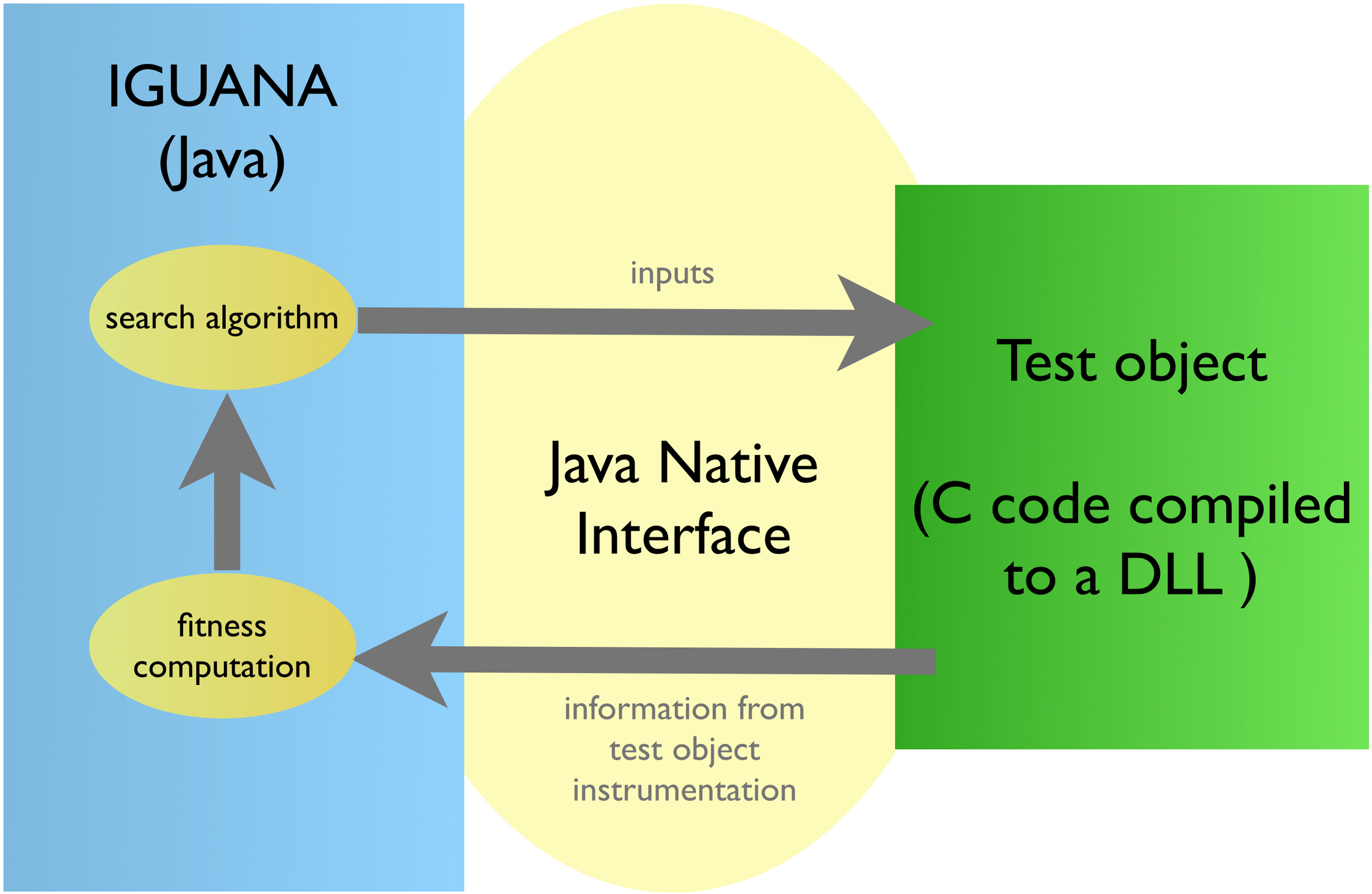
Test object

(C code compiled to a DLL)

fitness computation

information from test object instrumentation


IGUANA
(Java)
search algorithm
fitness
computation
inputs
information from
test object
instrumentation
Test object
(C code compiled
to a DLL)


IGUANA
(Java)
search algorithm
fitness
computation
inputs
Java Native
Interface
information from
test object
instrumentation
Test object
(C code compiled
to a DLL)

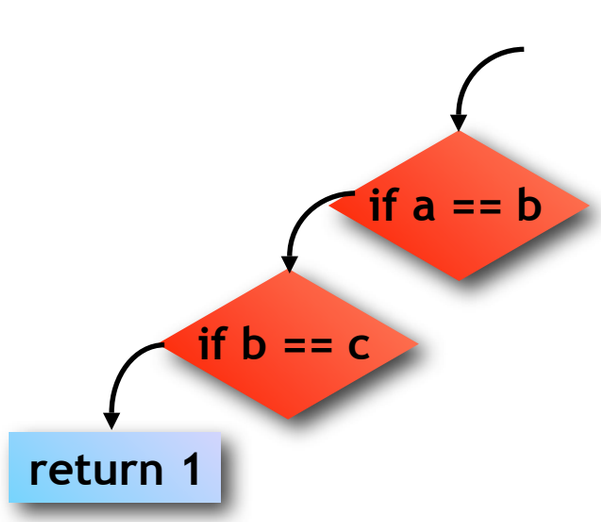
A function for testing

```c
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```

Test Object Preparation

1. Parse the code and extract control dependency graph

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```

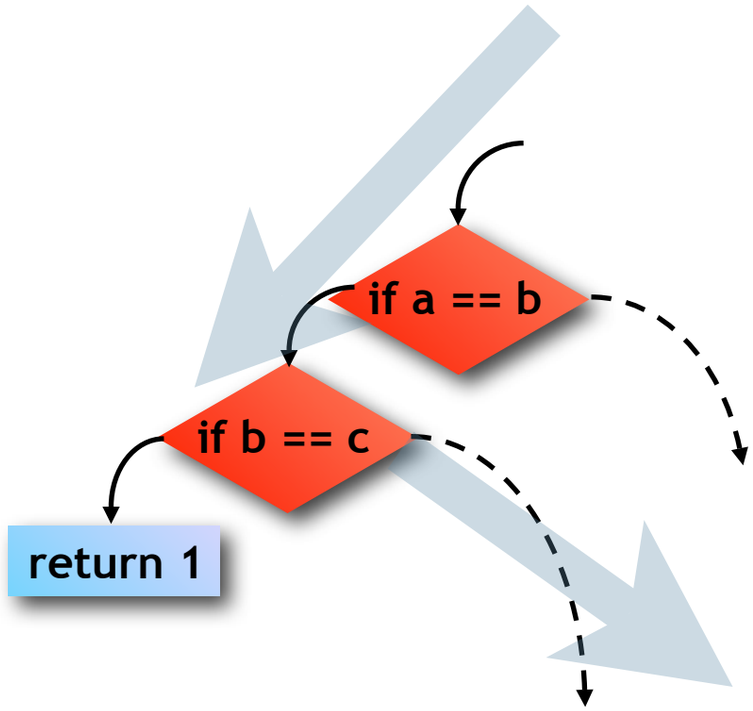


"which decisions are key for the execution of individual structural targets" ?

Test Object Preparation

2. Instrument the code

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```

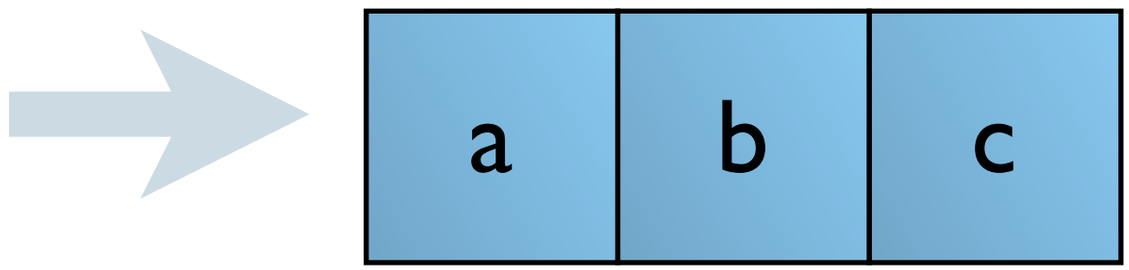


for monitoring control flow and variable values in predicates

Test Object Preparation

3. Map inputs to a vector

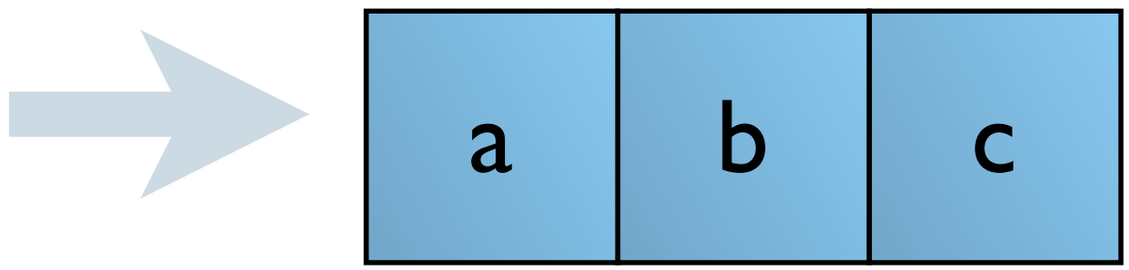
```
int test_me(int a,
            int b,
            int c)
{
  . . .
```



Test Object Preparation

3. Map inputs to a vector

```
int test_me(int a,
            int b,
            int c)
{
  . . .
```

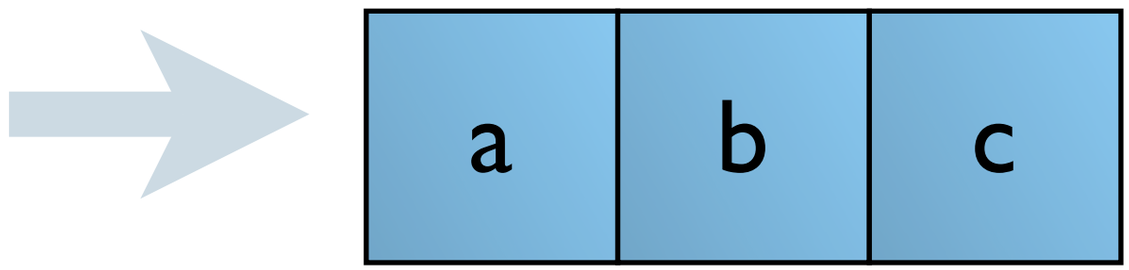


Straightforward in many cases

Test Object Preparation

3. Map inputs to a vector

```
int test_me(int a,
            int b,
            int c)
{
  . . .
```



Straightforward in many cases

Inputs composed of dynamic data structures are harder to compose

Kiran Lakhotia, Mark Harman and Phil McMinn.
Handling Dynamic Data Structures in Search-Based Testing.
Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008), Atlanta, Georgia, USA, July 12-16 2008, pp. 1759-1766, ACM Press.

Instrumentation

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```



```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

Instrumentation

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (a == b) {
        if (b == c) {
            return 1;
        }
    }

    if (a == 0) {
        flag = 1;
    }

    if (flag && b == 0) {
        return 2;
    }

    return -1;
}
```

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

Each branching condition is replaced by a call to the function node(...)

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

the instrumentation should only observe the program and not alter its behaviour

The first parameter is the control flow graph node ID of the decision statement

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

The second parameter is a boolean condition that replicates the structure in the original program

(i.e. including short-circuiting)

```c
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

Relational predicates are replaced with functions that compute branch distance.

The instrumentation tells us:

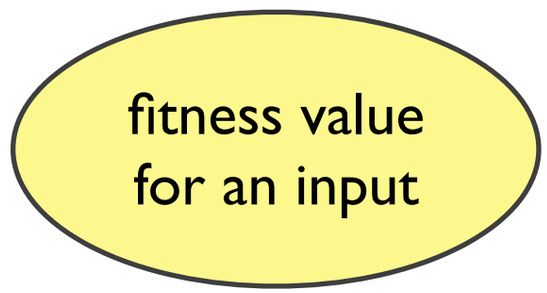
Which decision nodes were executed

and their outcome (branch distances)

Therefore we can find which decision control flow diverged from a target for an input....

... and compute the approach level from the control dependence graph

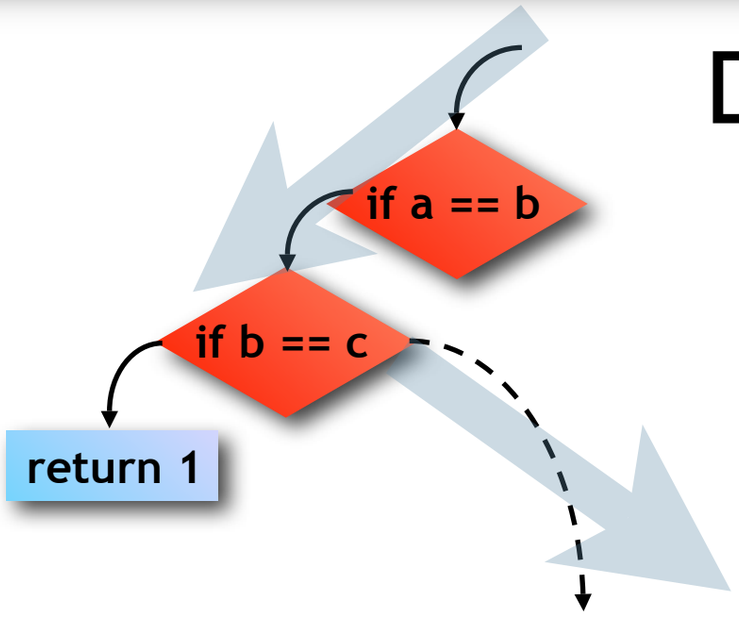
... and lookup the branch distance



Input: <20, 20, 30>

```
int test_me(int a, int b, int c)
{
    int flag = 0;

    if (node(1, equals(0, a, b))) {
        if (node(2, equals(0, b, c))) {
            return 1;
        }
    }

    if (node(4, equals(0, a, 0))) {
        flag = 1;
    }

    if (node(6, is_true(0, flag) && equals(1, b, 0))) {
        return 2;
    }

    return -1;
}
```

NODE	T	F
1	0	1
2	10	0
4	20	0
....		



Diverged at node 2

approach level: 0

branch distance: 10

fitness = 0.009945219