Search Based Software Engineering and Evolutionary Testing

Barbara Russo

SwSE - Software and Systems Engineering research group

References

- Phil McMinn Search-Based Software Testing: Past, Present and Future, 2009
- Joachim Wegener, Oliver Bühler Evaluation of Different Fitness Functions for the Evolutionary Testing of an Autonomous Parking System, 2004

- Finding the optimal inputs for testing is a key issue that can be NP-hard
- Search based techniques can be leverage to find good inputs

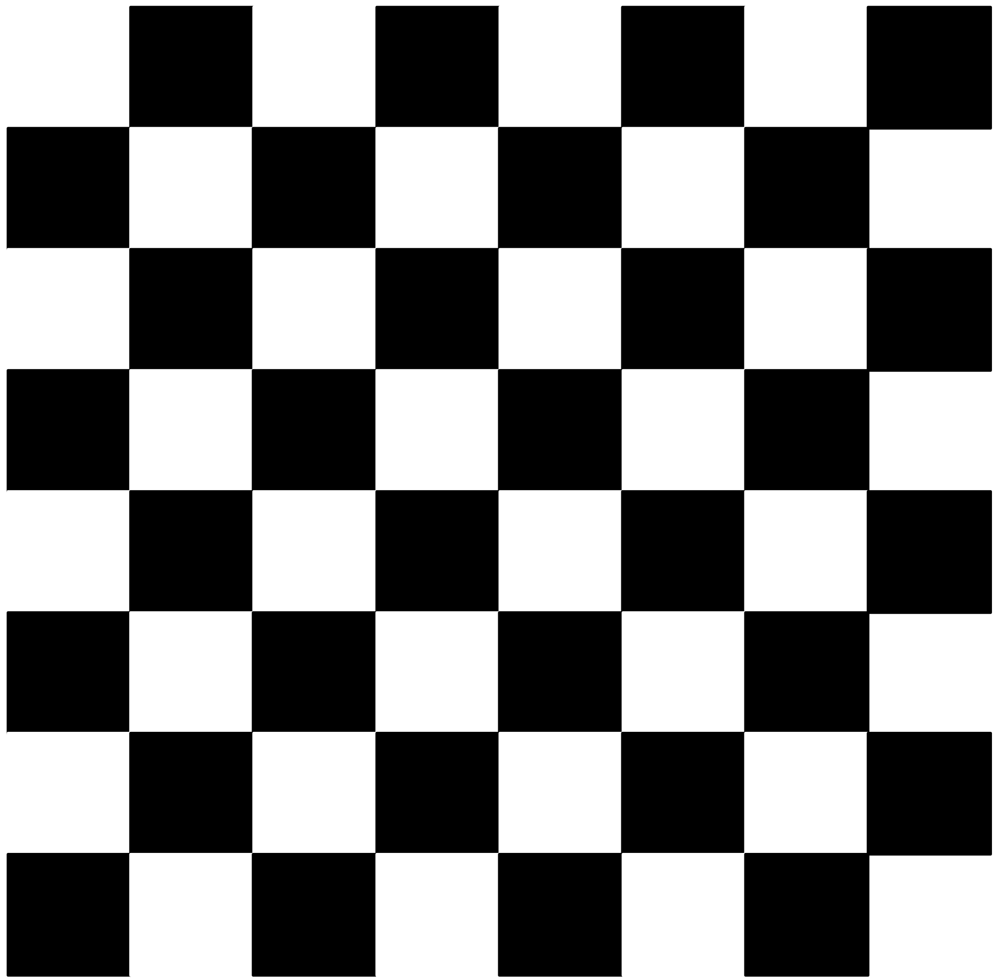
What is SBSE

Techniques to **search large spaces** guided by a **fitness function** that captures **properties** of the software artefacts we seek for

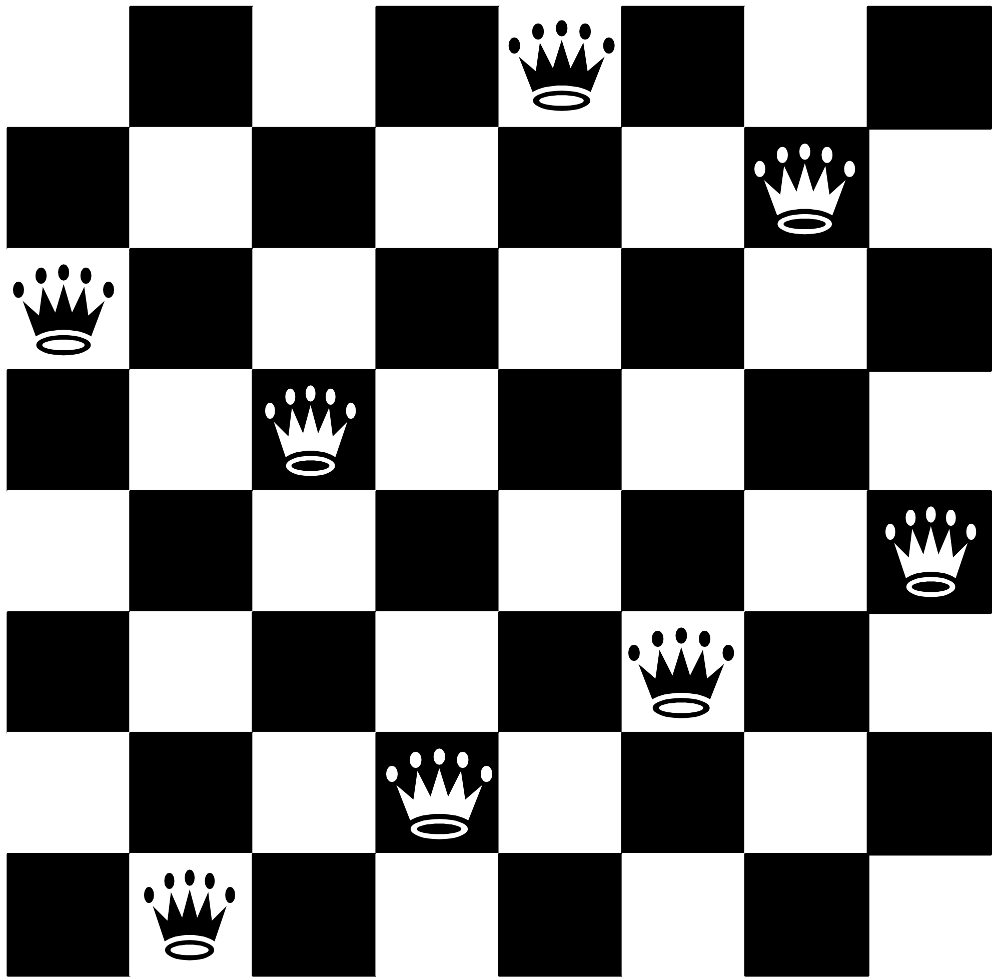
Example - the search

Place n queens on a board so that there is no attack

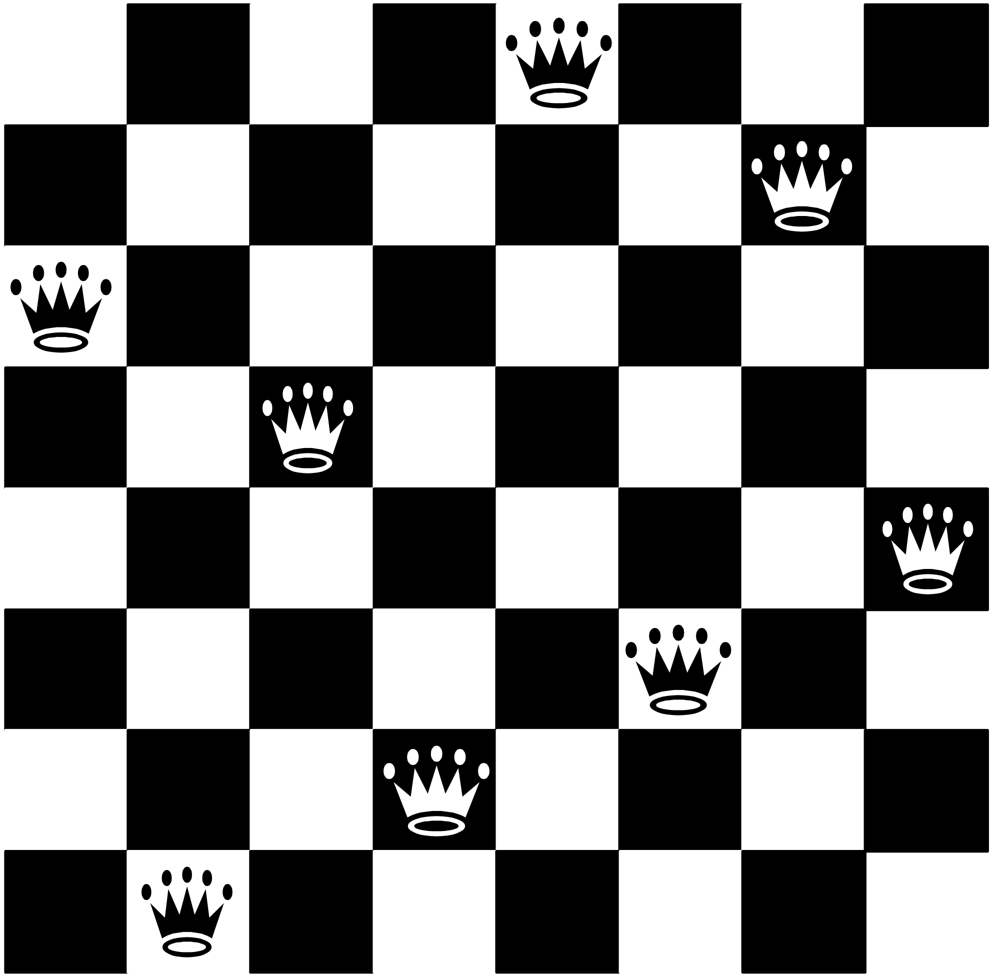
The Eight Queens Problem



The Eight Queens Problem

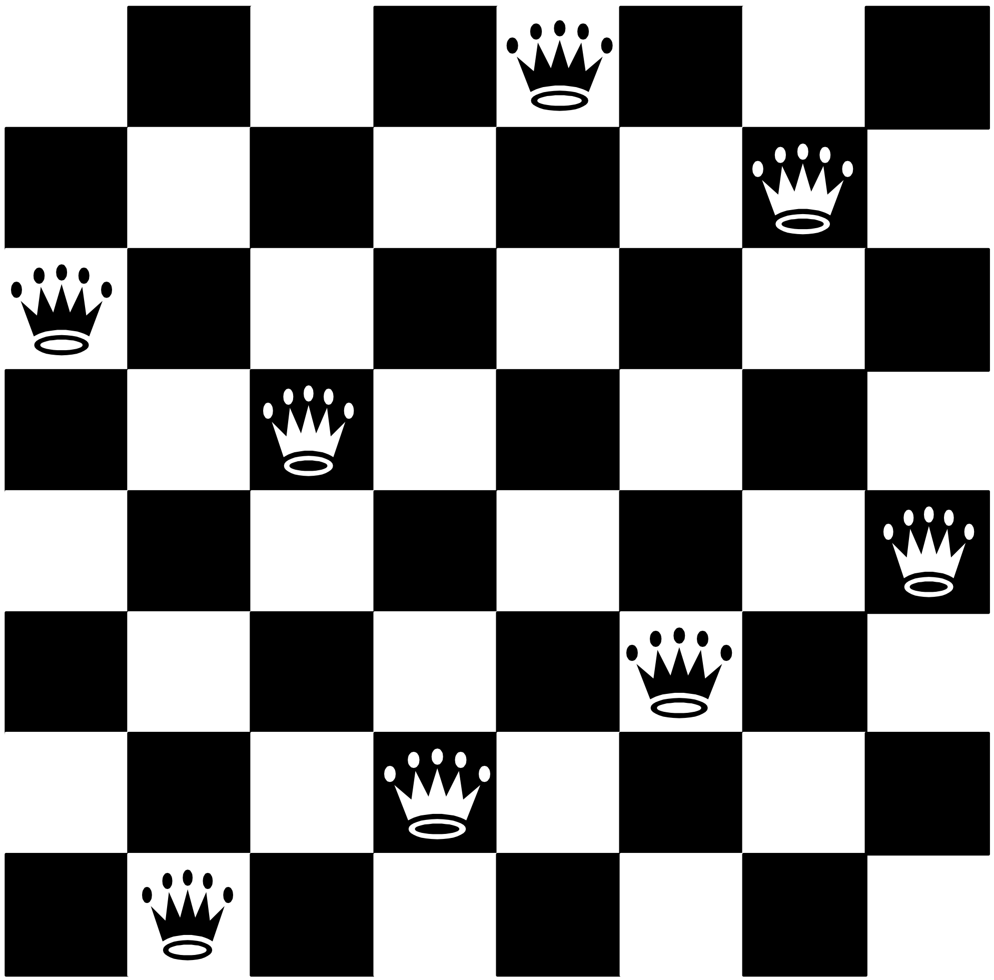


The Eight Queens Problem



Perfect !

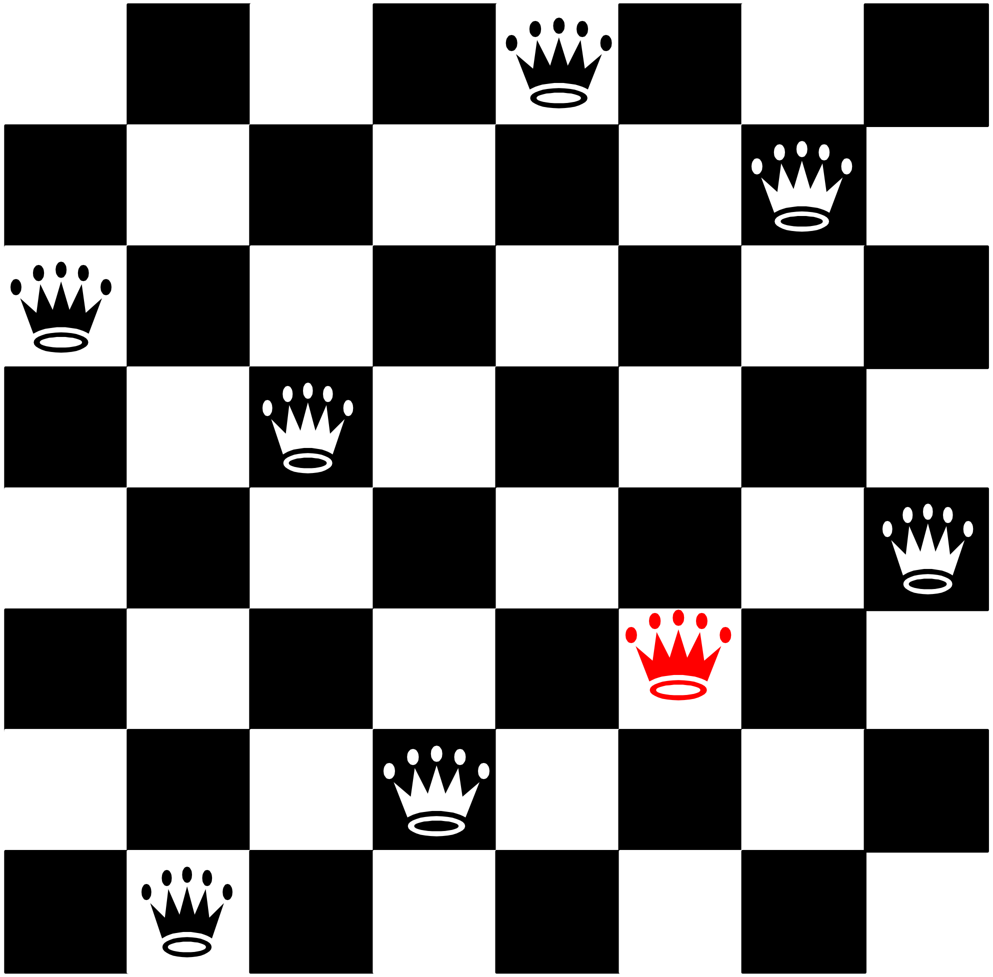
The Eight Queens Problem



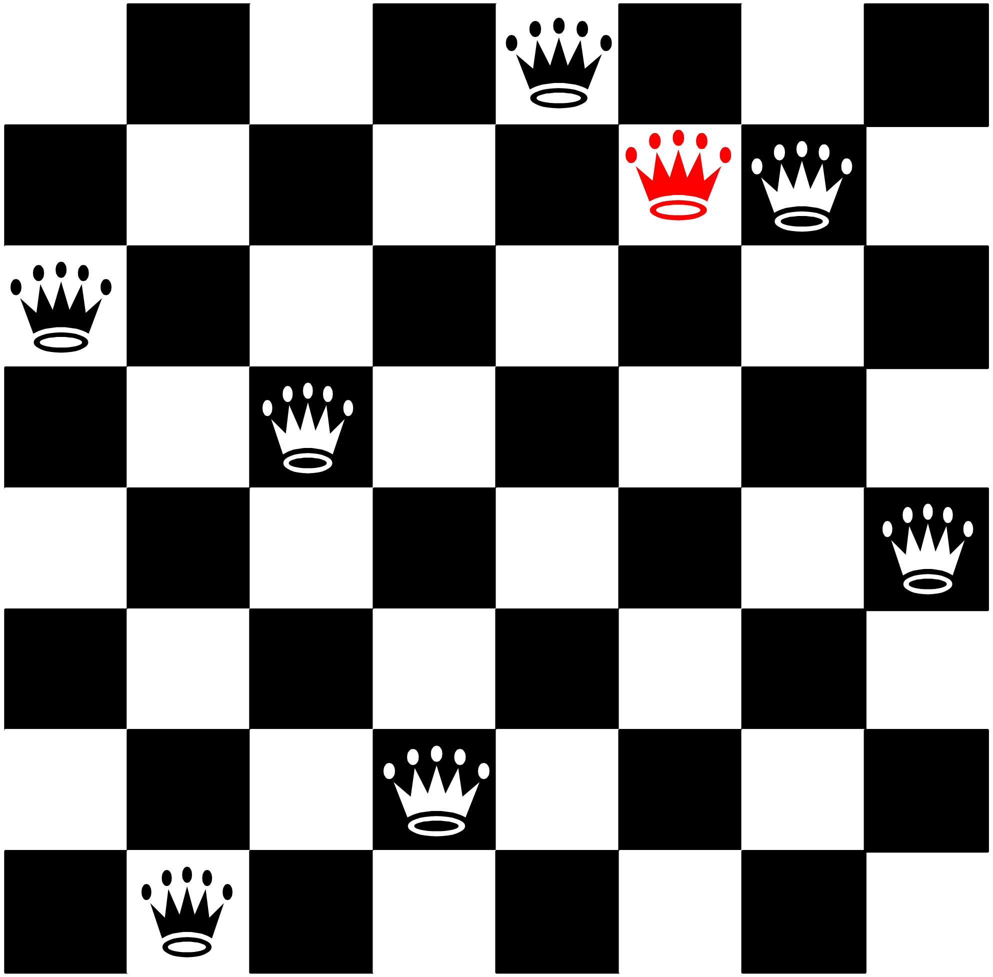
Perfect !

Score 0

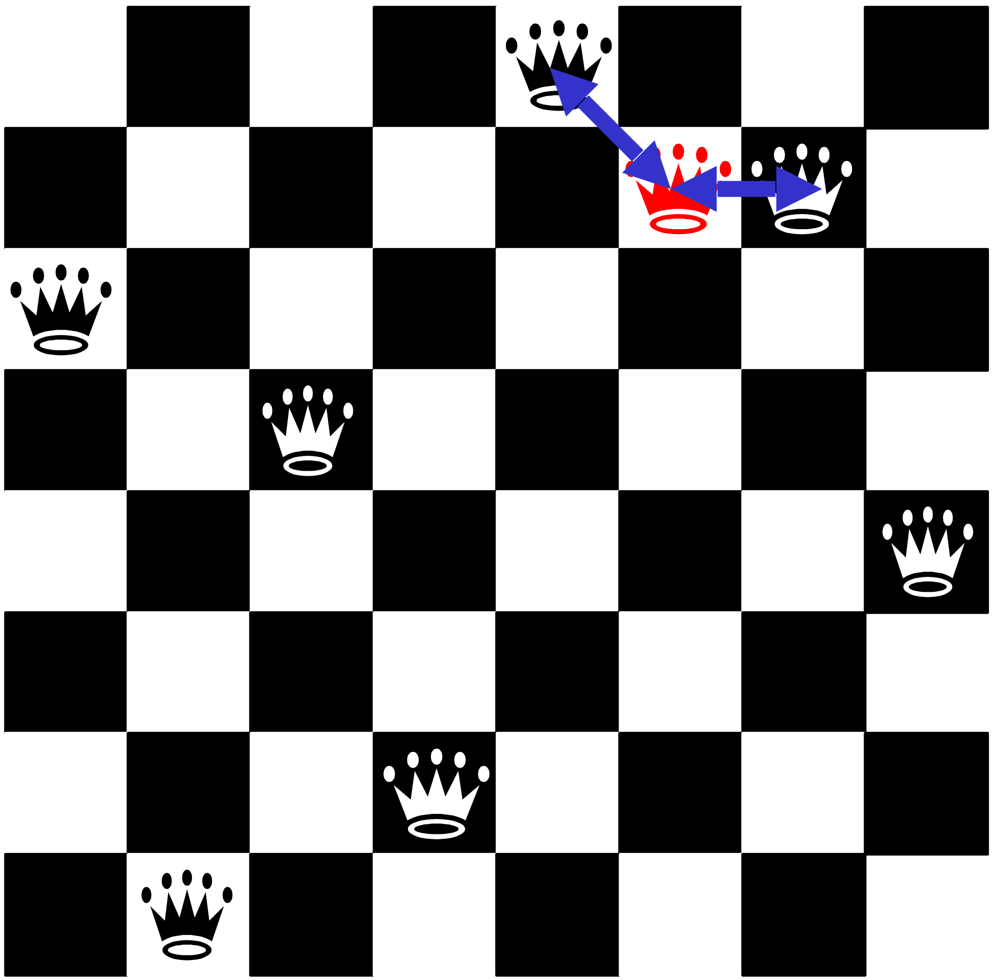
The Eight Queens Problem



The Eight Queens Problem



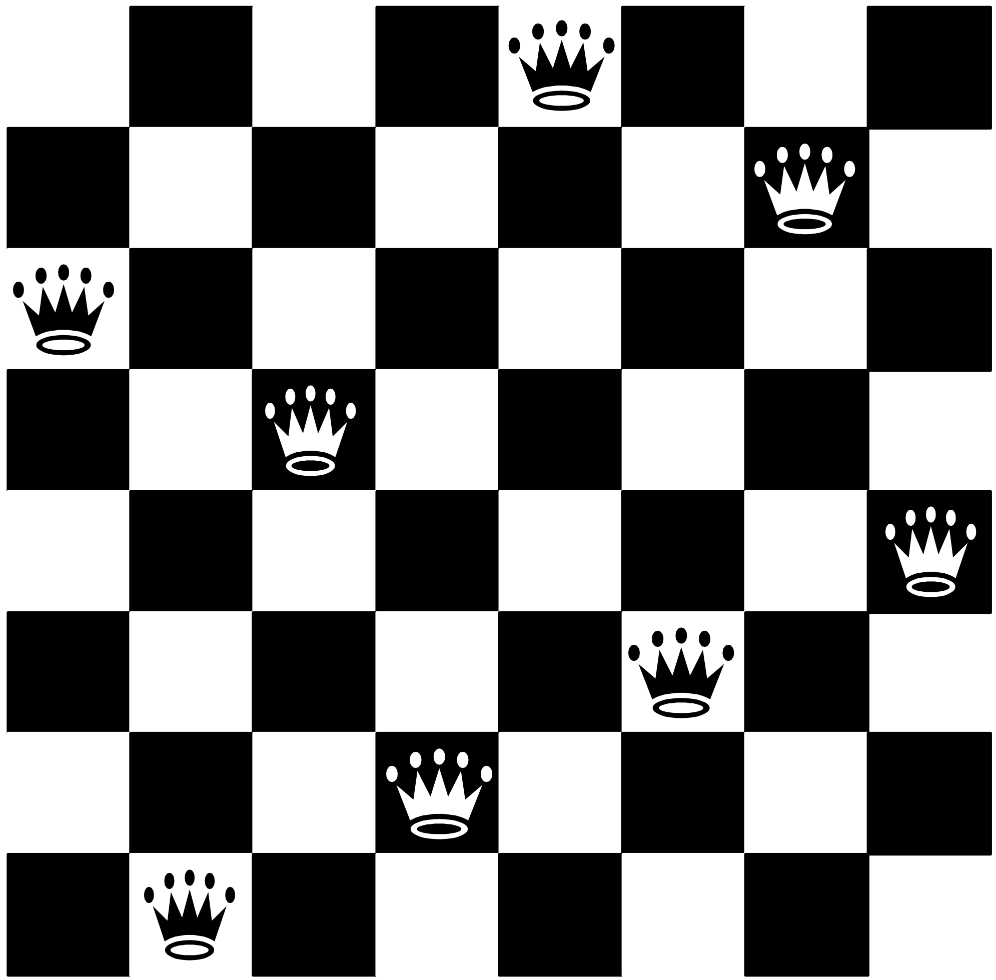
The Eight Queens Problem



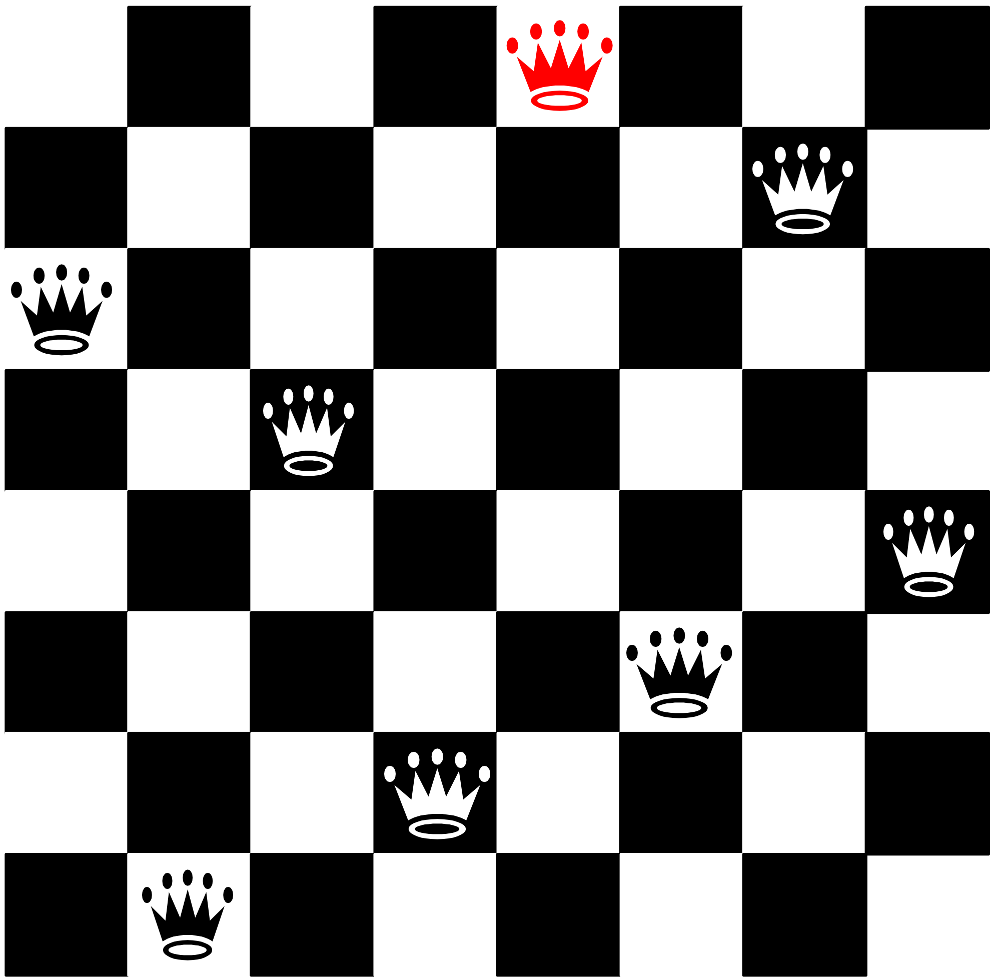
Two Attacks

Score -2

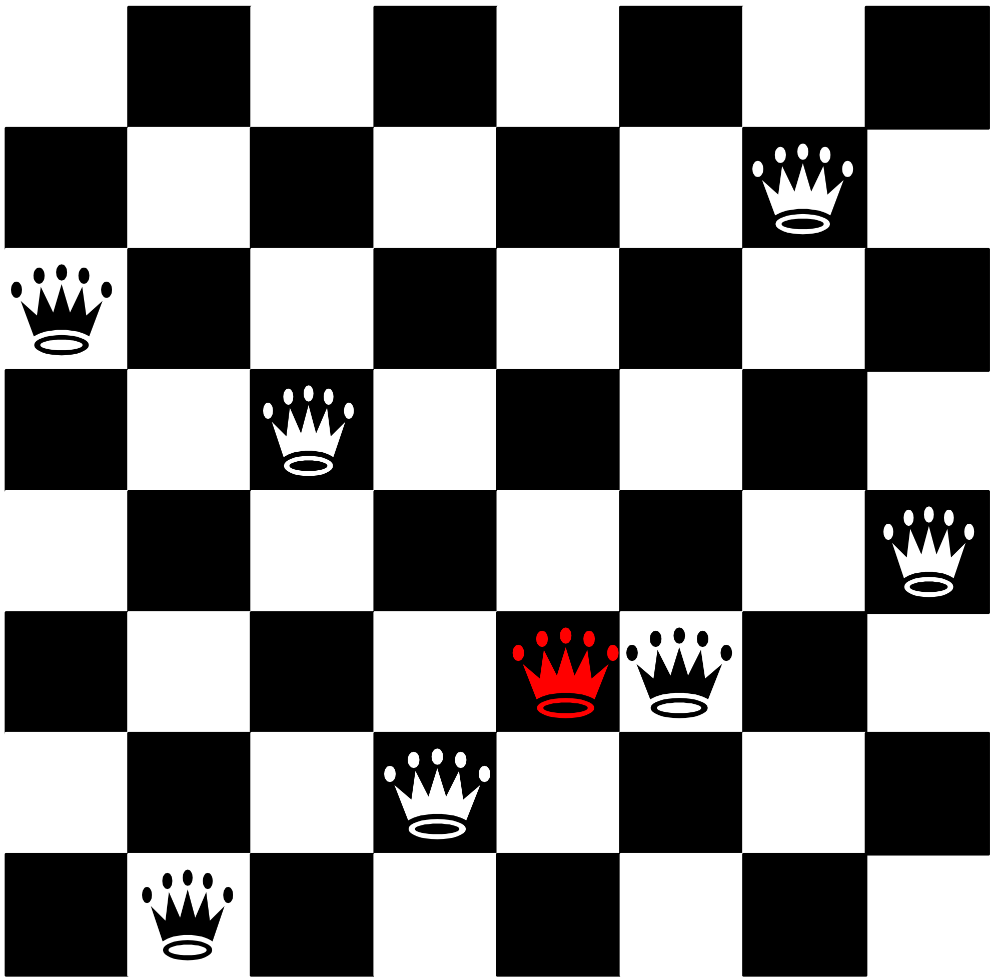
The Eight Queens Problem



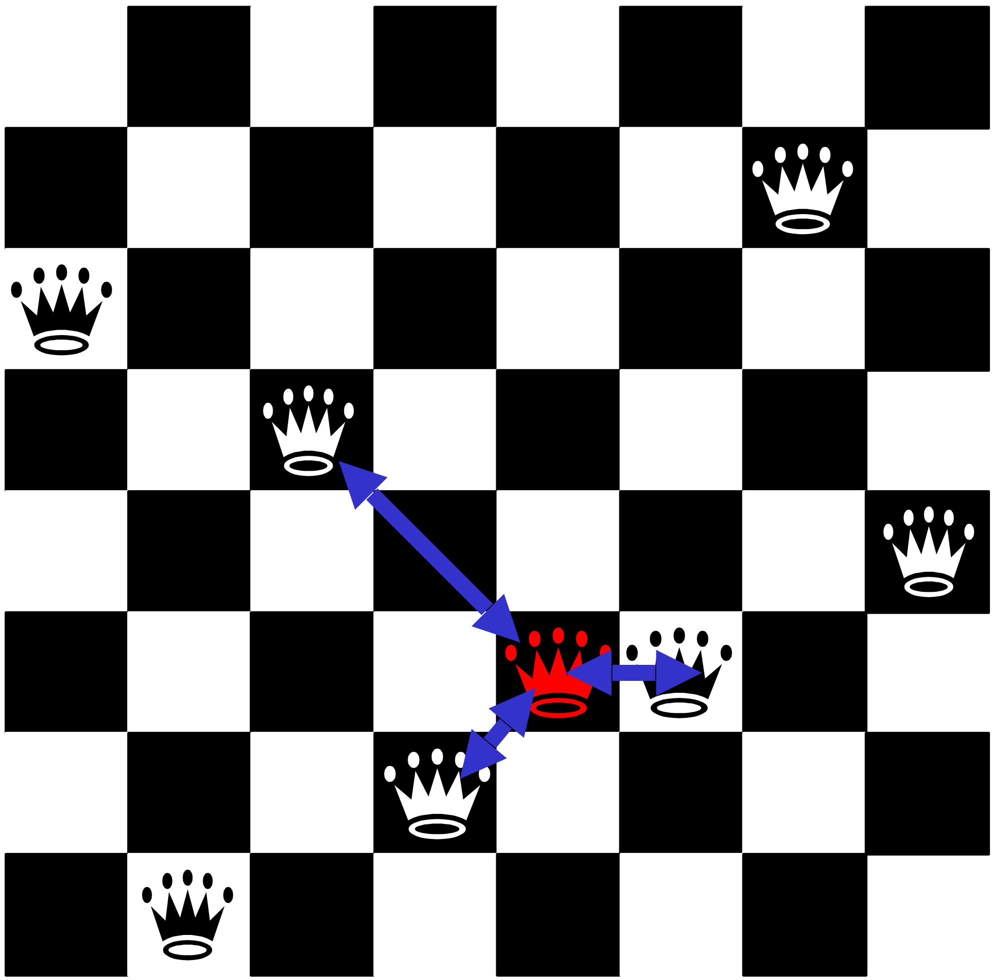
The Eight Queens Problem



The Eight Queens Problem



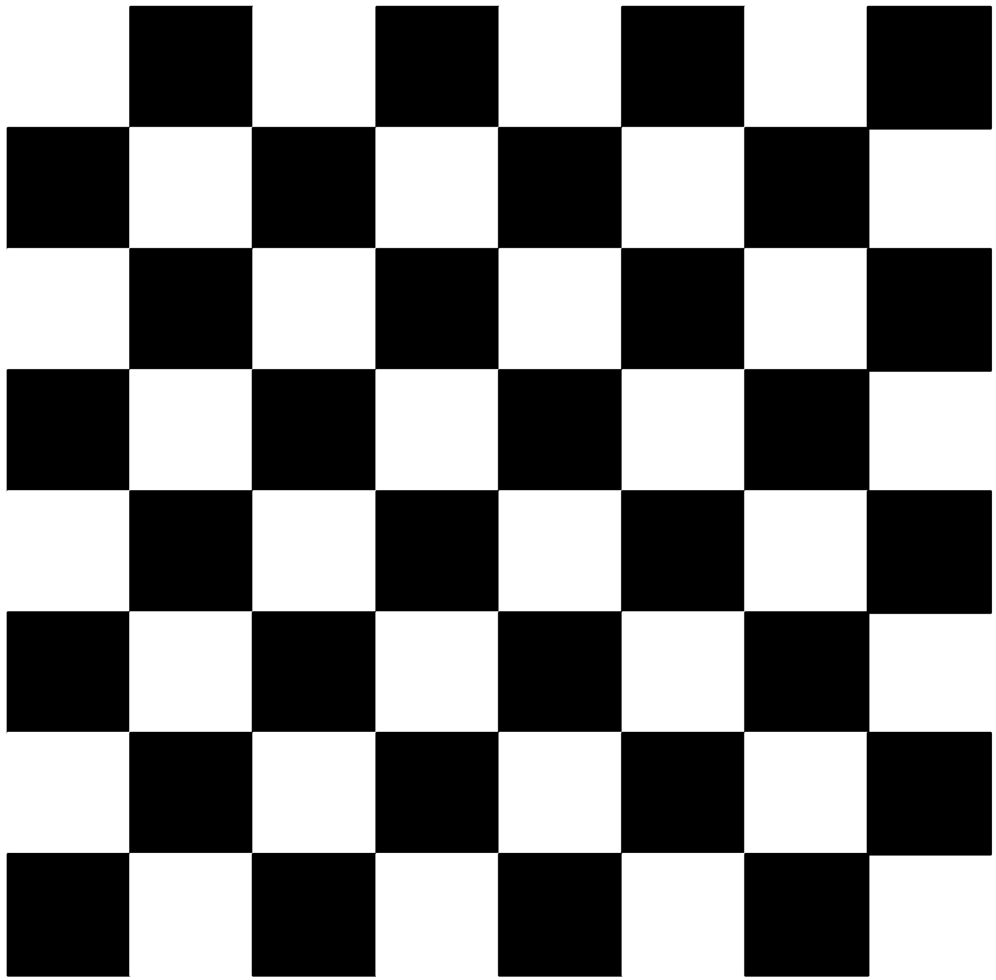
The Eight Queens Problem



Three Attacks

Score -3

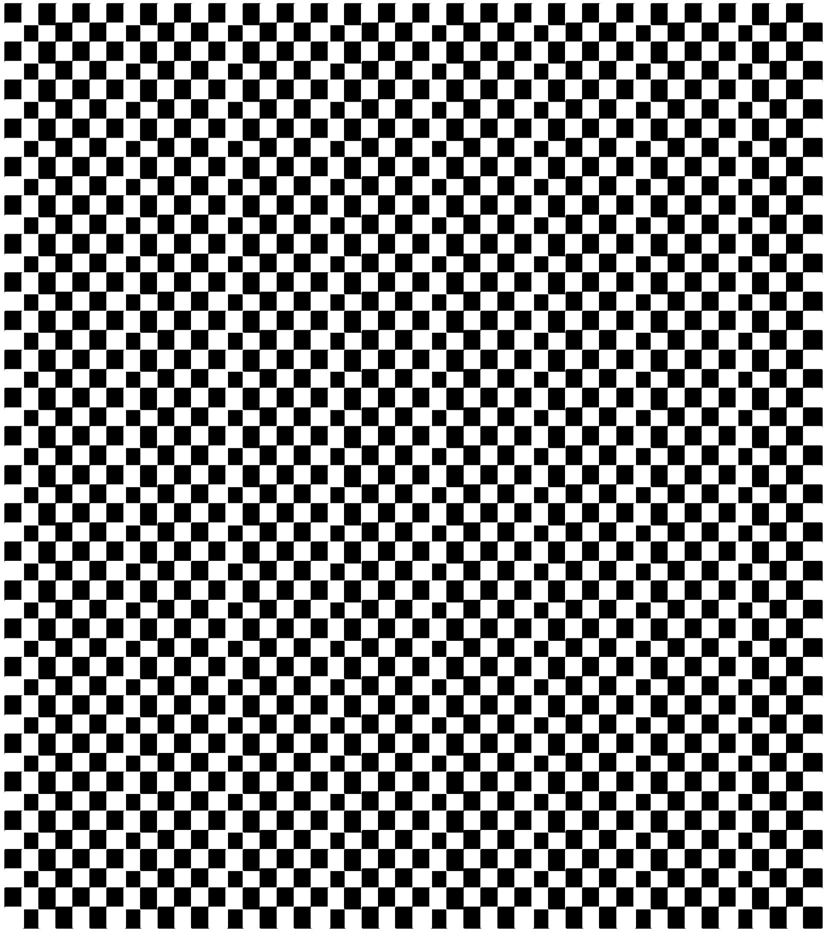
Generate a solution



Place
8 queens
on the
Board …

… so that
there are
no
attacks

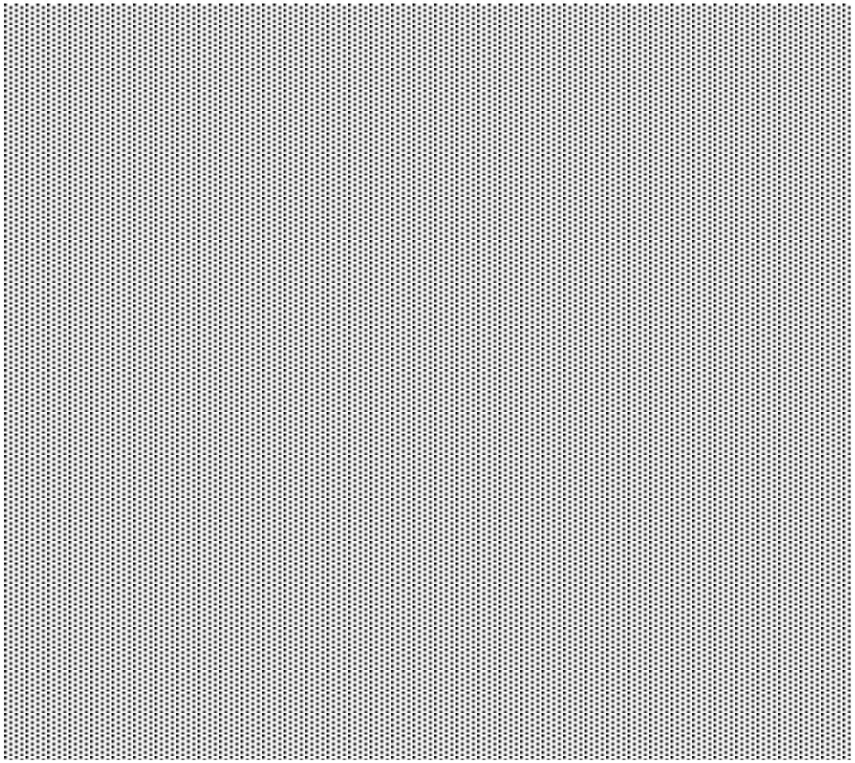
Scale up: Generate a solution



Place
44 queens
on the
Board …

… so that
there are
no
attacks

Scale up: Generate a solution

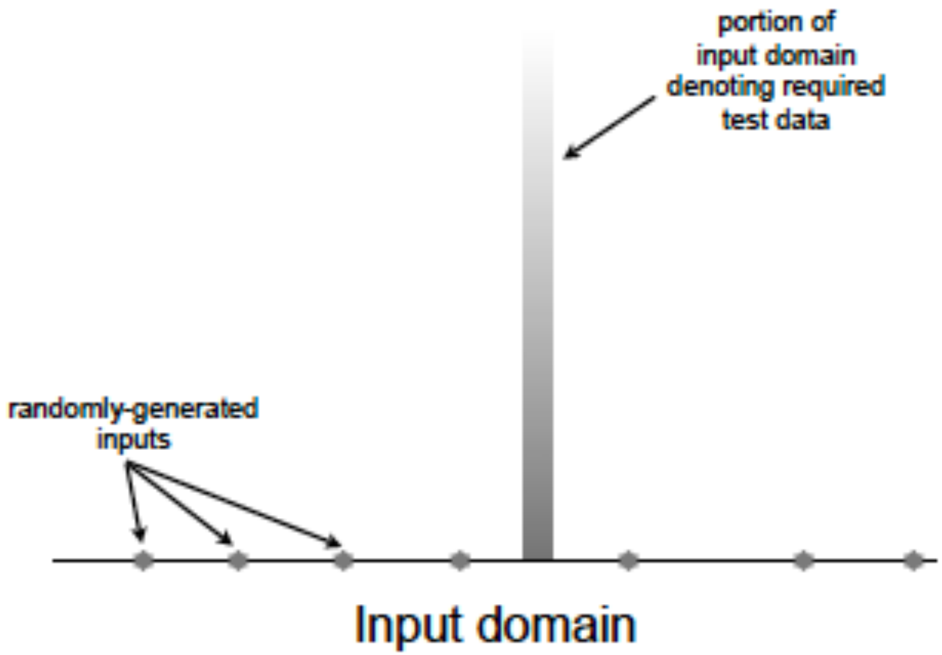


Place
400 queens
on the
Board …

… so that
there are
no
attacks

Random search

- Baseline for any search activity
- Inputs are generated at random until the goal of the test is fulfilled
- Random search is very poor at finding solutions when those solutions occupy a very small part of the overall search space



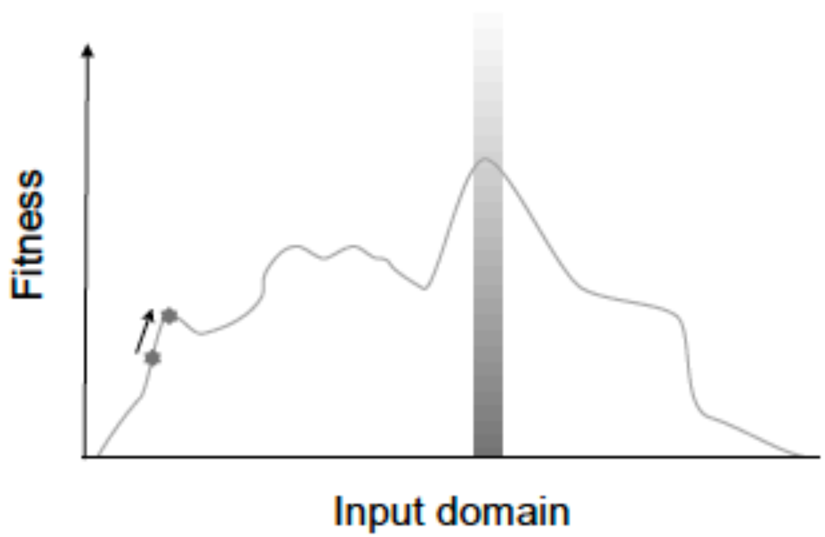
- Random search may fail to fulfil low-probability test goals

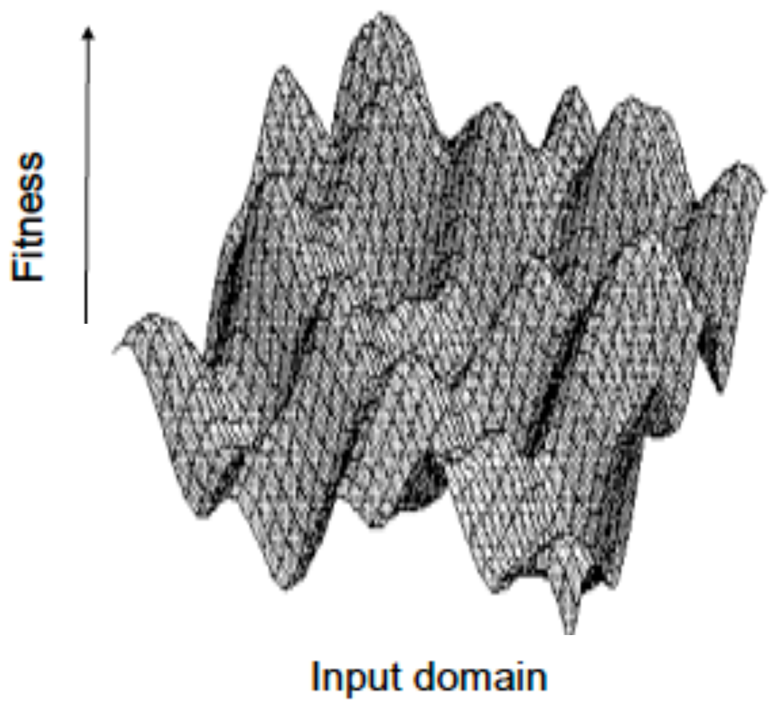
Random search vs guided search

- Random search is not effective in finding test input when they are in small parts of the input space
- Test data may be found faster and more reliably if the search is given some guidance
- This guidance can be provided by a **fitness function**
  - *It scores different points in the search space with respect to their 'goodness' or their suitability for solving the problem*

Fitness landscape

- A plot of a fitness function is referred to as the fitness landscape





Checking vs Generating

Task One:

- Write a method **to determine which is the better** of two placements of N queens

Task Two:

- Write a method **to construct a board placement** with N non attacking queens

Checking vs Generating

Which seems easier to you?

Task One:

- Write a method **to determine which is the better** of two placements of N queens

Task Two:

- Write a method **to construct a board placement** with N non attacking queens

Checking vs Generating

Search Based Software Engineering

- Write a method to determine which is the better of two solutions

Conventional Software Engineering

- Write a method to construct a perfect solution

Checking vs Generating

Search Based Software Engineering

- Write a method to determine which is the better of two solutions

Conventional Software Engineering

- Write a method to construct a perfect solution

Checking vs Generating

Search Based Software Engineering

- Write a fitness function (also said cost function) to determine which is the better of two solutions

Conventional Software Engineering

- Write a method to construct a perfect solution

Checking vs Generating

Search Based Software Engineering

- Write a fitness function to guide a search of a solution

Conventional Software Engineering

- Write a method to construct a perfect solution

Checking vs Generating

Search Based Software Engineering

- Write a fitness function to guide automated search

Conventional Software Engineering

- Write a method to construct a perfect solution

For the eight queens problem

Search Based Software Engineering

- Write a fitness function to guide automated search of placements of N queens

Conventional Software Engineering

- Write a method to **construct a board placement** with N non attacking queens

The two SBSE ingredients

- **Representation should be easy:**
  - We always represent Software Engineering problems with data structures

- **Fitness function is often easy:**
  - We often define metrics

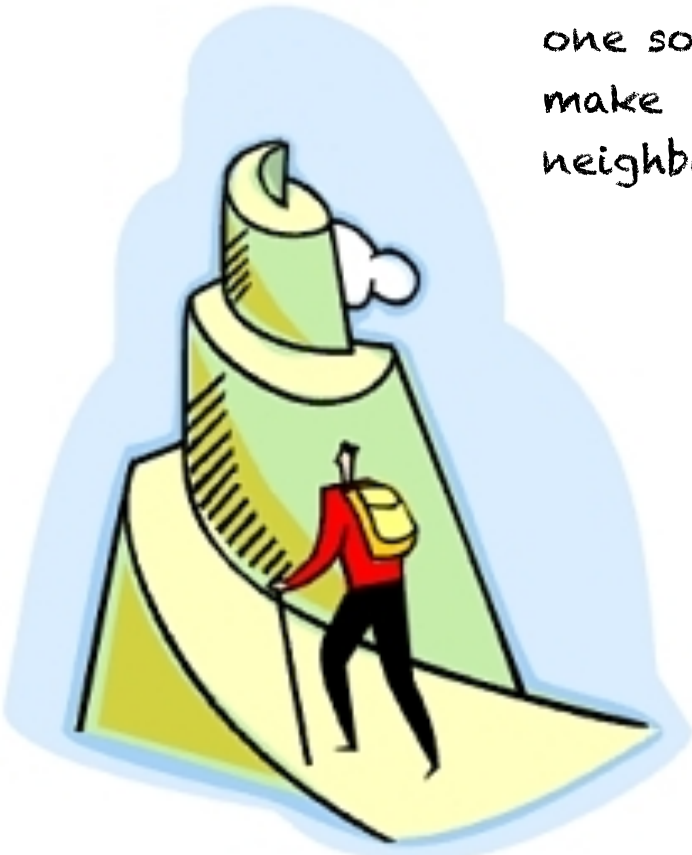
Major issue

Uncertainty: identify a suitable fitness function

We have plenty of search algorithm

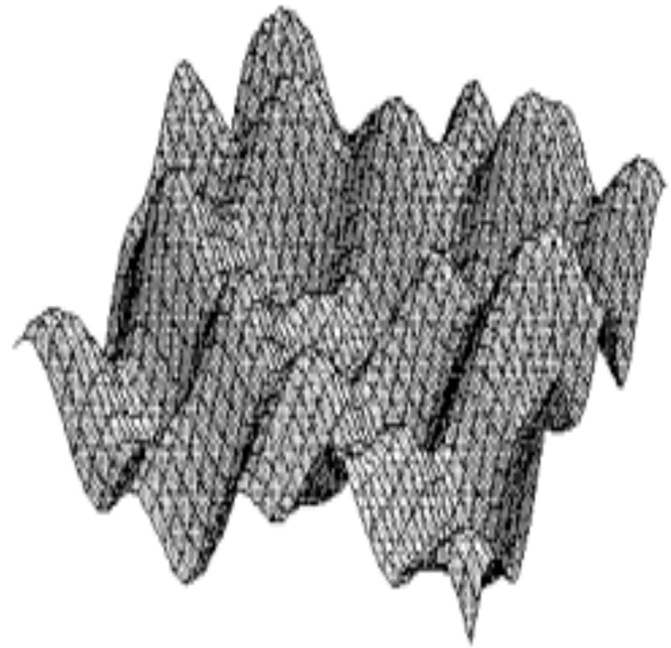
- Genetic Algorithms,
- Hill climbing,
- Simulated Annealing,
- Random,
- Tabu Search,
- Estimation of Distribution Algorithms,
- Particle Swarm Optimization,
- Ant Colonies,
- Greedy

Hill Climbing - local search

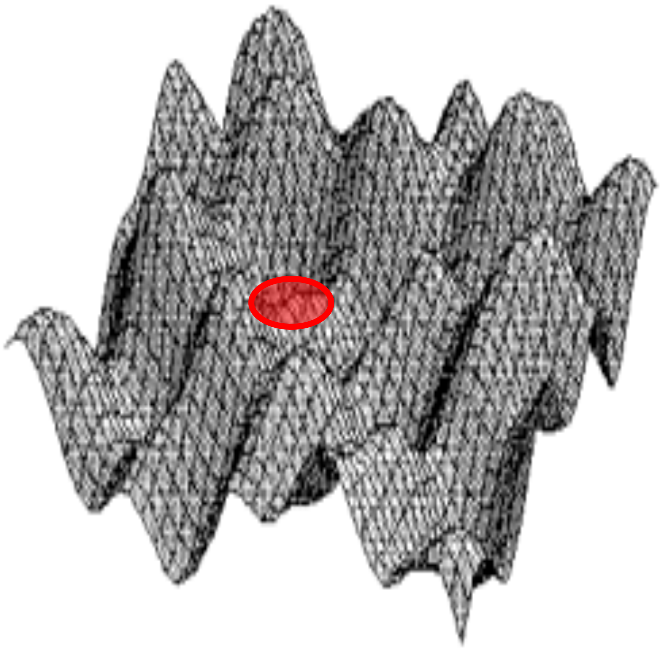


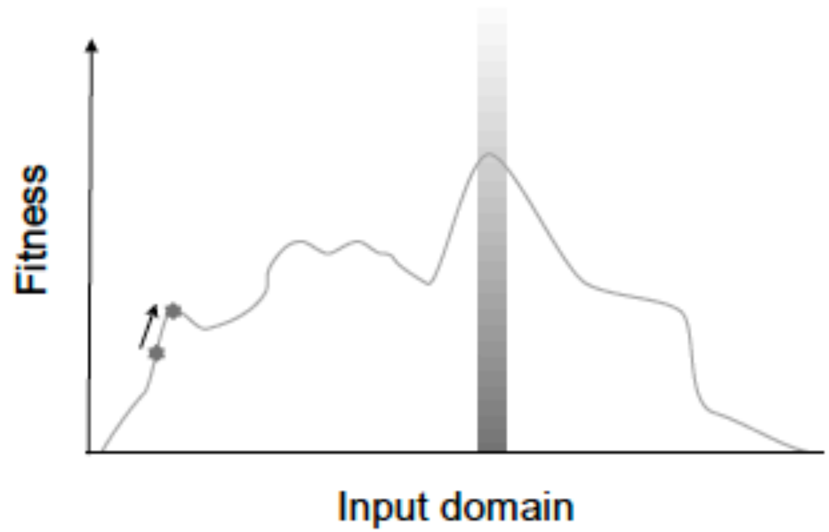
one solution at a time
make moves only in the local neighborhood of those solutions

Hill Climbing - local search

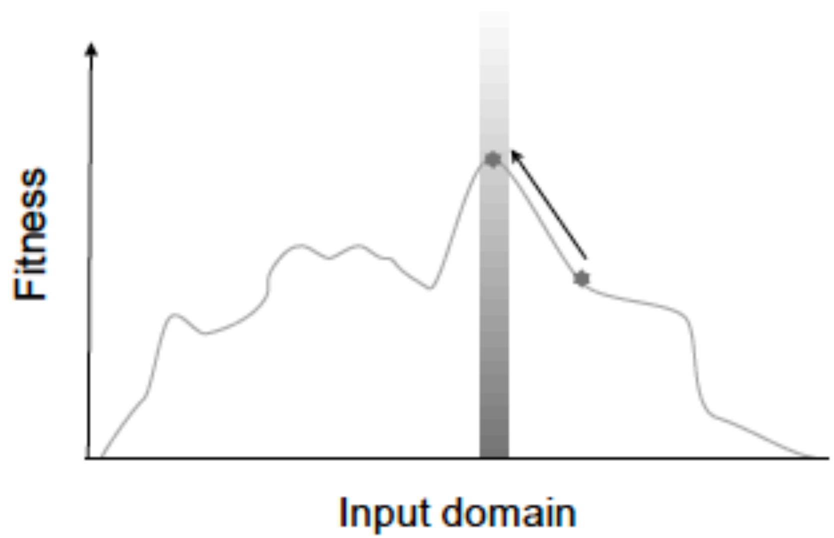


Hill Climbing - local search





(a) Climbing to a local optimum



(b) Restarting, on this occasion resulting in a climb to the global optimum

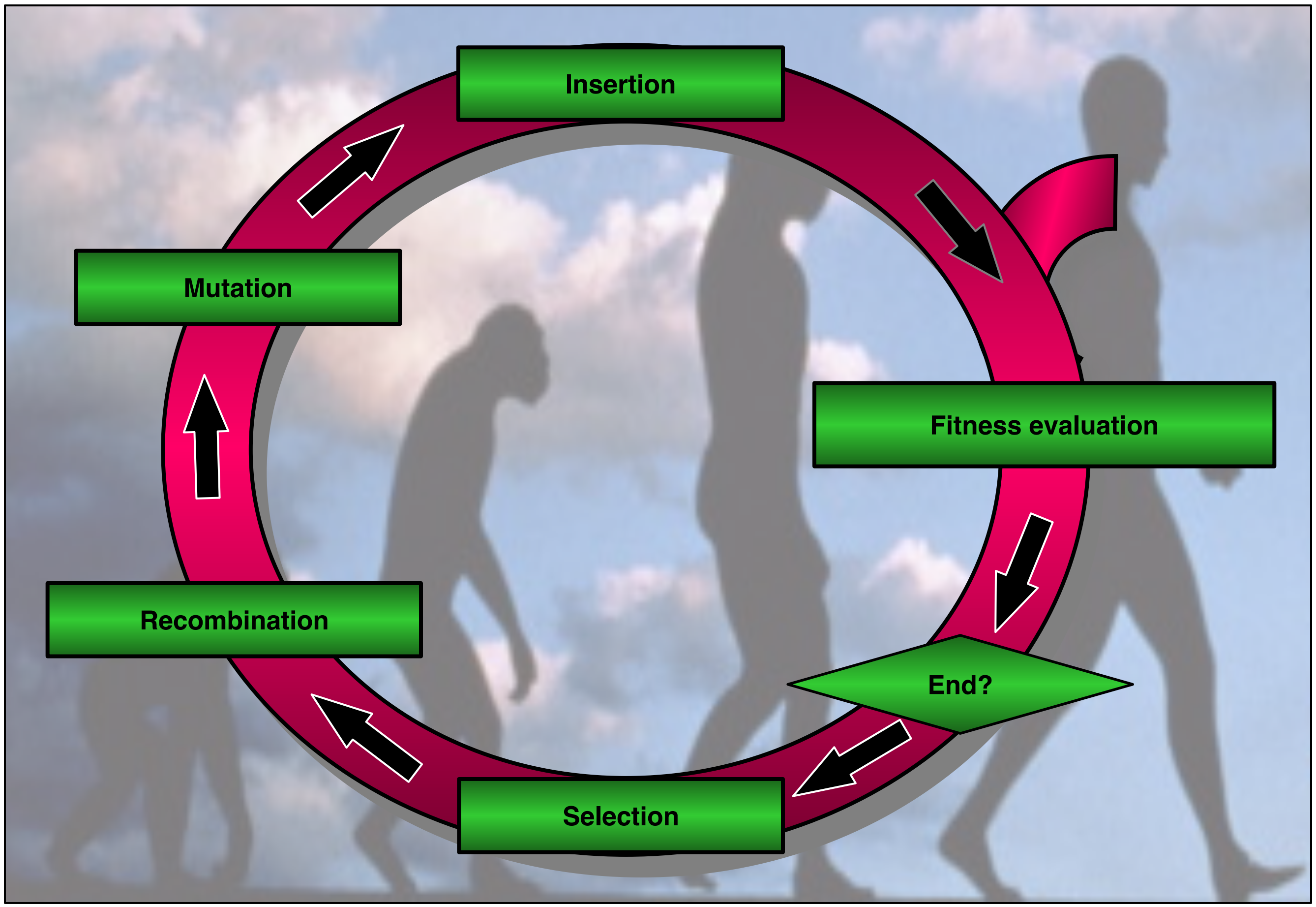
Bunch Hill Climbing Algorithm

```
bP = null;

while(searching()){
    p = selectNext();
    if(p.isBetter(bP))
        bP = p;
}
```

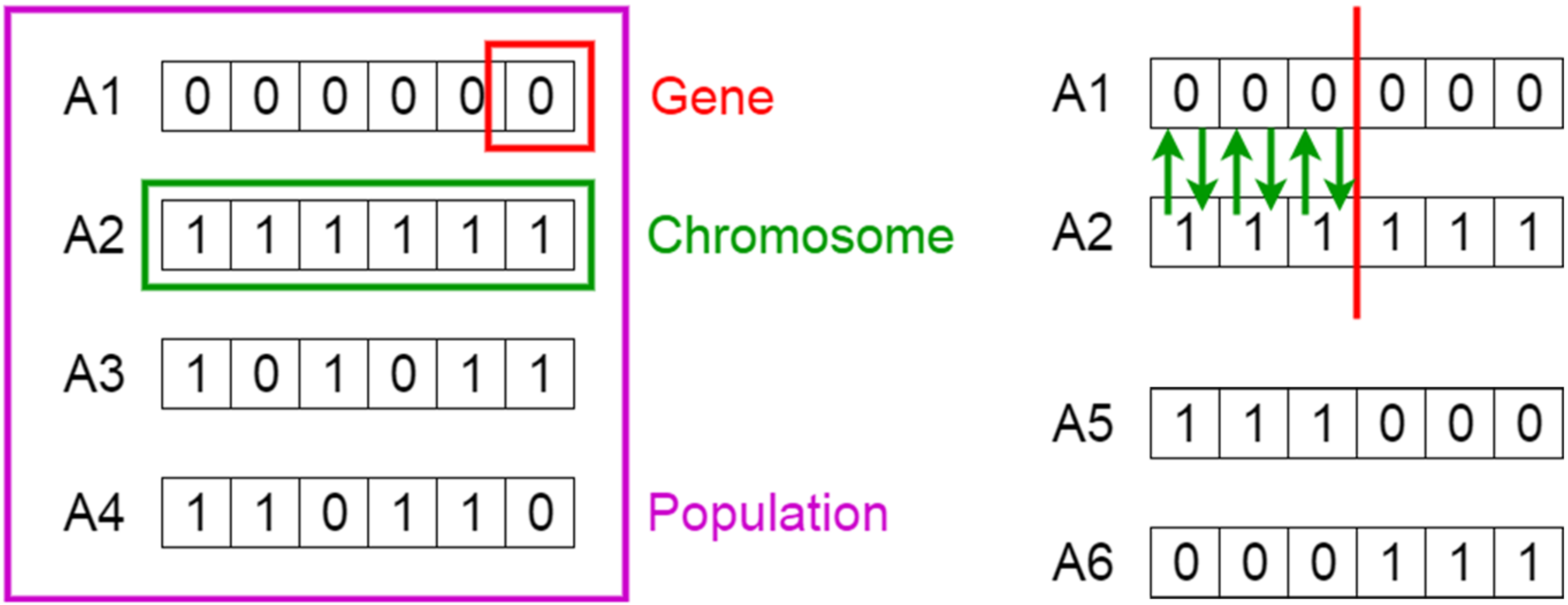
Local vs global search

- Hill climbing performs **local** search by finding one solution at a time and moving only in the local neighborhood of the solution
- Genetic Algorithms perform of **global** search, sampling many points in the search space at once

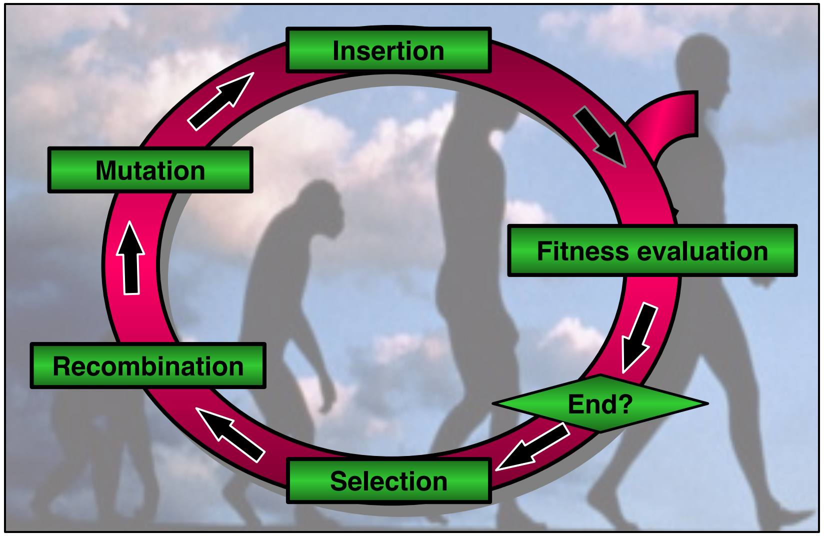
Evolutionary Algorithms



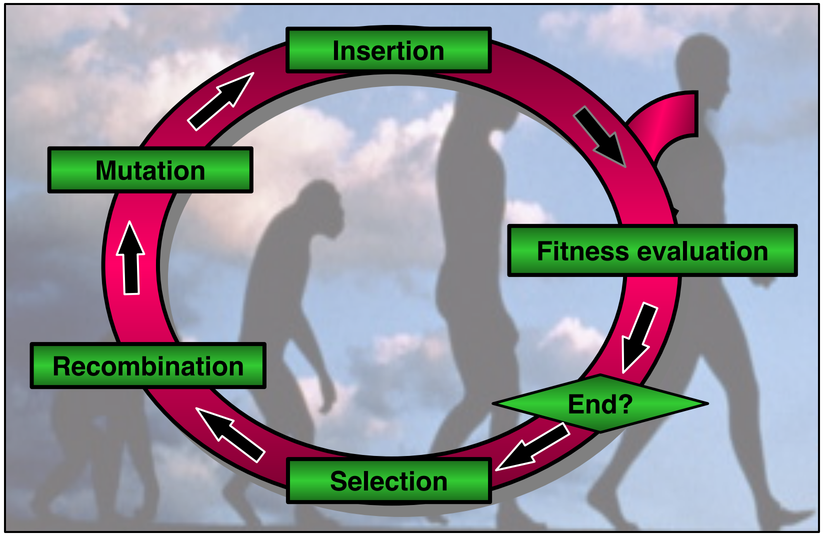
Genetic Algorithms



How does a GA work?



How does a GA work?



Population

A finite set of strings, arrays, …: the genome

1 1 1 0 0 1 1 1

1 0 1 0 0 1 1 0

1 0 0 1 1 1 1 0

1 0 0 1 1 0 0 1

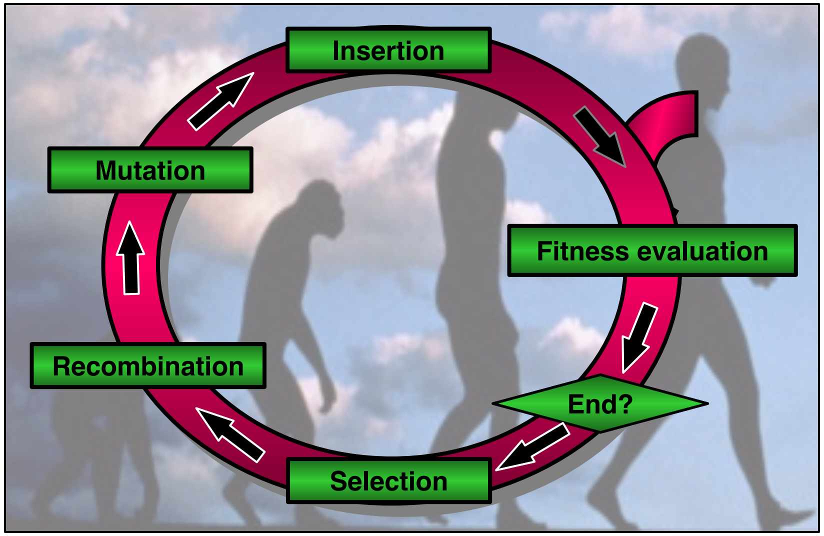
1 0 1 0 0 1 1 0

1 0 1 0 0 1 1 0

1 1 0 1 1 1 1 1

1 0 0 1 1 0 0 1

How does a GA work?



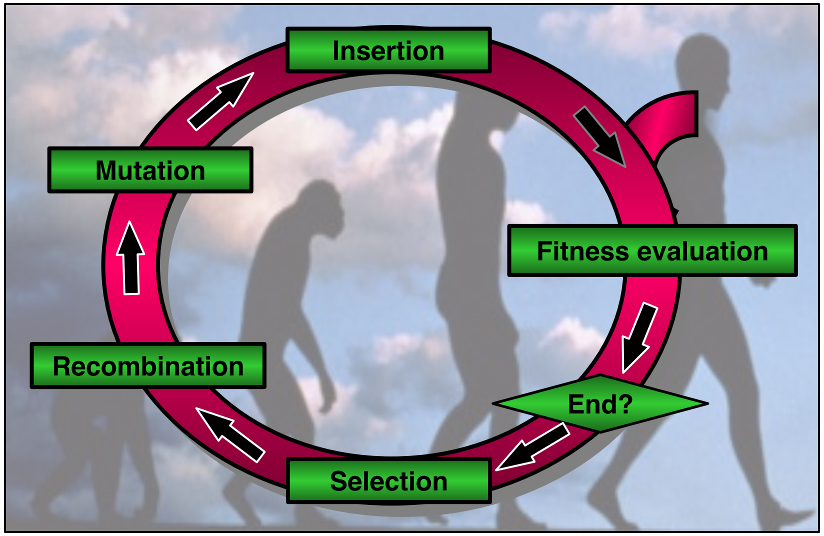
Selection operator: selects individuals for the reproduction

1 0 0 1 1 0 0 1

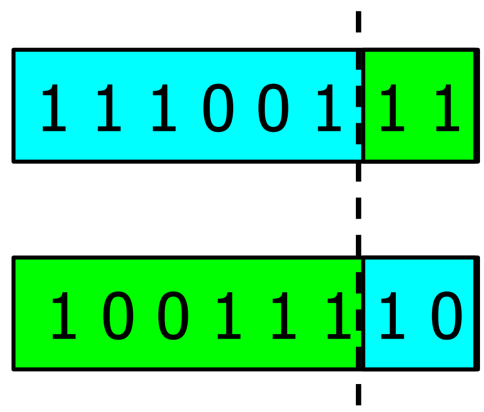
1 0 1 0 0 1 1 0

1 0 0 1 1 0 0 1

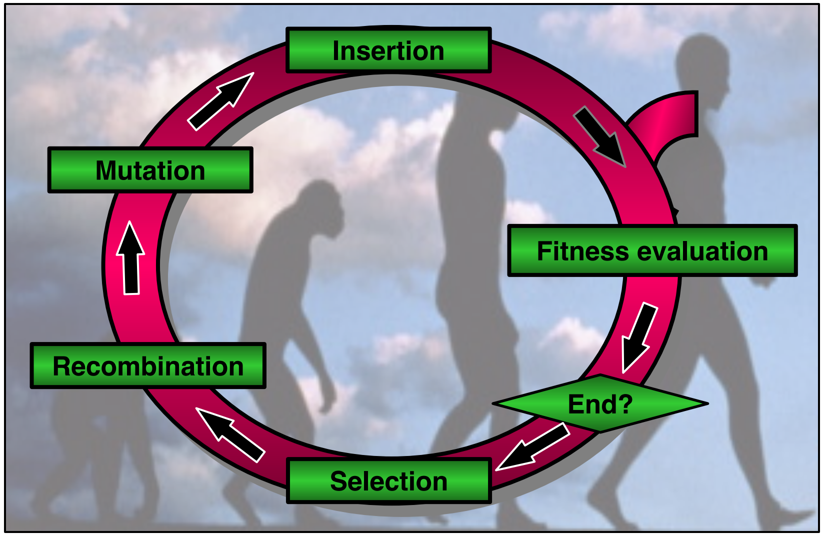
How does a GA work?



Crossover operator: produces new individuals from two *parents*, exchanging parts of their chromosomes

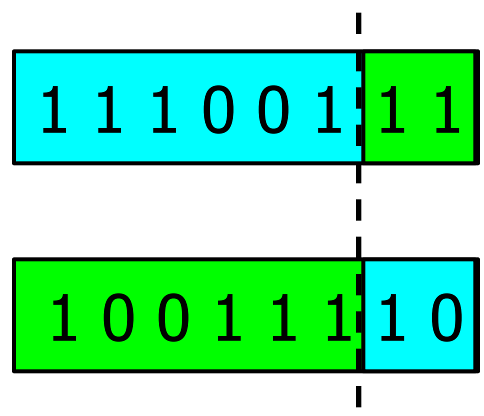


How does a GA work?



Mutation operator: randomly modifies an individual's genome

1 1 1 1 0 1



Ingredients and actions

- Chromosome (individual)
- Population
- Fitness function
- Selection
- Crossover
- Mutation
- Crossover

Encoding a chromosome

- A chromosome contains information about a solution that represents

- The most used way of encoding is a binary string

Chromosome	1101100100110110

Encoding

- **Binary encoding**
  - Arrays of Bits
- **Permutation encoding**
  - Permutations of arrays of natural numbers
- **Value Encoding**
  - Arrays of values
- **Tree encoding**
  - Trees (e.g., control flow model)

https://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

Crossover

- Generating offsprings from existing genomes

Crossover on Bits

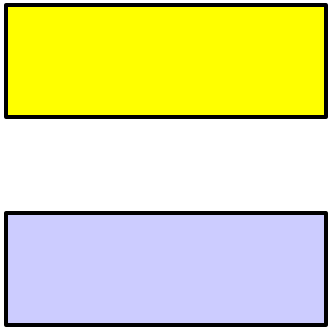
Single point crossover - one crossover point is selected, binary string from beginning of chromosome to the crossover point is copied from one parent, the rest is copied from the second parent
Two point crossover - two crossover point are selected, binary string from beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent
Uniform crossover - bits are randomly copied from the first or from the second parent
Arithmetic crossover - some arithmetic operation is performed to make a new offspring

https://www.obitko.com/tutorials/genetic-algorithms/crossover-mutation.php

Crossover

arithmetic
crossover

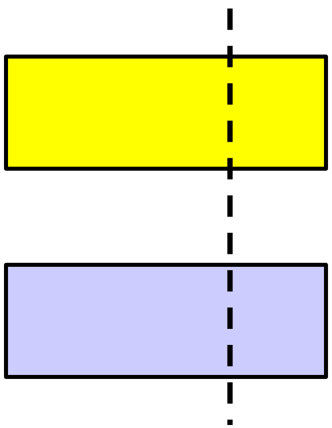
one-point
crossover



Crossover

arithmetic
crossover

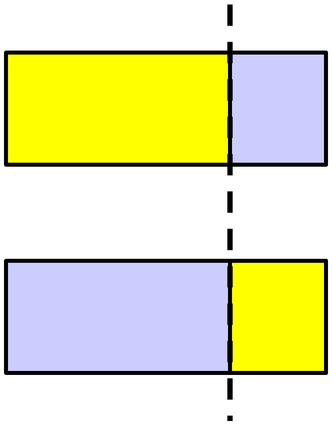
one-point
crossover



Crossover

arithmetic
crossover

one-point
crossover



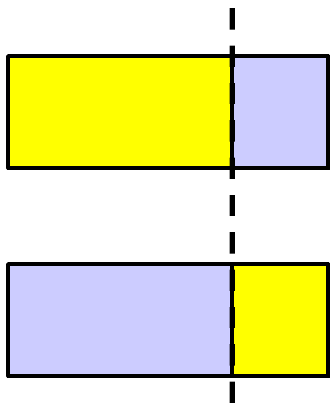
Crossover

arithmetic
crossover

0.1	0.2	0.3	0.4	0.5

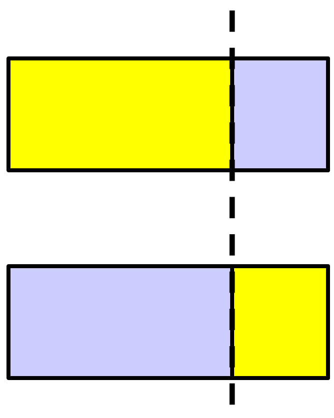
0.6	0.7	0.8	0.9	1.0

one-point
crossover



Crossover

one-point
crossover



arithmetic
crossover

0.1	0.2	0.3	0.4	0.5

0.6	0.7	0.8	0.9	1.0

Child 1=$a \cdot x+(1-a)\cdot y$ Child 2=$a \cdot y+(1-a)\cdot x$

Crossover

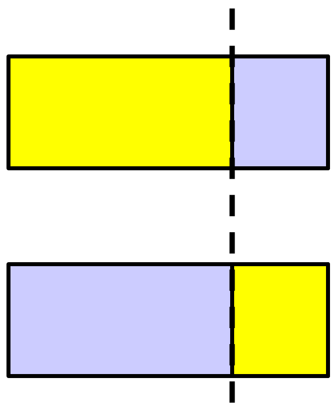
arithmetic crossover

0.1	0.2	0.3	0.4	0.5

0.6	0.7	0.8	0.9	1.0

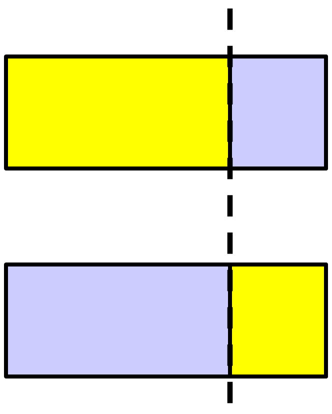
Child 1=$a \cdot x+(1-a)\cdot y$ Child 2=$a \cdot y+(1-a)\cdot x$

one-point crossover



Crossover

one-point
crossover



arithmetic
crossover

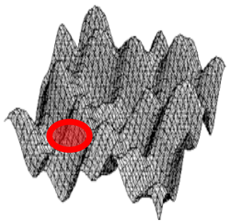
0.1	0.2	0.3	0.4	0.5
0.6	0.7	0.8	0.9	1.0

Child 1=$a \cdot x+(1-a)\cdot y$ Child 2=$a \cdot y+(1-a)\cdot x$

0.43	0.53	0.56	0.73	0.83
0.27	0.37	0.47	0.57	0.67

Mutation

- Mutate existing genomes
- Goal:
    - Try to avoid local minima/maxima by *preventing the population of chromosomes from becoming too similar to each other*, thus slowing or even stopping convergence to the global optimum



    - **Introduce diversity** into the sampled population

Mutation on Bits

- *Mutation according to a given probability*
- This probability should be set low:
  - If it is set too high, the search will turn into a *random search*

Bit string mutation

- Bit flips at **random positions**
- Example:

- The probability of a mutation of a bit is **1/L**, where **L** is the length of the binary vector

Bit string mutation

- Bit flips at **random positions**
- Example:

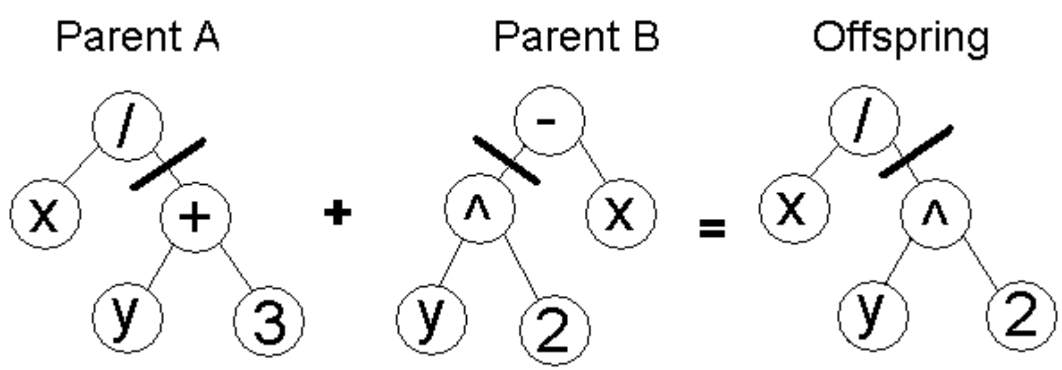
1 1 1 0 0 1

1 1 1 1 0 1

- The probability of a mutation of a bit is **1/L**, where **L** is the length of the binary vector

Other cases

- Mutation on real numbers: adding/subtracting a small number to selected entries
- Cross-over on trees: combine two parts of the parents tree at cross-over point



Effects of Genetic Operators

- Using *selection alone* will tend to fill the population with copies of the best individual from the population
- Using *selection and crossover* operators will tend to cause the algorithms to converge on a good but sub-optimal solution (e.g., no diversity)

Effects of Genetic Operators

- Using *mutation alone* induces a random walk through the search space
- Using *selection and mutation* creates a parallel, noise-tolerant, hill climbing algorithm (e.g., no inheritance)

Genetic Algorithms: pseudocode

```
Initialize population P[0];
generation=0;
while(generation <
  max_number_of_generations)
     Evaluate P[generation];
     generation=generation+1;
     Select P[generation] from
  P[generation-1];
     Crossover P[generation];
     Mutate P[generation];
end while
```

The algorithm

- Randomly initialize population(t)
- Determine fitness of population(t)
- Repeat
  - Select parents from population(t)
  - Perform crossover on parents creating population(t+1)
  - Perform mutation of population(t+1)
  - Determine fitness of population(t+1)
- until best individual is good enough

Evolutionary Testing

- Evolutionary testing aims at improving the effectiveness and efficiency of the testing process by

transforming **testing objectives into search problems** and applying evolutionary computation in order to solve them

Evolutionary Testing: HowTo

- Search space: **input domain**(s) of the system under test (**SUT**)
- Test objective needs to be **defined numerically** and **transformed in a fitness function**
- Fitness is computed by **monitoring program execution results** (output, performance, etc.)
- Iterative procedure

Testing

- Different test goals = different fitness functions

Two assumptions

- Two assumptions in order to use search based algorithm in testing
- **Representation.** Candidate solutions must be capable of being encoded as sequences of elements
- **Fitness function.** The fitness function guides the search by evaluating candidate solutions. The fitness function is problem-specific

Evolutionary Testing: HowTo

- Start with a set of randomly generated test input data
- Then execute on test data to gather information
- Monitored values are used to compute the fitness
- Evolution towards a testing objective, indicated in the fitness function

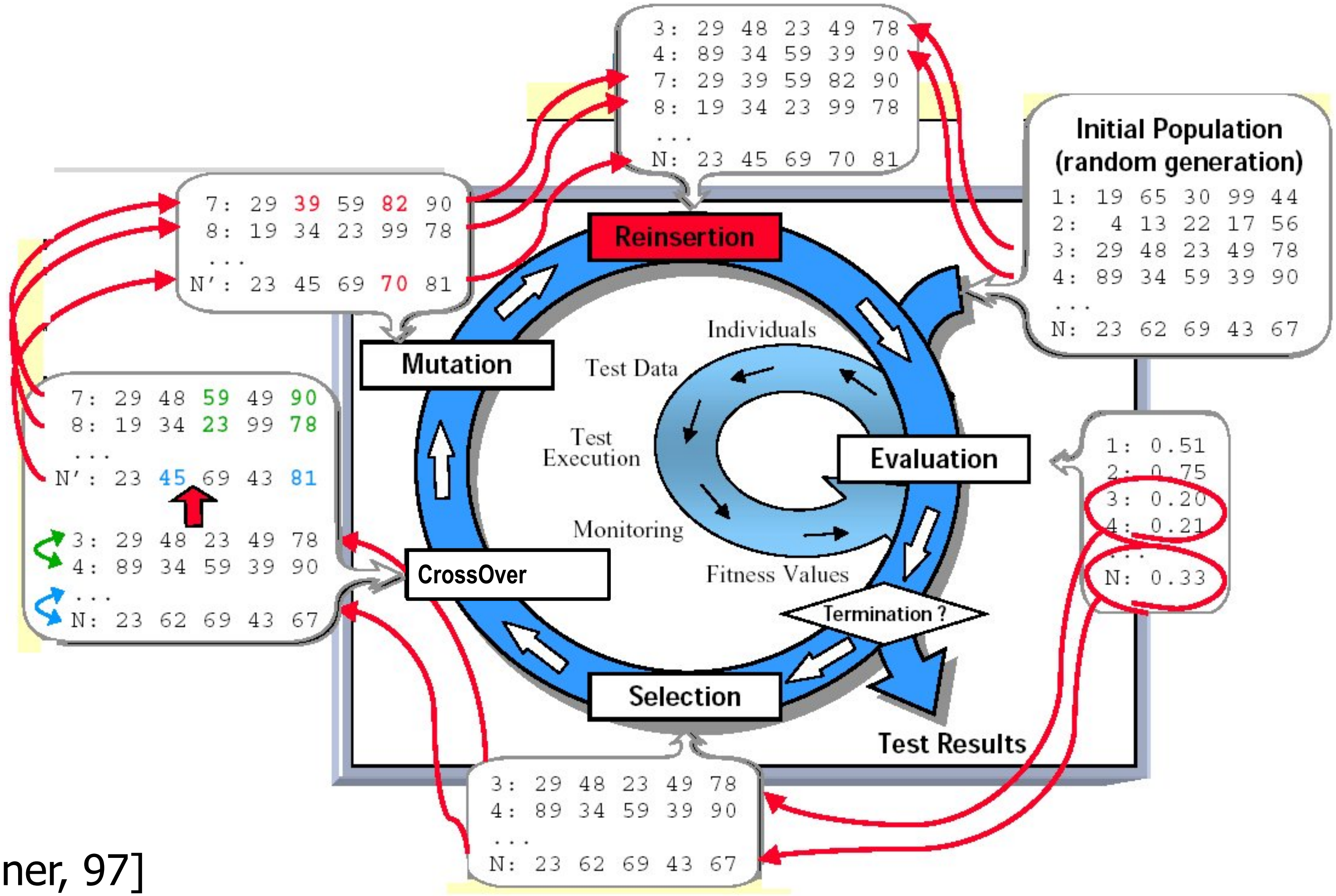
Genome encoding

	I_1	I_2	I_3	…	…	I_n
T_{x1}	<24.5>	<11.3>	<abc>	…	…	<ghi>
…	…	…	…	…	…	…
T_{x5}	<7.6>	<4.7>	<def>	…	…	<xza>

Typical Parameter Settings

- dim pop = 70
- n gen =500
- P mutation=0.01

More Details



[Wegener, 97]

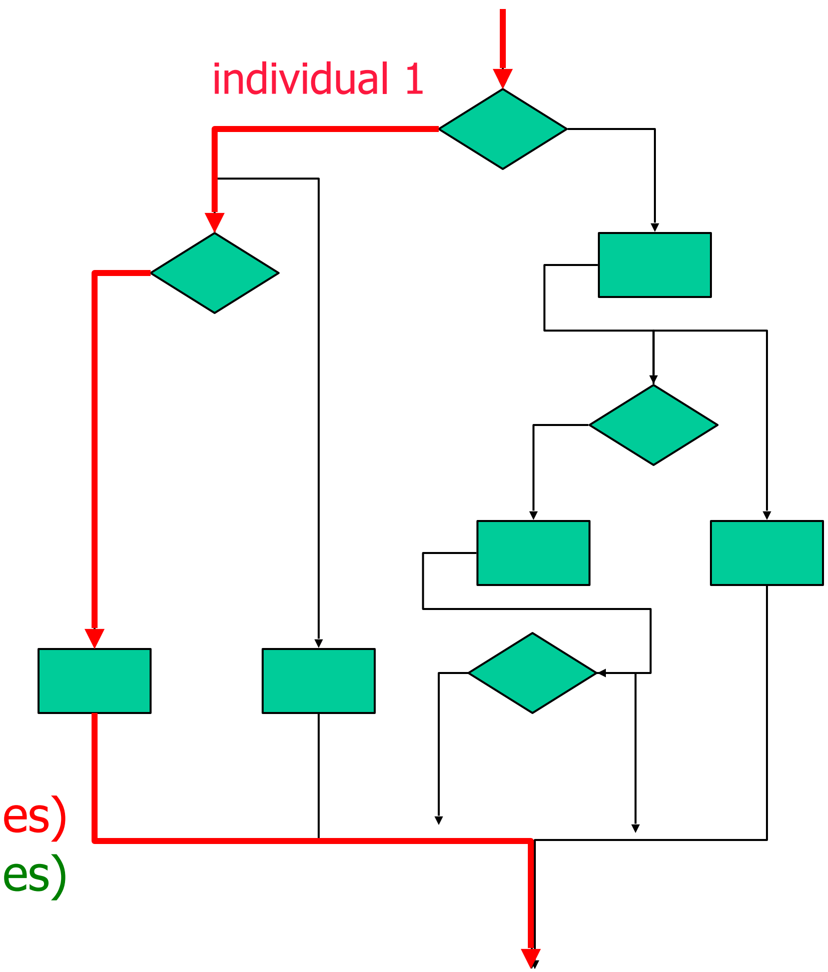
Example: Structural Testing

- Aim
  - Generate a set of test data to **cover structural properties**
- Fitness function a normalized combination of
  - **Approach level:** fitness is number of control nodes in a path and dependent from a target one not covered by test execution with a given input
  - **Branch distance.** At the node at which the test execution diverges from the target branch at some approach level, the branch distance is computed.

Example: Coverage-oriented approaches

Individual: path to same target

- Fitness of individual for different coverage criteria
  - % of covered statements
  - **% of covered branches**
  - Path coverage: % executed paths
- Good results, better than random testing

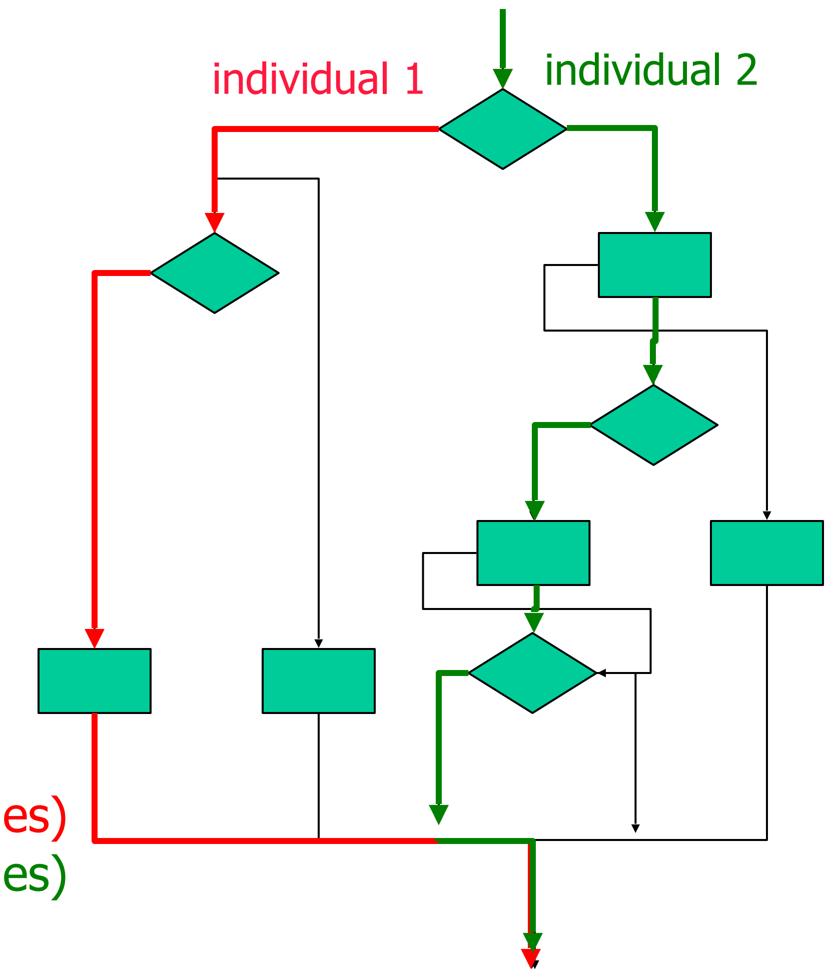


Individual 1: fitness=2 (2 branches)
Individual 2: fitness=3 (3 branches)

Example: Coverage-oriented approaches

Individual: path to same target

- Fitness of individual for different coverage criteria
  - % of covered statements
  - **% of covered branches**
  - Path coverage: % executed paths
- Good results, better than random testing

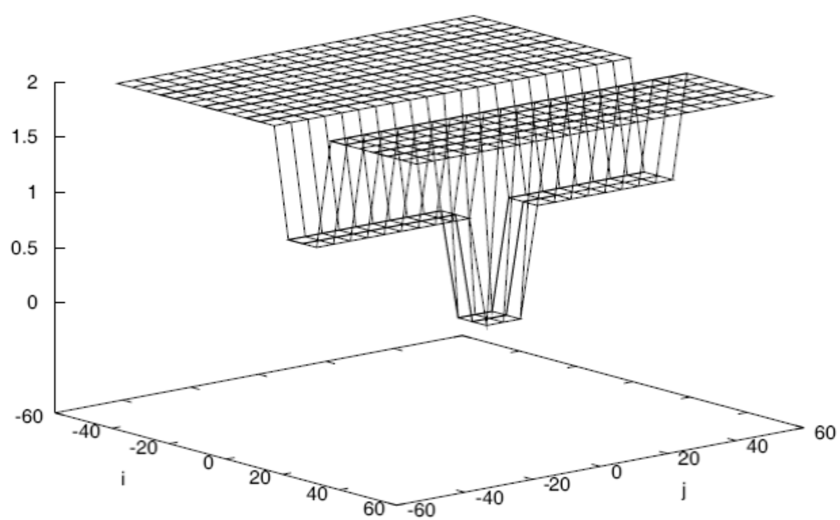


Individual 1: fitness=2 (2 branches)
Individual 2: fitness=3 (3 branches)

Example: Control-Oriented

```
s    void landscape_example(int i, int j)
     {
1        if (i >= 10 && i <= 20)
         {
2            if (j >= 0 && j <= 10)
             {
3                // target statement
                 // ...
             }
         }
     }
```

Fitness function:
True conditions
not executed



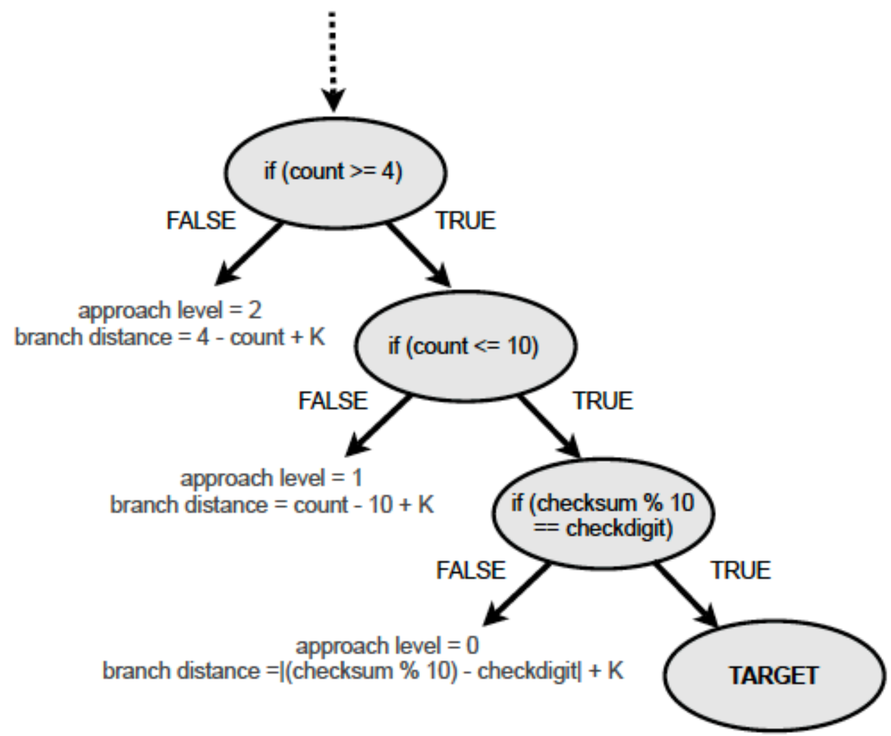
Fitness landscape

Example

```
int cas_check(char* cas) {
  int count = 0, checksum = 0, checkdigit = 0, pos;

  for (pos=strlen(cas)-1; pos >= 0; pos--) {
    int digit = cas[pos] - '0';

    if (digit >= 0 && digit <= 9) {
      if (count == 0)
        checkdigit = digit;
      if (count > 0)
        checksum += count * digit;

      count ++;
    }
  }

  if (count >= 4)
    if (count <= 10)
      if (checksum % 10 == checkdigit)
        return 0;
      else return 1;
    else return 2;
  else return 3;
}
```

Target

Code of interest



EXAMPLE OF HOW TO APPLY THE FUNCTION δ ON SOME PREDICATES. k CAN BE ANY ARBITRARY POSITIVE CONSTANT VALUE. A AND B CAN BE ANY ARBITRARY EXPRESSION, WHEREAS a AND b ARE THE ACTUAL VALUES OF THESE EXPRESSIONS BASED ON THE VALUES IN THE INPUT SET I.

Predicate θ	Function $\delta_\theta(I)$
A	if a is 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 $(a - b) + k$
$A > B$	$\delta_{B<A}(I)$
$A \geq B$	$\delta_{B \leq A}(I)$
$\neg A$	Negation is moved inward and propagated over A
$A \wedge B$	$\delta_A(I) + \delta_B(I)$
$A \vee B$	$min(\delta_A(I), \delta_B(I))$

δ = branch distance: how distant is the input I to the value that makes the predicate θ TRUE

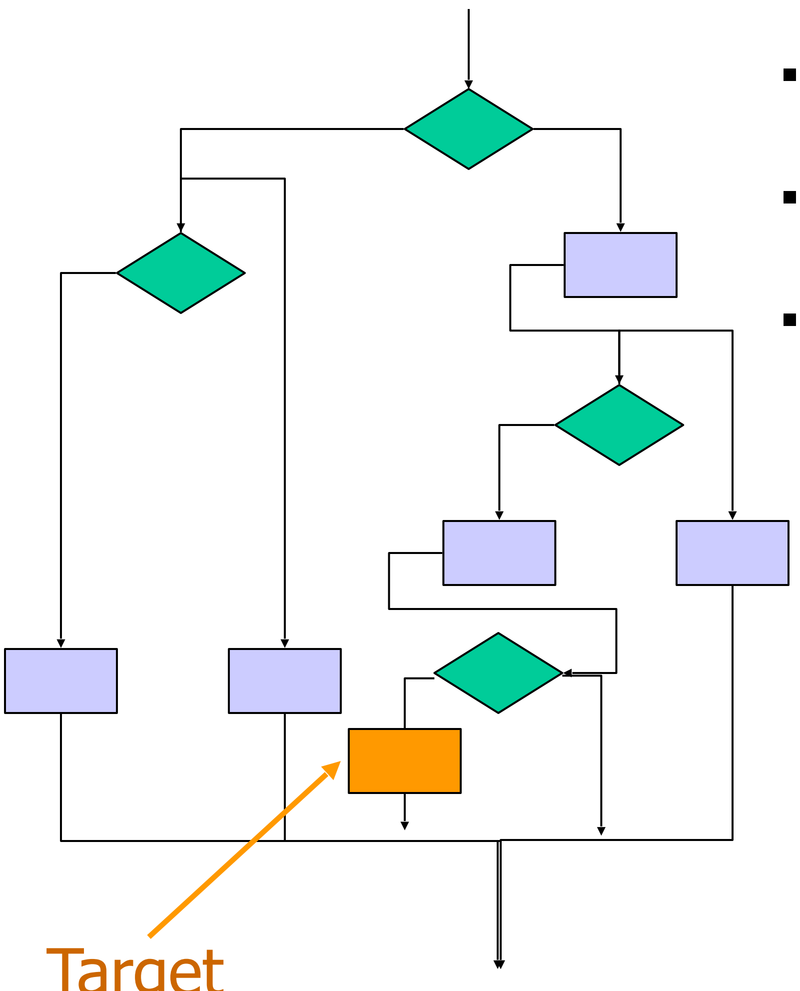
Exercise

```
[1] int foo (int a, int b, int c, int d, float e) {
[2]    if (a == 0) {
[3]         return 0;
[4]     }
[5]     int x = 0;
[6]     if ( (a==b) II ( (c == d) && bug(a) ) ) {
[7]         x=1;
[8]     }
[9]     e = 1/x;
[10]    return e;
[11] }
```

bug(a) = TRUE if !a==0 else 0

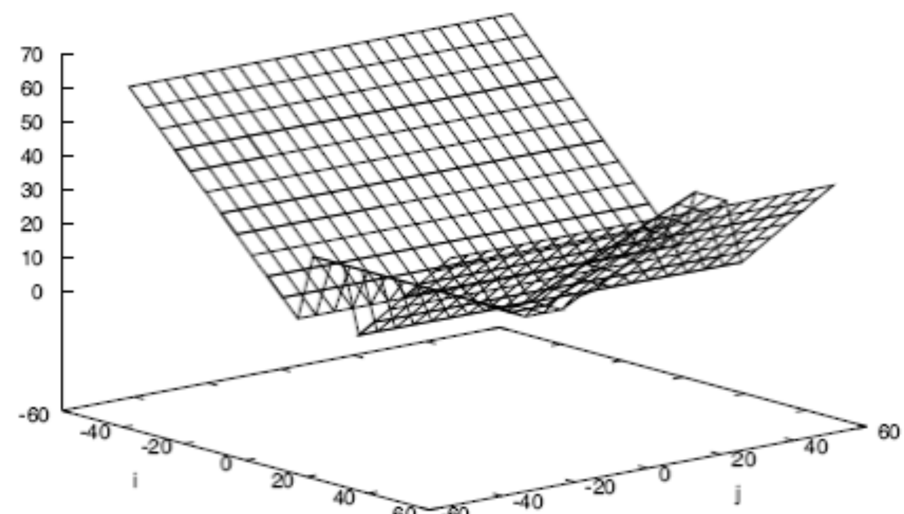
How many control nodes does T(0,0,0,0,0) execute if the target node is the output of the true branch of predicate at line [6]? Approach distance? Branch distance of (1,0,0,0,0)

Distance-Oriented



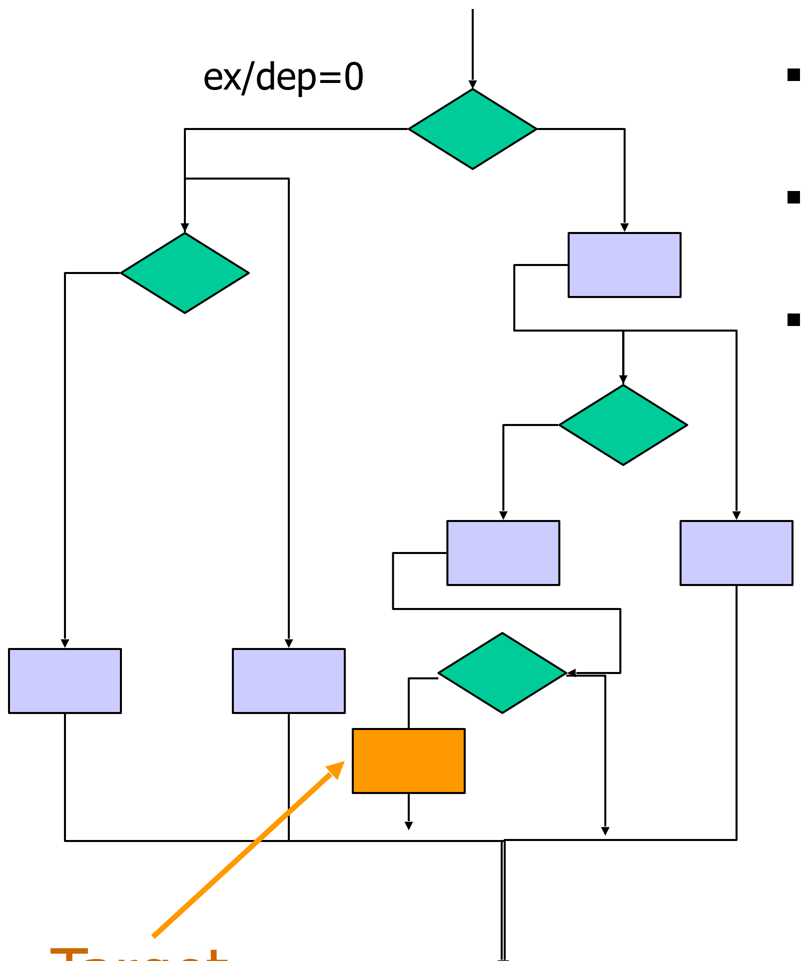
- Identify **relevant branching** statements from the CFG
- **Approximation level:** distance from the target
- **branch_distance:** condition distance when a critical node is taken
- **Objective function:**

$$\frac{\text{executed}}{\text{dependent}} \times \text{branch\_distance}$$



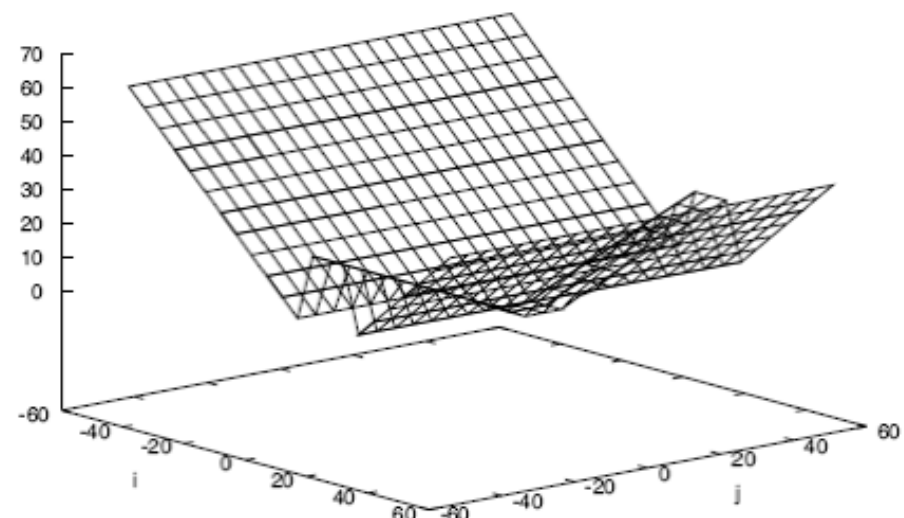
Fitness landscape

Distance-Oriented



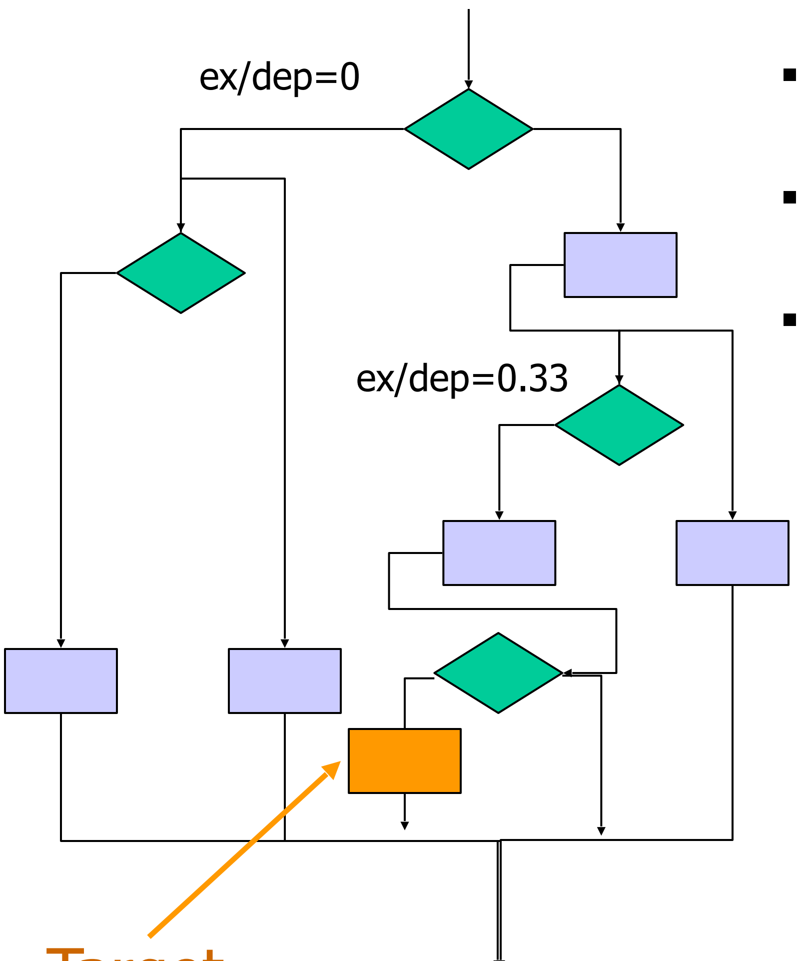
- Identify **relevant branching** statements from the CFG
- **Approximation level:** distance from the target
- **branch_distance:** condition distance when a critical node is taken
- **Objective function:**

$$\frac{\text{executed}}{\text{dependent}} \times \text{branch\_distance}$$



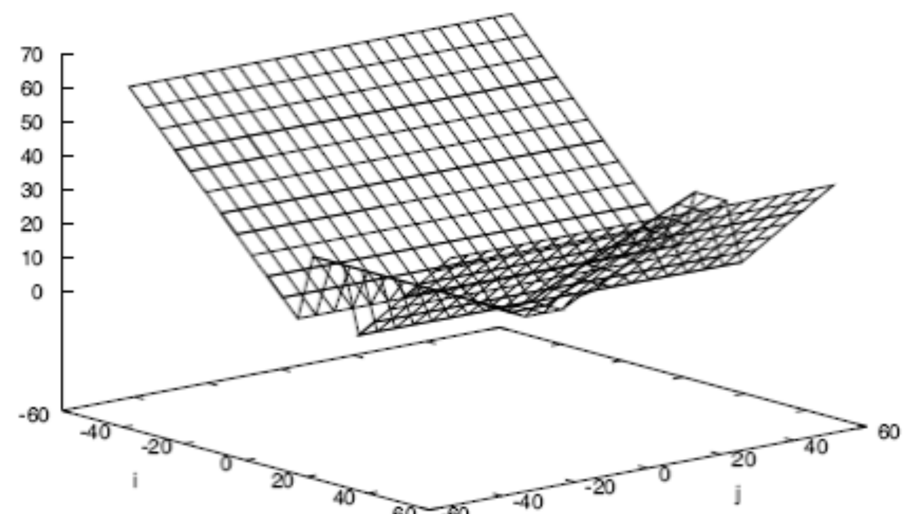
Fitness landscape

Distance-Oriented



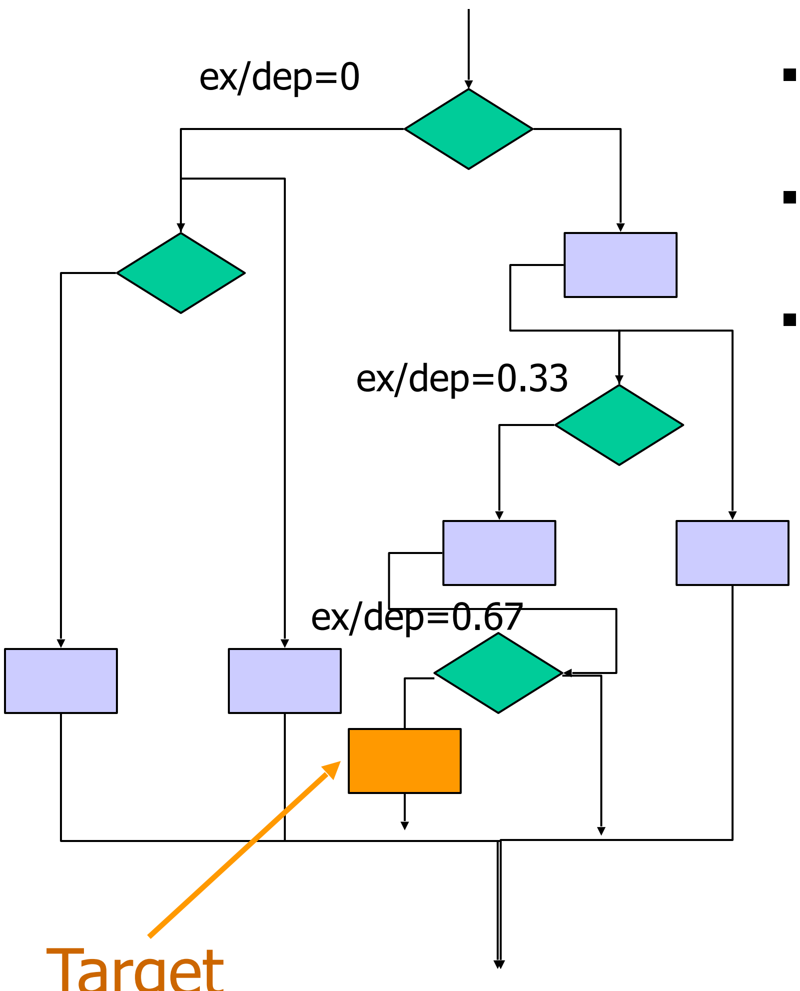
- Identify **relevant branching** statements from the CFG
- **Approximation level:** distance from the target
- **branch_distance:** condition distance when a critical node is taken
- **Objective function:**

$$\frac{\text{executed}}{\text{dependent}} \times \text{branch\_distance}$$



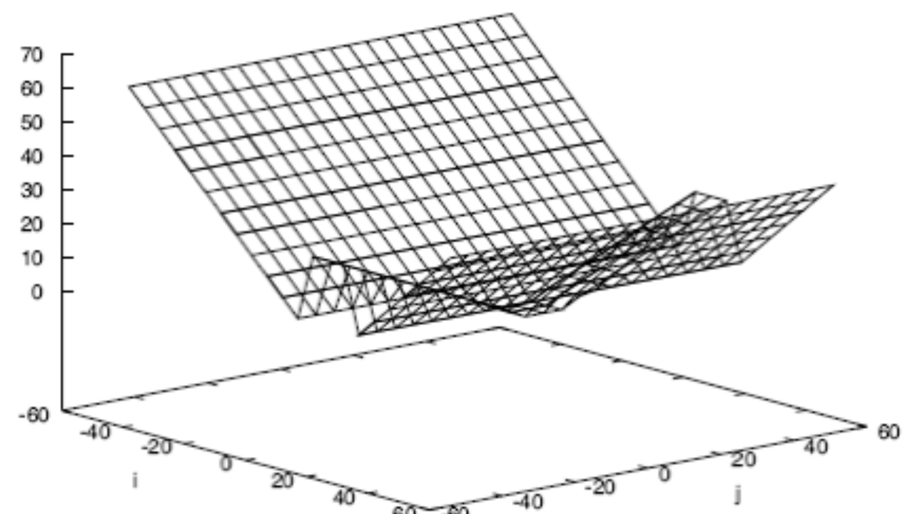
Fitness landscape

Distance-Oriented



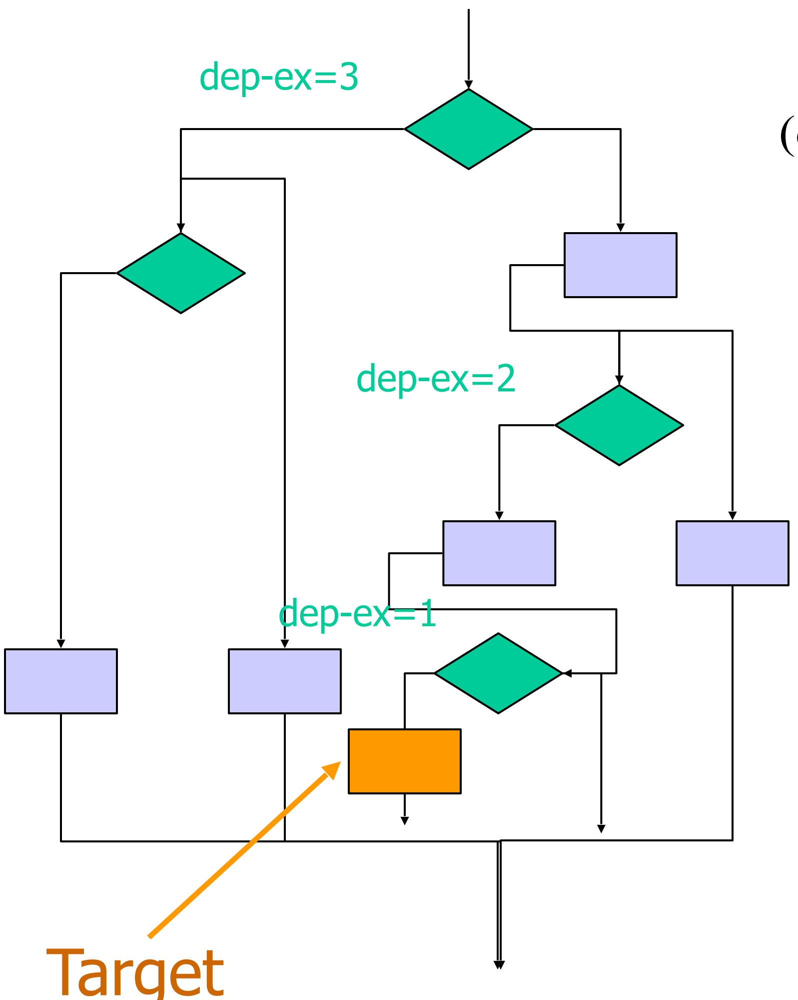
- Identify **relevant branching** statements from the CFG
- **Approximation level:** distance from the target
- **branch_distance:** condition distance when a critical node is taken
- **Objective function:**

$$\frac{\text{executed}}{\text{dependent}} \times \text{branch\_distance}$$



Fitness landscape

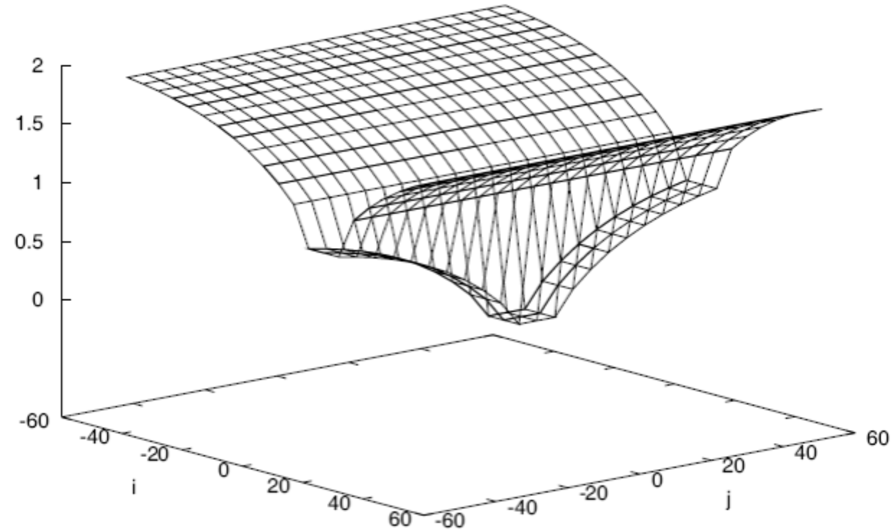
Distance-Oriented [Wegener]



- Objective function:

$$(\text{dependent} - \text{executed} - 1) + \text{m\_branch\_distance}$$

- m_branch_distance: branch_distance normalized in the interval [0,1]



Multi-objective search

Barbara Russo
SwSE - Software and Systems Engineering

Objectives of the search

- Optimization means search in the space to minimize or maximize one or more fitness functions
  - Objective of the search can be single or multiple

Objective function

considering a minimization scenario

- Formally, it can be expressed as:

$$argmin\{f_1(x); f_2(x); \ldots; f_m(x)\}$$

- where **m** is the number of objectives and **x** is a solution in the feasible solution space X, i.e., $x \in X \subset R^n$ (n is the number of decision variables of the problem)

Solutions

- Two components of a search algorithm:
  - The representation, i.e., how x can be structured and changed
  - The objective function, i.e., how each single f_i can be formulated to distinguish between the good and bad solutions
- A solution x_1 dominates x_2 if and only if x_1 is **not worse than x_2 for all the objectives and better for at least one objective**

Pareto front with multiple objectives

- For a solution $x \in X$, if there is no solution in X dominating x, then x is **Pareto optimal**
- **Pareto optimal set**: the set of all Pareto optimal solutions
  - Its image in the objective space is the **Pareto front**

Search budget

- **Budget**: number of evaluations and time
- **Reasonable convergence**: the best solution found does not change for some number of iterations under a search budget

Pareto search

- Goal: to search for a set of solutions that can well represent the Pareto front
- Widely used with population-based algorithms (e.g., evolutionary algorithms)
- Examples
  - **NSGA-II**: Pareto dominance relation
  - As long as they assume no clear preferences exist and seek to approximate the Pareto front:
    - **MOEA/D**: in combination with multiple weight vectors are used;
    - **AdaW** : with weights that are changed during the search
    - **IBEA** : a single dominance-preserving indicator
    - **NGA:** most used in requirements prioritization

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182–197, 2002.

D. E. Goldberg, Genetic algorithms. Pearson Education India, 2006.

Weighted search

- Goal: convert the problem into a single-objective one through aggregating the objective functions
- For example, given a weight vector $[w_1, w_2, \ldots, w_m]$ the multi-objective problem can be converted into finding the best fitness of a weighted sum:

$$argmin\{w_1 f_1(x) + w_2 f_2(x) + \ldots + w_m f_m(x)\}$$

Preference

- Weights can be unbalanced
- For instance: “the project profit is three times
- more important than the cost”, the weight for
- the profit and cost objectives can thus be 0.75 and 0.25, respectively

Quantify objectives of the search

- The objectives to be optimized must be used directly to assess the quality of a search algorithm for the SBSE problem
- For example, a higher testing coverage (a search objective) may not necessarily detect more bugs

Search process

- bi-objective minimization scenario