Artificial Intelligence

Evolutionary Algorithms

Lesson 1: Introduction to Evolutionary Computing

Vincenzo Piuri

Università degli Studi di Milano

Contents

- Introduction
- Biological basics
- Principles of evolutionary computing
- Introduction Example: The *n*-Queens Problem

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Introduction

Solving optimization problems

- Definition
  - An optimization problem $(\Omega, f, \succ)$ is given by a (search) space Ω, an evaluation function $f : \Omega \longrightarrow \mathbb{R}$, that assigns a quality assessment to all candidate solutions, as well as a (comparison) relation $\succ \in \{<, >\}$
  - The set of global optima $\mathcal{H} \subseteq \Omega$ is defined as

$$\mathcal{H} = \{x \in \Omega \mid \forall x' \in \Omega : f(x) \succcurlyeq f(x')\}$$

- Given: an optimization problem $(\Omega, f, \succ)$
- Wanted: an element $x \in \Omega$ which optimizes the function f in the whole search space

Fundamental approaches

- Analytical solution

  Efficient, but rarely applicable

- Exhausting exploration

  Very inefficient, so only usable in small search spaces

- Random search

  Always usable, but mostly inefficient

- Guided search

  Precondition: similar elements in Ω have similar function values

Biological basics

Motivation

- Evolutionary Algorithms (EA) are grounded on theory of biological evolution

- Fundamental principles
  - Beneficial traits resulting from random variation are favored by natural selection
  - Better chances of procreation and multiply of individuals with beneficial traits ("differential reproduction")

- Evolution theory explains diversity and complexity of species and allows unification of all different disciplines in biology

Principles of organismic evolution (1)

- Diversity
  - All forms of life (even of the same species) differ from each other
  - Even different genetic material $\Longrightarrow$ diversity of species
  - Currently existing life forms = tiny fraction of all theoretically possible ones
- Variation

  New variants are continuously created by mutation and genetic recombination (sexual reproduction)
- Inheritance
  - Variations are heritable, as long as entering the germ line
  - Are genetically passed to the next generation
  - Generally no inheritance of acquired traits (Lamarckisms)

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Principles of organismic evolution (2)

- Speciation
  - Genetically diverge of individuals and populations

    $\Longrightarrow$ new species (no crossbreeding of the members)
  - Character branching structure of phylogenetic "pedigree"
- Birth surplus / Overproduction, nearly all life forms:
  - More offspring that can ever become mature enough to procreate themselves
- Adaptation / Natural Selection / Differential Reproduction
  - On average: hereditary variations of the survivors of a population $\Longrightarrow$ increases adaptation to the local environment
  - The slogan "survival of the fittest" is misleading
  - Rather: "different fitness $\Longrightarrow$ different reproduction"

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Principles of organismic evolution (3)

- Randomness / Blind Variation
  - Variations are triggered / initiated / caused by random
  - No concentration on certain traits / beneficial adaptions
  - Non teleological, from the Greek: $\tau\epsilon\lambda o\varsigma$— goal, purpose
- Gradualism
  - Variations happen in comparatively small steps
  - Phylogenetic changes = gradual and relatively slow
- Evolution / Transmutation / Inheritance with Modification
  - Adaptation to environment $\Longrightarrow$ species evolve in the course of time
  - Theory of evolution opposes creationism (claim: immutability of the species)

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Principles of organismic evolution (4)

- Discrete Genetic Units
  - Store / Transfer / Change of genetic information in discrete units
  - no continuously blend of hereditary traits
  - otherwise: Jenkins nightmare through recombination (disappearance of any differences in a population)
- Opportunism
  - processes of evolution work exclusively on what is present
  - better/optimal solutions are not found if intermediary stages exhibit certain fitness handicaps
- Evolution-strategic Principles
  - not only organisms are optimized, but also the mechanisms of evolution: reproduction and mortality rates, life spans, vulnerability to mutations, mutation step sizes, etc.

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Principles of organismic evolution (5)

- Ecological Niches
  - competitive species can tolerate each other if they occupy different ecological niches ("biospheres")
  - biological diversity of species is possible in spite of competition and natural selection
- Irreversibility
  - course of evolution is irreversible and unrepeatable
- Unpredictability
  - course of evolution is neither determined, nor programmed, not predictable
- Increasing Complexity
  - biological evolution is increasingly more complex
  - open problem: how can we measure the complexity of life forms?

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Principles of evolutionary algorithms

Fundamental terms and meaning (1)

notion	biology	computer science
individual	living organism	solution candidate
chromosome	DNA-histone-protein-strand	sequence of comp. objects
	describes „construction plan“ or (some of the traits) of an individual in encoded form	
	usually multiple chromosomes per individual	usually only one chromosome per individual
gene	part of a chromosome	computational object
	is the fundamental unit of inheritance which determines a (partial) characteristic of an individual	
allele (allelomorph)	form or „value“ of gene	value of comp. object
	in each chromosome at most one form/value of a gene	
locus	position of a gene	position of comp. object
	at each position in chromosome exactly one gene	

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Fundamental terms and meaning (2)

notion	biology	computer science
phenotype	physical appearance of a living organism	implementation of a solution candidate
genotype	genetic constitution of a living organism	encoding of a solution candidate
population	set of living organism	bag/multiset of chromosomes
generation	population at a point in time	
reproduction	creating offspring of one or multiple (usually two) (parent) organisms	creating (child) chromosomes from one or multiple (parent) chromosomes
fitness	aptitude/conformity of a living organism	aptitude/quality of a solution candidate
	determines chances of survival and reproduction	

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Ingredients of an evolutionary algorithm (1)

- Encoding for the solution candidates
  - highly problem-specific
  - no general rules
- A method to create an initial population
  - commonly created as simple random sequences
  - depending on the chosen encoding
- Evaluation function (fitness function) to evaluate the individuals
  - represents environment and assess quality of individuals
  - often: identical to the function to optimize
  - may also contain additional elements (e.g. constraints)

Ingredients of an evolutionary algorithm (2)

- Selection method related to the fitness function
  - chooses parental individuals to create offspring
  - selects individuals transferred to the next generation without change
- A set of genetic operators to modify chromosomes
  - Mutation — randomly changes of individual genes
  - Crossover — recombination of chromosomes
    - better: “crossing over” (meiosis-process, cell division phase)
    - chromosomes are dissipated and assembled cross-over
- Various parameters (population size, mutation probability, etc.)
- Termination criterion
  - user-specified number of generations

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Formal definition: Decoding function

- For every optimization problem: different representations of solution candidates
  - EA seperates space Ω (so called phenotype) from representation of the solution candidate in individual (so called genotype $\mathcal{G}$)
  - Fitness function f is defined on Ω
  - Mutation und Recombination is defined on $\mathcal{G}$
  - For evaluation: transformation of genotype represented individual in Ω

- Definition: Decoding function

  A decoding function dec : $\mathcal{G} \longrightarrow \Omega$ is a transformation of a genotype to the phenotype Ω

Formal definitions: Individual

- An individual contains in general:
  1) genotype $A.G \in \mathcal{G}$ of an individual A
  2) additional information or strategy parameters $A.S \in Z$
     - e.g. parameter settings for genetic operators
     - space Z of all possible additional information
     - $A.S$ as well as $A.G$ are modifiable by operators
  3) quality or fitness $A.F \epsilon \mathbb{R}$

- Definition: Individual
  - An individual A is a tuple $(A.G, A.S, A.F)$ containing the solution candidate (genotype $A.G \in \mathcal{G}$), the optional additional information $A.S \in Z$ and the quality assessment $A.F = f\left(\mathrm{dec}(A.G)\right) \epsilon \mathbb{R}$

Formal definitions: Genetic operators

- Genetic changes
  - Modification in the genome
  - Ξ („X_i“): set of all states of the random number generator
  - no definition of the change of the actual state $\xi \in \Xi$

- Definition: Genetic operators
  - A mutation operator (which is applied on a $\mathcal{G}$-encoded optimization problem and Z) is defined by the mapping

$$\mathrm{Mut}^{\xi}: G \times Z \to \mathrm{G} \times \mathrm{Z}$$

  - A recombination operator with $\mathrm{r} \geq 2$ parents and $s \geq 1$ offspring $(r, s \in \mathbb{N})$ is defined by the mapping

$$\mathrm{Rek}^{\xi}: (G \times Z)^r \to (G \times Z)^s$$

Formal definitions: Selection operator

- Selection
  - Input: population of *r* individuals, whereas *s* are chosen
  - Selection is changing/creating no new individuals
  - Selection defines indices of individuals only by fitness

- Definition: Selection operator
  - $P = \left\langle A^{(1)}, \ldots, A^{(r)} \right\rangle$:

$$\mathrm{Sel}^{\xi} : (\mathcal{G} \times \mathcal{Z} \times \mathbb{R})^r \to (\mathcal{G} \times \mathcal{Z} \times \mathbb{R})^r$$
$$\left\langle A^{(i)} \right\rangle_{1 \le i \le r} \mapsto \left\langle A^{(\mathrm{IS}^{\xi}(c_1,\ldots,c_r)_k)} \right\rangle_{1 \le k \le s} \text{ mit } A^{(i)} = (a_i, b_i, c_i)$$

  - $\mathrm{IS}^{\xi} : \mathbb{R}^r \to \{1, \ldots, r\}^s$ ‹-selection has the shape

Simple example for a selection operator

- Parental population consists of individuals $A^{(1)}, A^{(2)}, \ldots, A^{(5)}$
- Related quality assessments of the individuals are given by
  1) $A^{(1)}.F = 2.5$
  2) $A^{(2)}.F = 1.9$
  3) $A^{(3)}.F = 3.7$
  4) $A^{(4)}.F = 4.1$
  5) $A^{(5)}.F = 2.4$
- Selection chooses with $\mathrm{IS}^{\xi}: \mathbb{R}^5 \rightarrow \{1,\ldots,5\}^3$ indices 4, 3 and 1 respectively individuals $A^{(4)}$, $A^{(3)}$ and $A^{(1)}$

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Fundamental Evolutionary Algorithm

- Definition

  A simple evolutionary algorithm on an optimization problem $(\xi, f, \succ)$ is an 8-tuple $(G, \mathrm{dec}, \mathrm{Mut}, \mathrm{Rek}, \mathrm{IS}_{\mathrm{Parents}}, \mathrm{IS}_{\mathrm{Environment}}, \mu, \lambda)$.

  where

  μ describes the amount of individuals of the parental population

  λ defines the offspring per generation

$$\mathsf{Rek} : (\mathcal{G} \times \mathcal{Z})^k \to (\mathcal{G} \times \mathcal{Z})^{k'},$$

$$\mathsf{IS}_{\mathsf{parents}} : \mathbb{R}^\mu \to (1, \ldots, \mu)^{\frac{k}{k'} \cdot \lambda} \qquad \text{with } \frac{k}{k'} \cdot \lambda \in \mathbb{N},$$

$$\mathsf{IS}_{\mathsf{Environment}} : \mathbb{R}^{\mu+\lambda} \to (1, \ldots, \mu + \lambda)^\mu$$

Generic Evolutionary Algorithm

Algorithm 1 General Scheme of an Evolutionary Algorithm

Input: optimization problem $(\xi, f, \succ)$
$t \leftarrow 0$
pop(t) ← create the initial population of size μ
evaluate pop(t)
while not termination criterion pop {
    1 ← select parents of offsprings with size λ from pop(t)
    pop_2 ← create offspring by recombination of pop_1
    pop_3 ← mutate individuals in pop_2
    evaluate pop_3
    $t \leftarrow t + 1$
    pop(t) ← select μ individuals from pop3 , pop($t - 1$)
}
return best individual of pop(t)

Genetic vs. Evolutionary Algorithm

- Genetic algorithm:
  - Encoding: Sequence of ones and zeros

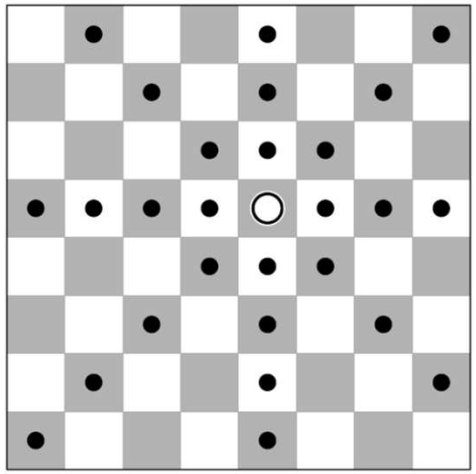
  $\Longrightarrow$ Chromosome is Bitstring (word on alphabet {0, 1})

- Evolutionary algorithm:
  - Encoding: problem-related (Sequence of letters, graphs, formulas, etc.)
  - Genetic operators: defined in relation to encoding and problem

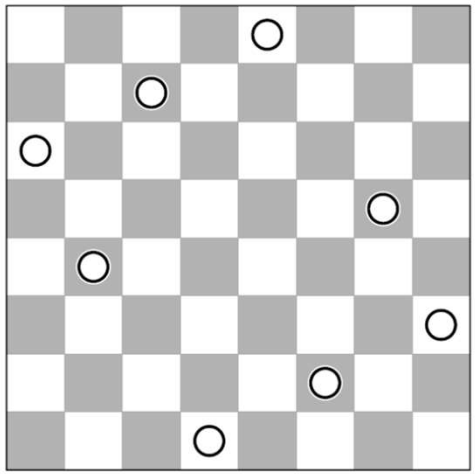
Introduction Example: The *n*-Queens Problem

The *n*-Queens Problem

- Place n queens onto a $n \times n$ chessboard in such a way that no rank (row), no file (column) and no diagonal contains more than one queen
- Or: place queens in such a way that no queen is in the way of another queen



Draw options of a queen



Solution of the n-Queen Problem

Backtracking Solution of the *n*-Queens Problem

1) place queens rank-by-rank bottom-up

   (or column by column from left to right)

2) consider each row as follows

   - place one queen in a rank sequentially from left to right onto the squares of the board
   - for each placement: check if queen collides with queens in lower ranks
   - if not, work on next rank recursively
   - afterwards: shift queen one square rightwards

3) return solution if queen is placed in top line without any collision

© Vincenzo Piuri, Università degli Studi di Milano, Italy

Backtracking Solution of the *n*-Queens Problem

```
int search (int y)
{                                                   /* --- depth first search */
        int x, i, d;                                /* loop variables, buffer */
        int sol = 0;                                /* solution counter */
        if (y >= size) {                            /* if a solution has been found, */
                show(); return 1; }                 /* show it and abort the function */
        for (x = 0; x < size; x++) {                /* traverse fields of the current row */
                for (i = y; --i >= 0; ) {           /* traverse the preceding rows */
                        d = abs(qpos[i] -x);        /* and check for collisions */
                        if ((d == 0) || (d == y-i)) break;
                }                                   /* if there is a colliding queen, */
                if (i >= 0) continue;               /* skip the current field */
                qpos[y] = x;                        /* otherwise place the queen */
                sol += search(y+1);                 /* and search recursively */
        }
        return sol;                                 /* return the number of */
} /* search() */                                    /* solutions found */
```

Analytical Solution

- If only one solution (a placement of queens) is required, calculation of positions for all $n > 3$
  - n mod 2 = 1 ⇒ place 1 queen

    on $(n-1, n-1)$ and $n \leftarrow n-1$
  - n mod 6 ≠ 2 ⇒ place queens

    in the rows $y = 0,\ldots,\frac{n}{2}-1$ in the columns $x = 2y+1$

    in the rows $y = \frac{n}{2},\ldots,n-1$ in the columns $x = 2y-n$
  - n mod 6 = 2 ⇒ place queens

    in the rows $y = 0,\ldots,\frac{n}{2}-1$ in the columns $x = (2y+\frac{n}{2}) \bmod n$

    in the rows $y = \frac{n}{2},\ldots,n-1$ in the columns $x = (2y-\frac{n}{2}+2) \bmod n$

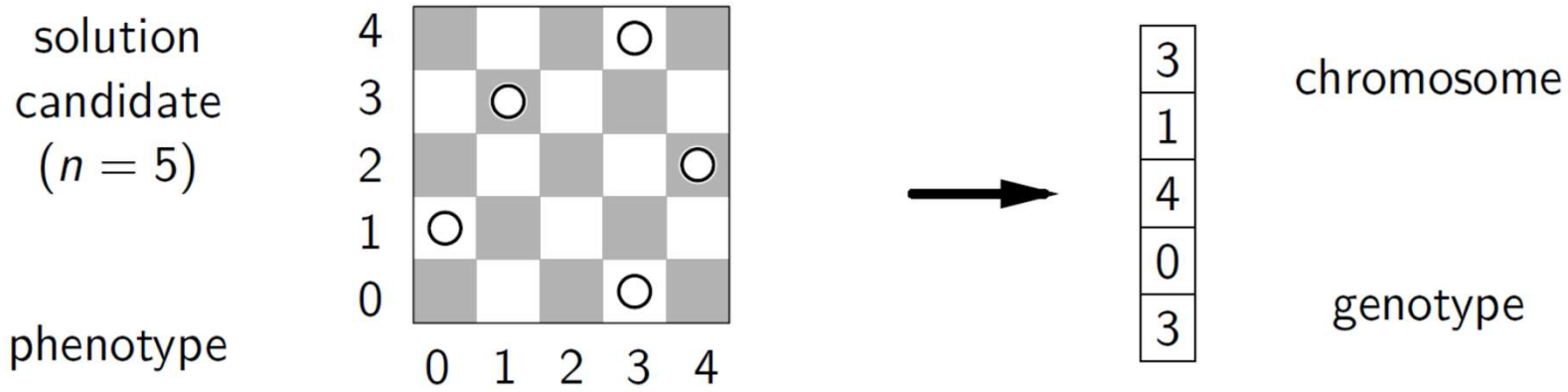
© Vincenzo Piuri, Università degli Studi di Milano, Italy

Evolutionary Algorithm

- Evolves the placement by changing positions of queens to optimize the objective function implementing the constraints on placement

EA: Encoding

- Representation: 1 solution candidate = 1 chromosome with n genes
- Each gene: one rank of the board with n possible alleles
- Value of the gene: position of the queen in corresponding rank



EA: Data structure

- Data type for 1 chromosome, which stores the fitness
- Data type for 1 chromosome with buffer for "intermediary population" and flag for the best individual

```
typedef struct {                /* --- an individual --- */
        int fitness;            /* fitness (number of collisions) */
        int cnt;                /* number of genes (number of rows) */
        int genes[1];           /* genes (queen positions in rows) */
} IND;                          /* (individual) */

typedef struct {                /* --- a population --- */
        int size;               /* number of individuals */
        IND **inds;             /* vector of individuals */
        IND **buf;              /* buffer for individuals */
        IND *best;              /* best individual */
} POP;                          /* (population) */
```

EA: Main loop

- Basic implementation

```
pop_init(pop);                                  /* initialize the population */
        while ((pop_eval(pop) < 0)              /* while no solution found and */
        && (--gencnt >= 0)) {                   /* not all generations computed */
                pop_select(pop, tmsize, elitist);
                pop_cross (pop, frac);          /* select individuals, */
                pop_mutate(pop, prob);          /* do crossover, and */
}                                               /* mutate individuals */
```

- Parameters

gencnt	maximum amount of remaining generations
tmsiz	size of tournament selection
elitist	indicates, if best individual will always be taken
frac	fraction of individuals, which will be submitted by cross-over
prob	mutation probability

EA: Initialize

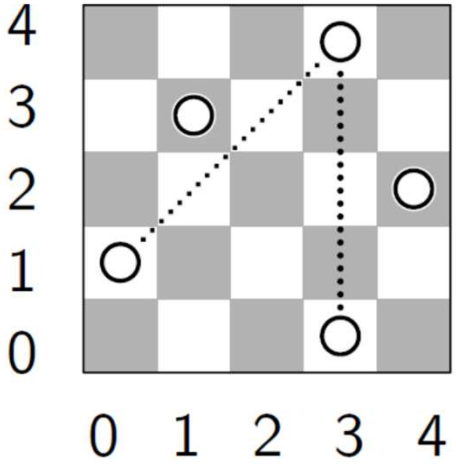
- Initialize individuals and population: create random series of *n* numbers from $\{0, 1, \ldots, n-1\}$

```
void ind_init (IND *ind)
{                                                   /* --- initialize an individual */
        int i;                                      /* loop variable */
        for (i = ind->n; --i >= 0; )                /* initialize the genes randomly */
                ind->genes[i] = (int)(ind->n *drand());
        ind->fitness = 1;                           /* fitness is not known yet */
}                                                   /* ind_init() */


void pop_init (POP *pop)
{                                                   /* --- initialize a population */
        int i;                                      /* loop variable */
        for (i = pop->size; --i >= 0; )
                ind_init(pop->inds[i]);             /* initialize all individuals */
}                                                   /* pop_init() */
```

EA: Evaluation (1)

- Fitness: negated number of columns and diagonals with ≥ 1 queen (negated number due to maximizing fitness)



2 collisions $\rightarrow$ fitness $= -2$

- If queens in 1 column/diagonal ≥ 2: count each pair (easier to implement)
- Fitness-function results immediately in termination criterion: Solution has (highest possible) fitness 0
- Also: termination is guaranteed when maximal generation is reached

© Vincenzo Piuri, Università degli Studi di Milano, Italy

EA: Evaluation (2)

- Count collisions by computation on chromosomes

```
int ind_eval (IND *ind)
{                                                       /* --- evaluate an individual */
        int i, k;                                       /* loop variables */
        int d;                                          /* horizontal distance between queens */
        int n;                                          /* number of collisions */
        if (ind->fitness <= 0)                          /* if the fitness is already known, */
                return ind->fitness;                    /* simply return it */
                for (n = 0, i = ind->n; --i > 0; ) {
                        for (k = i; --k >= 0; ) {       /* traverse all pairs of queens */
                                d = abs(ind->genes[i] -ind->genes[k]);
                                if ((d == 0) || (d == i-k)) n++;
                }                                       /* count the number of pairs of queens */
        }                                               /* in the same column or diagonal */
        return ind->fitness = -n;                       /* return the number of collisions */
}                                                       /* ind_eval() */
```

EA: Evaluation (3)

- Calculation of the fitness of all individuals of the population
- Simultaneously: determination of best individual
- Best individual fitness 0 $\Longrightarrow$ solution is found

```
int pop_eval (POP *pop)
{                                                    /* --- evaluate a population
                                                     */
        int i;                                       /* loop variable */
        IND *best;                                   /* best individual */
        ind_eval(best = pop->inds[0]);
        for (i = pop->size; --i > 0; )
                if (ind_eval(pop->inds[i]) >= best->fitness)
                        best = pop->inds[i];         /* find the best individual */
        pop->best = best;                            /* note the best individual */
        return best->fitness;                        /* and return its fitness */
}                                                    /* pop_eval() */
```

EA: selection of individuals (1)

- Tournament selection
  - Consider tmsize arbitrarily chosen individuals
  - Best (of these) individual "wins" tournament and will be chosen
  - The higher the fitness the better chance to get chosen

```
IND* pop_tmsel (POP *pop, int tmsize)
{                                                    /* --- tournament selection */
      IND *ind, *best;                               /* competing/best individual */
      best = pop->inds[(int)(pop->size *drand())];
      while (--tmsize > 0) {                         /* randomly select tmsize individuals */
            ind = pop->inds[(int)(pop->size *drand())];
            if (ind->fitness > best->fitness) best = ind;
      }                                              /* det. individual with best fitness */
      return best;                                   /* and return this individual */
}                                                    /* pop_tmsel() */
```

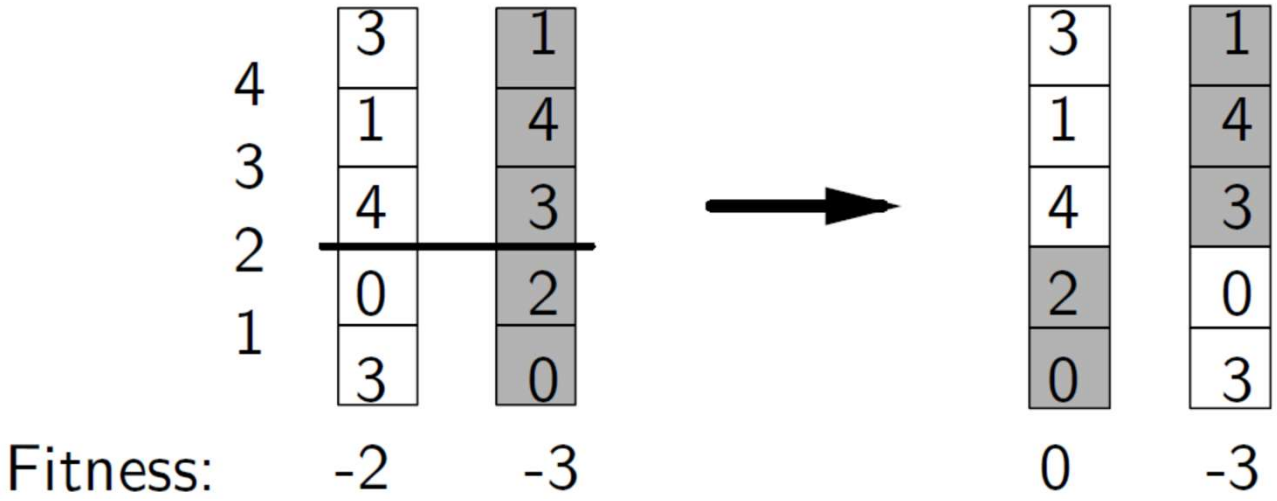
EA: selection of individuals (2)

- Tournament selection for individuals of the next population generation
- Perhaps best individuals will be applied (and not changed)

```
void pop_select (POP *pop, int tmsize, int elitist)
{                                                          /* --- select individuals */
      int i;                                               /* loop variables */
      IND **p;                                             /* exchange buffer */
      i = pop->size;                                       /* select 'popsize' individuals */
      if (elitist)                                         /* preserve the best individual */
            ind_copy(pop->buf[--i], pop->best);
      while (--i >= 0)                                     /* select (other) individuals */
            ind_copy(pop->buf[i], pop_tmsel(pop, tmsize));
      p = pop->inds; pop->inds = pop->buf;
      pop->buf = p;                                        /* set selected individuals */
      pop->best = NULL;                                    /* best individual is not known yet */
}                                                          /* pop_select() */
```

EA: Crossover (1)

- Exchange of a piece of the chromosomes between two individuals
- Here: so called One-Point-Crossover
  - choose cutting line between two genes by random
  - change sequences of genes on one side of the cutting line
  - Example: choose cutting line 2



© Vincenzo Piuri, Università degli Studi di Milano, Italy

EA: Crossover (2)

- Exchange of pieces of the chromosomes between two individuals

```
void ind_cross (IND *ind1, IND *ind2)
{                                                   /* --- crossover of two chromosomes */
      int i;                                        /* loop variable */
      int k;                                        /* gene index of crossover point */
      int t;                                        /* exchange buffer */
      k = (int)(drand() *(ind1->n-1)) +1;           /* choose a crossover point */
      if (k > (ind1->n >> 1)) { i = ind1->n; }
      else { i = k; k = 0; }
      while (--i >= k) {                            /* traverse smaller section */
            t = ind1->genes[i];
            ind1->genes[i] = ind2->genes[i];
            ind2->genes[i] = t;                     /* exchange genes */
}                                                   /* of the chromosomes */
```

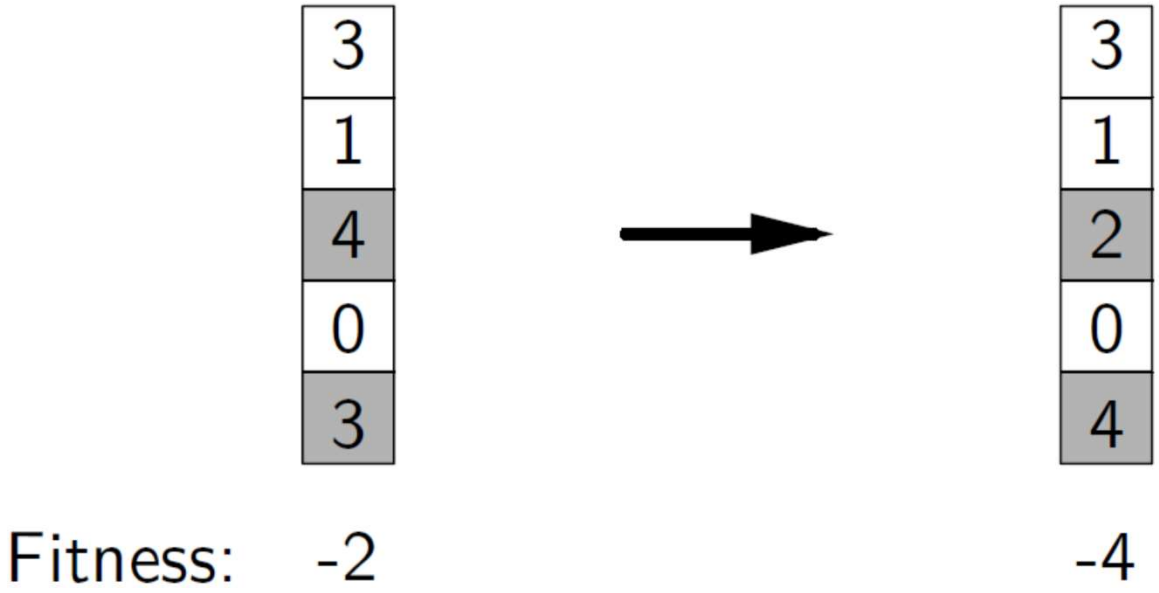
EA: Crossover (3)

- Certain rate of individuals is submitted by crossover
- Include of both crossover-products in new population
- "Parental individuals" are getting lost
- No crossover on best individual (if taken over)

```
void pop_cross (POP *pop, double frac)
{                                                    /* --- crossover in a population */
      int i, k;                                      /* loop variables */
      k = (int)((pop->size -1) *frac) & ~1;
      for (i = 0; i < k; i += 2)                     /* crossover of pairs of individuals */
            ind_cross(pop->inds[i], pop->inds[i+1]);
}                                                    /* pop_cross() */
```

EA: Mutation (1)

- Replacement of randomly chosen genes (changing of alleles)
- Perhaps number of replaced genes is chosen by random (number of replaced genes should be as small as possible)



- Mutations are mostly damaging (decrease the fitness)
- Not existing alleles can be created by mutation

© Vincenzo Piuri, Università degli Studi di Milano, Italy

EA: Mutation (2)

- Decide whether to continue mutating or not
- Best individual (if taken over) will not be submitted by mutation

```
void ind_mutate (IND *ind, double prob)
{                                                          /* --- mutate an individual */
        if (drand() >= prob) return;                       /* det. whether to change individual */
        do ind->genes[(int)(ind->n *drand())] = (int)(ind->n *drand());
        while (drand() < prob);                            /* randomly change random genes */
        ind->fitness = 1;                                  /* fitness is no longer known */
}                                                          /* ind_mutate() */


void pop_mutate (POP *pop, double prob)
{                                                          /* --- mutate a population */
        int i;                                             /* loop variable */
        for (i = pop->size -1; --i >= 0; )
                ind_mutate(pop->inds[i], prob);
}  /* pop_mutate() */                                      /* mutate individuals */
```