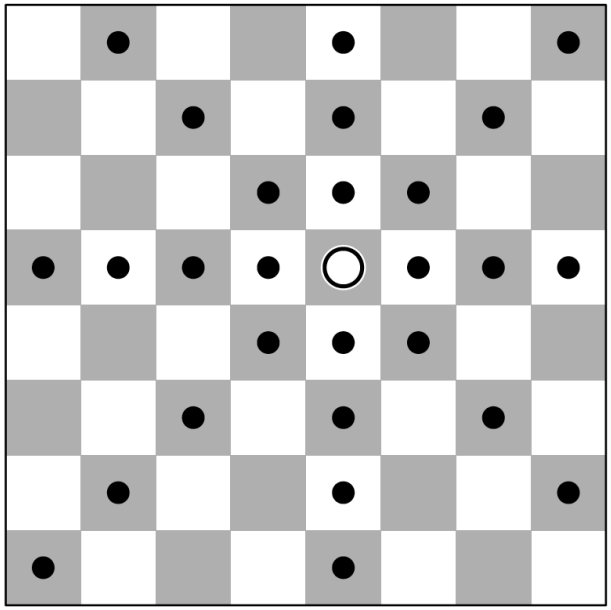
EE496 : COMPUTATIONAL INTELLINGENCE
EA02 : N-QUEEN : AN EA SOLUTION

UGUR HALICI
METU: Department of Electrical and Electronics Engineering (EEE)
METU-Hacettepe U: Neuroscience and Neurotechnology (NSNT)

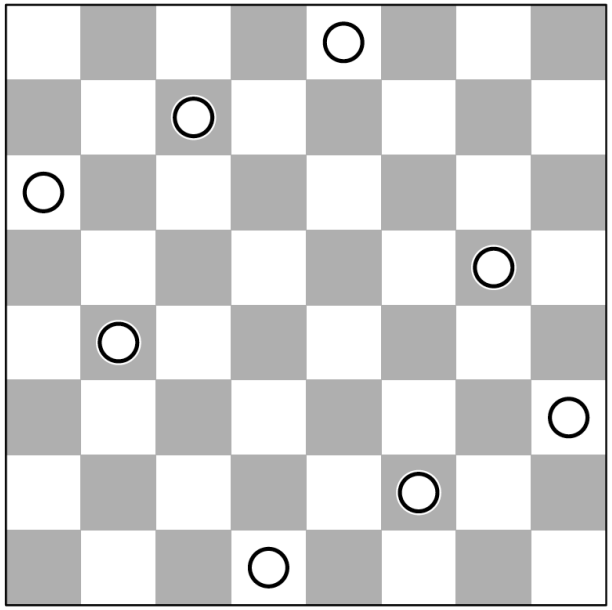
The n-Queens Problem

place n queens onto a n x n chessboard in such a way that no row (rank) , no column (file) 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



A solution for n-queen problem

The n-Queens Problem

Backtracking Solution of the n-Queens Problem

1. place queens row-by-row from bottom to up

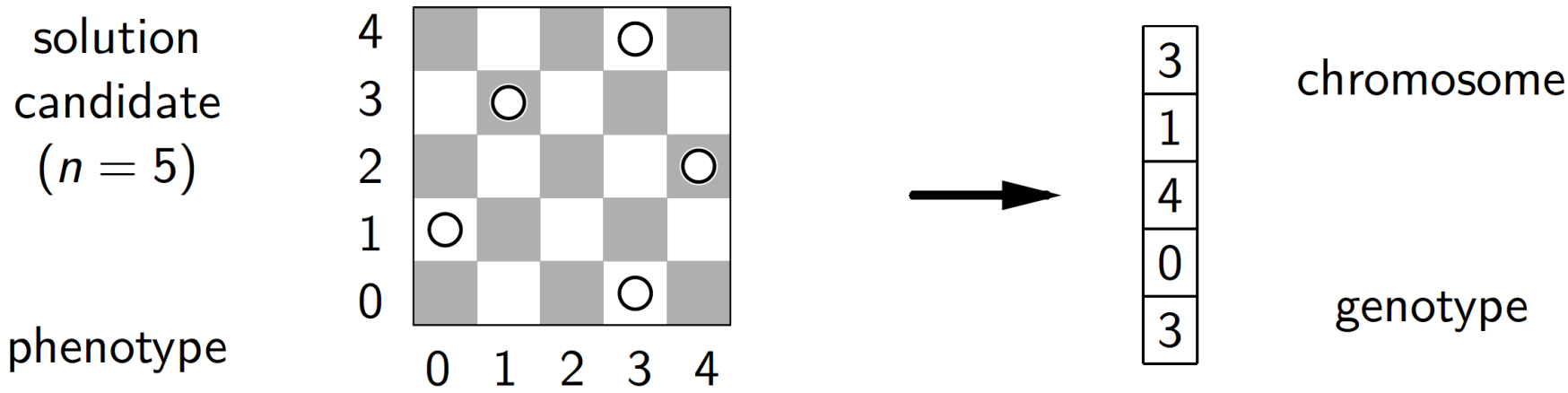
   (or column by column from left to right)

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 rows
   - if not, work on next row recursively
   - afterwards: shift queen one sqare rightwards

3. return solution if queen is placed in top row with no collision

EA: Encoding

- Representation: 1 solution candidate = 1 chromosome with n genes
- each gene: one row of the board with n possibles alleles
- value of the gene: position of the queen in corresponding rank



solution candidates with > 1 queen each rank not permitted

⇒ smaller search space

EA: data structure

- data type for 1 chromosome, which stores the fitness
- data type for 1 chromosome with buffer for “intermediate population“ and flag for the best individual

```
typedef struct {          /* --- an individual --- */
    int fitness;          /* fitness (number of collisions) */
    int n;                /* number of genes (number of rows) */
    int genes[n];         /* 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

shows basic form of an EA:

```
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); /* select individuals (natural selection) */
      pop_cross (pop, frac);            /* do crossover (parental sel. & crossover */
      pop_mutate(pop, prob);            /* and mutate individuals */
}
```

parameters:

gencnt: maximum amount of remaining generations

tmsize: size of tournament selection

elitist: indicates, if best individual will always be taken

frac: fraction of individuals, which will be submitted by cross-over

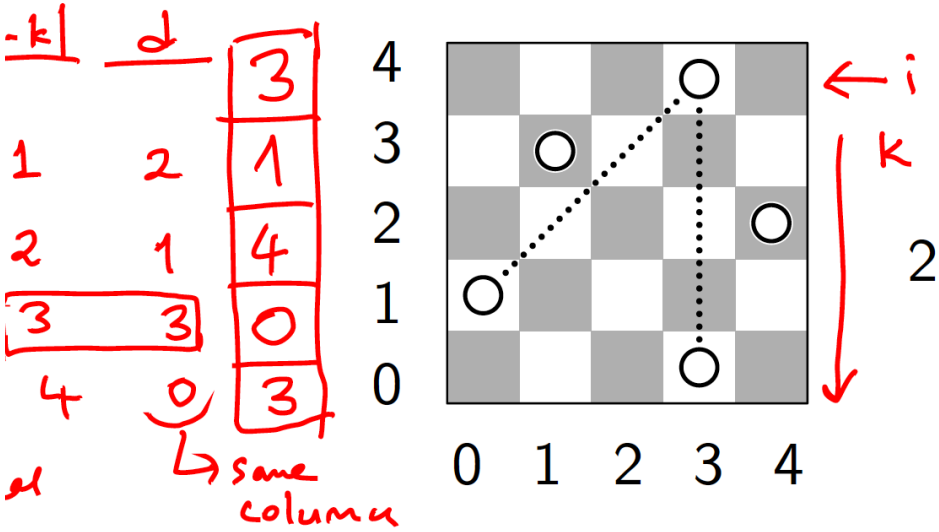
prob: mutation probability

EA: Initialize

```
create random series of n numbers from {0, 1, . . . , n − 1}
/* 0 < drand( ) < 1 , so 0 <= (int) n * drand ( ) < 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

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



- 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

EA Evaluation

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() */
```

same column

same diagonal

↳ if n is large, fitness is poor

EA Evaluation

- calculation of the fitness of all individuals of the population
- simultaneously: determination of best individual
- best individual fitness 0 $\Rightarrow$ solution is found

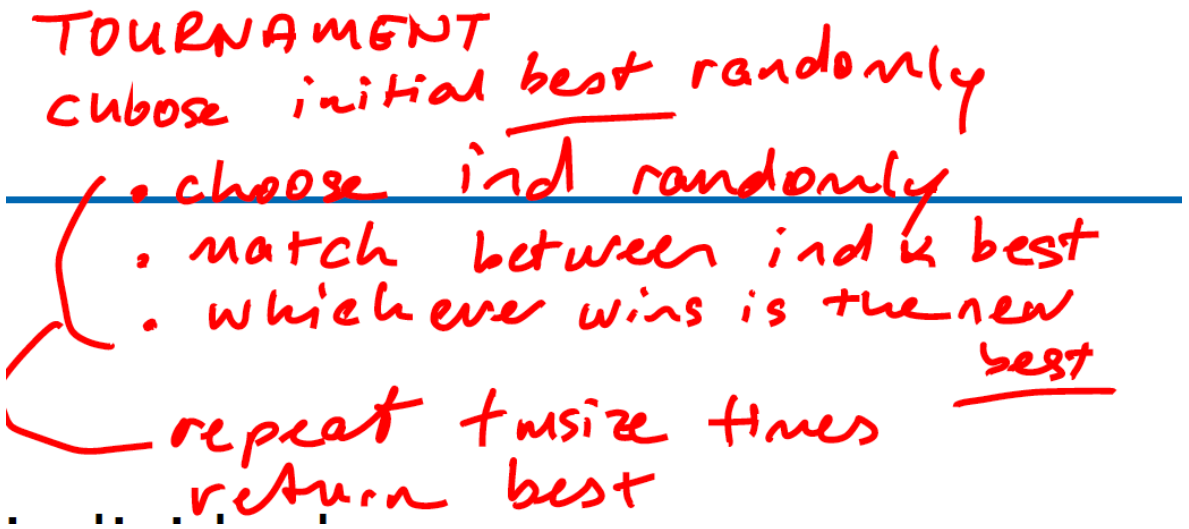
```
int pop_eval (POP *pop)
{                                   /* --- evaluate a population */
  int i;                            /* loop variable */
  IND *best;                        /* best individual */

  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 (for next generation)

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



EA: selection of individuals (for next generation)

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() */
```

repeated tmsize times

/* choose the initial best randomly */

Choose ind randomly

if ind is better it is the new best

in the tournament

EA: selection of individuals

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

```
{                                       /* --- 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() */
```

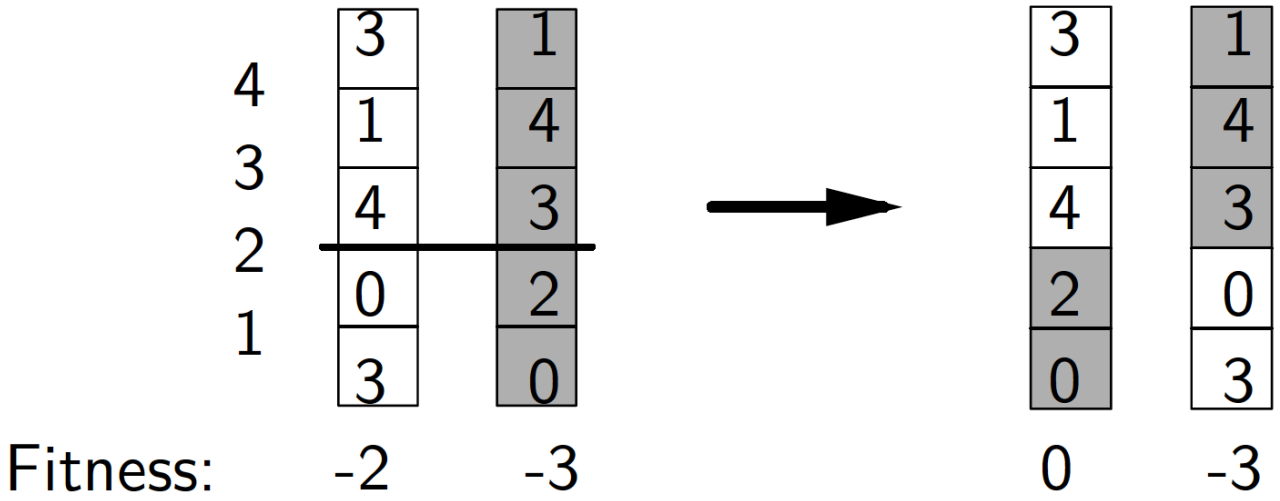
temporary buffer for swap

Swap (pop->inds, pop->buf)

returns the best of the tournament

Crossover

- 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



EA: Crossover

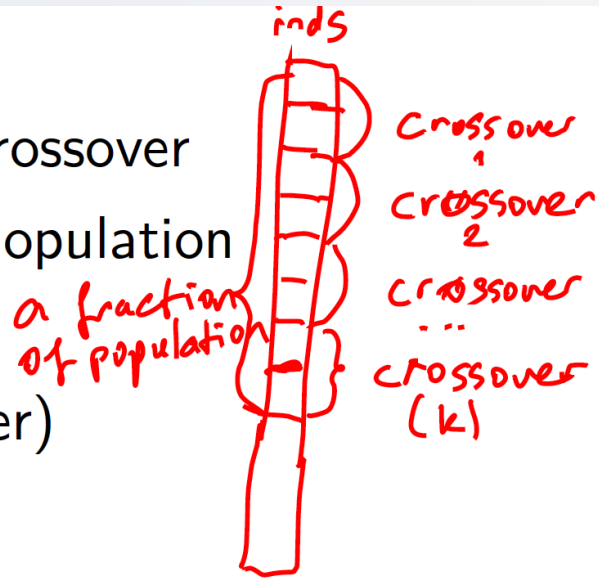
- Exchange of pieces of the chromosomes between two individuals

```
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 */
  ind1->fitness = 1;                /* invalidate the fitness */
  ind2->fitness = 1;                /* of the changed individuals */
}  /* ind_cross() */
```

EA: Crossover

- 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)

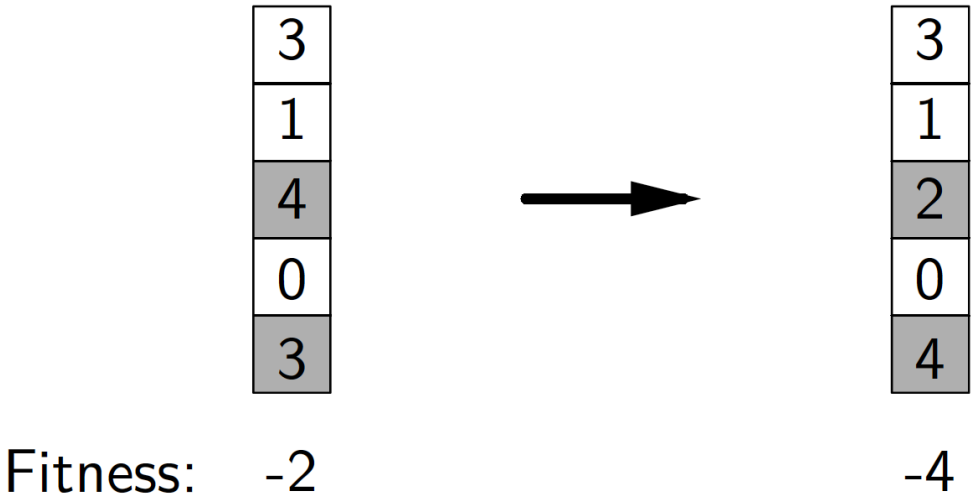



```
void pop_cross (POP *pop, double frac)
{                                   /* --- crossover in a population */
  int i, k;                         /* loop variables */

  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 Mutataion

- 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)
- non existing alleles can be created by mutation

- decide whether to continue mutating or not
- best individual (if taken over) won't be submitted to 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 */
```

no mutation

Loop

decide to continue randomly

mutate a random gene

mutates some of individuals