## Genetic algorithm(s)

- Developed: USA in the 1970's
- Early names: J. Holland, K. DeJong, D. Goldberg
- Typically applied to:
  - discrete optimization
- Attributed features:
  - not too fast
  - good solver for combinatorial problems
- Special:
  - many variants, e.g., reproduction models, operators
  - formerly: *the* GA, nowdays: *a* GA, GA*s*



## Genetic algorithms

- The simple genetic algorithm
- Other GAs by different:
  - Representations
  - Mutations
  - Crossovers
  - Selection mechanisms



## Simple genetic algorithm (SGA)

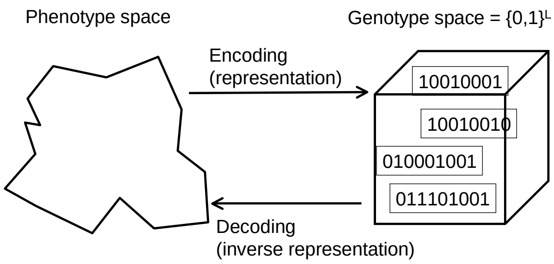
- The first GA, being "standard" for many years
- Formerly quoted as THE genetic algorithm
- Nowdays one uses the term A genetic algorithm, the SGA is just one of them

Here we consider

- representation
- variation operators (crossover, mutation)
- selection
- reproduction cycle: generational model
- $x^2$ example



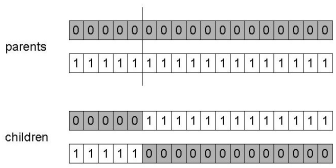
## Representation






## Genetic operators: 1-point crossover

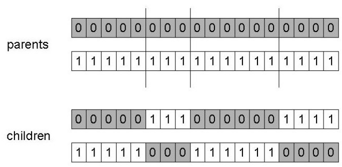
- Choose a random point on the two parents
- Split parents at this crossover point
- Create children by exchanging tails






## Genetic operators: n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents



## Genetic operators: uniform crossover

- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child

parents
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

children
0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1
1 0 1 1 0 0 0 0 1 1 1 0 1 0 0 1 1 0



## Genetic operators: mutation

- Alter each gene with a probability $p_m$
- $p_m$ is called the mutation rate

parent
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

child
0 1 0 0 1 0 1 1 0 0 0 1 0 1 1 0 0 1



## Crossover OR mutation?

- Decade long debate: which one is better / necessary / main-background
- Answer (at least, rather wide agreement):
  - it depends on the problem, but
  - in general, it is good to have both
  - both have another role
  - mutation-only-EA is possible, xover-only-EA would not work



## Crossover OR mutation? (cont'd)

Exploration: Discovering promising areas in the search space, i.e. gaining information on the problem

Exploitation: Optimising within a promising area, i.e. using information

There is co-operation AND competition between them

Crossover is explorative, it makes a *big* jump to an area somewhere "in between" two (parent) areas

Mutation is exploitative, it creates random *small* diversions, thereby staying near (i.e., in the area of ) the parent



## Crossover OR mutation? (cont'd)

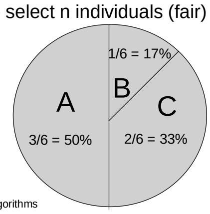
- Only crossover can combine information from two parents
- Only mutation can introduce new information (alleles)
- Crossover does not change the allele frequencies of the population (thought experiment: 50% 0's on first bit in the population, ?% after performing *n* crossovers)
- To hit the optimum you often need a 'lucky' mutation.



## Selection

- Main idea: better individuals get higher chance
- 2ndary idea: chances proportional to fitness
- Implementation: roulette wheel technique
  - Assign to each individual a part of the roulette wheel ("unfair": size proportional to its fitness)
  - Spin the wheel n times to select n individuals (fair)

fitness(A) = 3
fitness(B) = 1
fitness(C) = 2



## Generational GA reproduction cycle

1. Select parents for the mating pool (size of mating pool = population size)
2. Shuffle the mating pool
3. For each consecutive pair apply crossover with probability $p_c$
4. For each new-born apply mutation (bit-flip with probability $p_m$)
5. Replace the whole population by the resulting mating pool



## Generational GA reproduction cycle

| Generation t | Mating pool | Generation t+1 |
|---|---|---|
| string1 | string2 | $child_1$(2,4) |
| string2 | string4 | mut($child_2$(2,4)) |
| string3 | string2 | string2 |
| string4 | string1 | mut(string1) |
| … | … | … |
| | | |

Notes:
• Offspring can be: clone, pure mutant, pure crossing, mutated crossing
• Generational replacement: whole population deleted/replaced
To be discussed: no survival of the fittest here?



## An example after Goldberg '89 (1)

- Simple problem: max $x^2$ over {0,1,…,31}
- GA approach:
  - Representation: binary code, e.g. 01101 ↔ 13
  - Population size: 4
  - 1-point xover, no mutation (just an example!)
  - Roulette wheel selection
  - Random initialisation
- One generational cycle with the hand shown



## An example after Goldberg '89 (2)

| String number | Initial population | $x$ value | $f(x)$ | pselect$_i$ | Expected count | Actual count |
|---|---|---|---|---|---|---|
| | (Randomly generated) | (Unsigned integer) | $(x^2)$ | $\left(\frac{f_i}{\Sigma f}\right)$ | $\left(\frac{f_i}{\bar{f}}\right)$ | (From roulette wheel) |
| 1 | 0 1 1 0 1 | 13 | 169 | 0.14 | 0.58 | 1 |
| 2 | 1 1 0 0 0 | 24 | 576 | 0.49 | 1.97 | 2 |
| 3 | 0 1 0 0 0 | 8 | 64 | 0.06 | 0.22 | 0 |
| 4 | 1 0 0 1 1 | 19 | 361 | 0.31 | 1.23 | 1 |
| Sum | | | 1170 | 1.00 | 4.00 | 4.0 |
| Average | | | 293 | 0.25 | 1.00 | 1.0 |
| Max | | | 576 | 0.49 | 1.97 | 2.0 |



## An example after Goldberg '89 (3)

| Mating pool after reproduction | Mate | Crossover site | New population | $x$ Value | $f(x)$ |
|---|---|---|---|---|---|
| (Cross site shown) | (Randomly selected) | (Randomly selected) | | | $(x^2)$ |
| 0 1 1 0 \| 1 | 2 | 4 | 0 1 1 0 0 | 12 | 144 |
| 1 1 0 0 \| 0 | 1 | 4 | 1 1 0 0 1 | 25 | 625 |
| 1 1 \| 0 0 0 | 4 | 2 | 1 1 0 1 1 | 27 | 729 |
| 1 0 \| 0 1 1 | 3 | 2 | 1 0 0 0 0 | 16 | 256 |
| | | | | | 1754 |
| | | | | | 439 |
| | | | | | 729 |



## The simple GA

- Has been subject of many (early) studies
- Is often used as benchmark for novel GAs
- Shows many shortcomings, e.g.
  - Representation is too restrictive
  - Mutation & crossovers only applicable for bit-string & integer representations
  - Selection mechanism sensitive for converging populations with close fitness values
  - Generational population model can be improved with explicit survivor selection

## Other representations

Variations on standard bit string encoding:

- Gray coding

Three types of non-standard representation:

- Floating point variables
- Order based (permutations)
- Trees (see at Genetic Programming)



## Gray coding

Hamming distance (HD) of two bit strings a, b $\in$ $\{0,1\}^L$

$$HD(a,b) = \sum_{i=1}^{L} | a_i - b_i |$$

= # different bits
= # 1point mutations needed to change a into b

Problem with standard coding:
consecutive integers are mapped on strings with Hamming distance > 1, e.g.: 5=101, 6=110

Disadvantage from GA point of view:
small genotypic changes $\neq$ small phenotypic changes



## Pseudo code for Gray coding 1

Bit string $b_1, \ldots, b_m$ transformed into Gray code $g_1, \ldots, g_m$

```
int[] binaryToGray (int[] b) {
   g[1] = b[1];
   for (k=2; k<=m; k++) {
       g[k] = xor(b[k-1], b[k]);
   }
return g;
}
```



## Pseudo code for Gray coding 2

Gray code $g_1, \ldots, g_m$ transformed into bit string $b_1, \ldots, b_m$

```
int[]grayToBinary (int[] g) {
   value = g[1];
   b[1] = value;
   for (k=2; k<=m; k++) {
       if (g[k] == 1)
              value = !value;
   b[k] = value;
   }
return b;
}
```



## Illustration of Gray coding for m = 3

Gray coding: consecutive integers mapped on strings with HD =1

Advantage in GAs: small genotypic changes lead to small phenotypic changes (smooth mapping)

| Integer | Standard | Gray |
|---|---|---|
| 0 | 000 | 000 |
| 1 | 001 | 001 |
| 2 | 010 | 011 |
| 3 | 011 | 010 |
| 4 | 100 | 110 |
| 5 | 101 | 111 |
| 6 | 110 | 101 |
| 7 | 111 | 100 |
| neighbours | H.D. = 3 | H.D. = 1 |

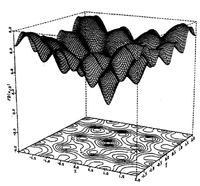


## Real valued problems

- Many problems occur as real valued problems, e.g. continuous parameter optimisation $f : \mathbb{R}^n \rightarrow \mathbb{R}$
- Illustration: Ackley's function (often used in EC)

$$f(\overline{x}) = -c_1 \cdot exp\left(-c_2 \cdot \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - exp\left(\frac{1}{n} \cdot \sum_{i=1}^{n} cos(c_3 \cdot x_i)\right) + c_1 + 1$$

$c_1 = 20,\ c_2 = 0.2,\ c_3 = 2\pi$



## Real valued problems

Options for solving real valued problems with GAs:

- Mapping real values on bit strings
- Mapping real values on floating point variables



## Mapping real values on bit strings

$z \in [x,y] \subseteq \mathbb{R}$ represented by $\{a_1,\ldots,a_L\} \in \{0,1\}^L$

- $[x,y] \rightarrow \{0,1\}^L$ must be invertible (one phenotype per genotype)
- $\Gamma: \{0,1\}^L \rightarrow [x,y]$ defines the representation

$$\Gamma(a_1,\ldots,a_L) = x + \frac{y-x}{2^L-1} \cdot \left(\sum_{j=0}^{L-1} a_{L-j} \cdot 2^j\right) \in [x,y]$$

- Only $2^L$ values out of infinite are represented
- L determines possible maximum precision of solution
- High precision → long chromosomes (slow evolution)



## Mapping real values on floating point variables

- Precision is an implicit choice (depends on computer)
- Genotype: vector $(x_1, \ldots, x_k)$ with $x_i \in \mathbb{R}$
- New genetic operators might be needed
  - Old bit-flip mutation does not work
  - Old crossovers (n-point, uniform) do work
  - New crossovers can be invented to utilize possibilities of new representation better



## Genetic operators for real valued GAs

- Arithmetical crossovers based on averaging corresponding genes from different parents
  - Single arithmetic crossover
  - Whole arithmetic crossover
  - Simple crossover
- Mutation: random new value between some upper and lower bound
  - Uniform mutation
  - Non-uniform mutation



## Single arithmetic crossover

- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- $child_1$ is:

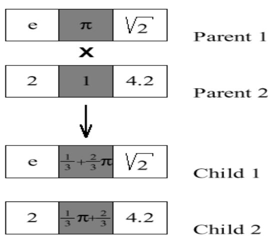
$$\left\langle x_1, \ldots, x_k, a \cdot y_k + (1-a) \cdot x_k, \ldots, x_n \right\rangle$$

The parameter *a* can

- be constant: uniform arithmetical crossover
- vary (e.g. depend on the age of the population): non-uniform crossover



## Single arithmetic crossover illustrated



## Whole arithmetic crossover

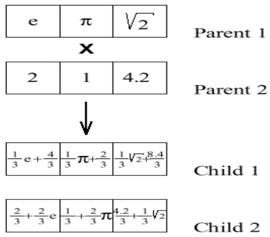
- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- $child_1$ is:

$$a \cdot \bar{x} + (1-a) \cdot \bar{y}$$

The parameter *a* can
- be constant: uniform arithmetical crossover
- vary (e.g. depend on the age of the population): non-uniform crossover



## Whole arithmetic crossover illustrated






## Simple crossover

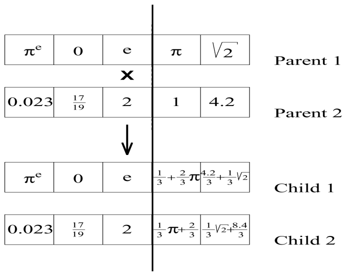
- Parents: $\langle x_1,\ldots,x_n \rangle$ and $\langle y_1,\ldots,y_n \rangle$
- $child_1$ is:

$$\left\langle x_1, ..., x_k, a \cdot y_{k+1} + (1-a) \cdot x_{k+1}, ..., \mathrm{a} \cdot \mathrm{y}_{\mathrm{n}} + (1-a) \cdot x_n \right\rangle$$

The parameter *a* can
- be constant: uniform arithmetical crossover
- vary (e.g. depend on the age of the population): non-uniform crossover



## Simple crossover illustrated






## Floating point mutations 1

General scheme of floating point mutations

$$\bar{x} = \langle x_1, \ ..., x_l \rangle \rightarrow \bar{x}' = \langle x_1', ..., x_l' \rangle$$
$$x_i, x_i' \in [LB_i, UB_i]$$

Uniform mutation:

$$x_i' \text{ drawn randomly (uniform) from } [LB_i, UB_i]$$



## Floating point mutations 2

Non-uniform mutation:
- *t* is the number of the current generation
- *T* is the maximum generation number

$$x_i' = \begin{cases} x_i + \Delta(t, UB_i - x_i) & \text{if a random digit is } 0 \\ x_i - \Delta(t, x_i - LB_i) & \text{if a random digit is } 1 \end{cases}$$

$$\Delta(t, y) = y \cdot (1 - r^{(1-\frac{t}{T})^b})$$

- $r \in [0,1]$ is a random number and
- b is a parameter (b = 5 used)
- if *t* increases then the chance for a small $\Delta(t,y)$ increases

## Bit vs. float: experimental comparison (1)

- After Michalewicz'96
- Experiments on one single problem only – just to illustrate
- Function to be minimised (dynamic control problem):

$$f(\bar{u}) = \left( x_N^2 + \sum_{k=0}^{N-1} (x_k^2 + u_k^2) \right)$$

With:
- $x_0 = 100$
- $x_{k+1} = x_k + u_k$ (k = 0, 1, …, N - 1)
- N = 45
- $u_i \in [-200, 200] \subset \mathbb{R}$



## Bit vs. float: experimental comparison (2)

Representation:
- Floating point: vector of 45 floats ($\bar{u}$)
- Bit string: 30 bits per variable ⇒ 1350 bits

Experimental setup:

| | |
|---|---|
| Averaged over | 10 independent runs |
| # function evaluations | 20.000 |
| Crossover rate ($p_c$) | constant 0.25 |
| Mutation rate ($p_m$) | varying (see table) |
| Population size | 60 |

Mutation rate ($p_m$) that determines the prob. of chrom. update

| Repr. | Probability of chromosome's update 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
|---|---|---|---|---|---|
| Bit, $p_m$ | 0.00047 | 0.00068 | 0.00098 | 0.0015 | 0.0021 |
| Float, $p_m$ | 0.014 | 0.012 | 0.03 | 0.045 | 0.061 |



## Bit vs. float: experimental comparison (3)

Mutation rate ($p_m$) that determines the prob. of chrom. update

| Repr. | Probability of chromosome's update 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
|---|---|---|---|---|---|
| Bit, $p_m$ | 0.00047 | 0.00068 | 0.00098 | 0.0015 | 0.0021 |
| Float, $p_m$ | 0.014 | 0.012 | 0.03 | 0.045 | 0.061 |

Results with 1-point xover, uniform mutation:

| Repr. | Probability of chromosome's update 0.6 | 0.7 | 0.8 | 0.9 | 0.95 | std. dev. |
|---|---|---|---|---|---|---|
| Bits | 42179 | 46102 | 29290 | 52769 | 30573 | 31212 |
| Floats | 46594 | 41806 | 47454 | 69624 | 82371 | 11275 |

Lowest function value found, averaged over 10 runs



## Bit vs. float: experimental comparison (4)

Results with 1-point xover, non-uniform mutation

| Representation | Probability of chromosome's update 0.8 | 0.9 | standard deviation |
|---|---|---|---|
| Bits | 35265 | 30373 | 40256 |
| Floats | 20561 | 26164 | 2133 |

Lowest function value found, averaged over 10 runs

Non-uniform mutation mechanism adapted for bit representaion



## Bit vs. float: experimental comparison (5)

Result with other operators

New operators
- Bit string representation: multi-point crossover
- Float representation: multi-point arithmetical crossover

Results:

| Repr. | Probability of chromosome's update 0.7 | 0.8 | 0.9 | standard deviation | Best |
|---|---|---|---|---|---|
| Bits | 23814 | 19234 | 27456 | 6078 | 16188.2 |
| Floats | 16248 | 16798 | 16198 | 54 | 16182.1 |

Lowest function value found, averaged over 10 runs



## Bit vs. float: experimental comparison (6)

Time performance (experiments with the new operators)

| Repr. | Number of variables 5 | 15 | 25 | 35 | 45 |
|---|---|---|---|---|---|
| Bits | 1080 | 3123 | 5137 | 7177 | 9221 |
| Floats | 184 | 398 | 611 | 823 | 1072 |

Speed results (CPU time in sec.) for various problem sizes

| Repr. | Number of bits per element 5 | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|---|
| Bits | 4426 | 5355 | 7438 | 9219 | 10981 | 12734 |
| Floats | 1072 (constant) | | | | | |

Speed results for various precisions for bit string representation

## Bit vs. float: experimental comparison (7)

Conclusions about float representation:
- More 'natural' representation (one variable ↔ one gene)
- No Hamming cliffs
- Better solution accuracy (with float specific operators)
- Faster
- More consistent (smaller standard deviation)



## Order based representation

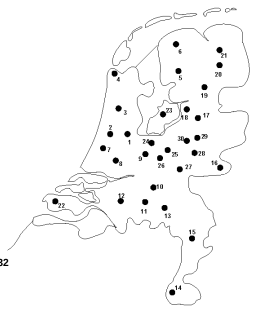
- Ordering/sequencing problems form a special type
- Task is (or can be solved by) arranging some object in a certain order
- Example: sort algorithm
- Example: Travelling Salesman Problem (TSP)



## Order based representation

TSP example
- Problem:
  - Given n cities.
  - Find a complete tour with minimal length.
- Encoding:
  - Label the cities 1, 2, … , *n.*
  - One complete tour is one permutation (e.g. for n =4 [1,2,3,4], [3,4,2,1] are OK)
- Search space is BIG:

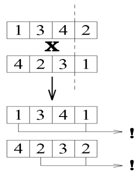
  for 30 cities there are 30! ≈ $10^{32}$ possible tours





## Genetic operators for order based representation

Old crossovers can be be applied, but they can lead to inadmissible genotypes (i.e. without corresponding phenotype).




Example: children of two permutations are not permutations



## Genetic operators for order based representation

Some mutation operators for order based representations:
- swap two alleles
- shift a couple of alleles (with wrapping around at the end)
- invert a substring

Some crossover operators for order based representations:
- order1 (treated here)
- order2
- pmx: partially mapped crossover (treated here)
- cycle (treated here)
- position
- edge crossover (special for TSPs)

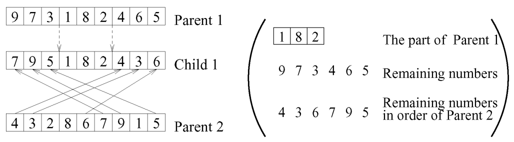


## Order 1 crossover

Informal procedure:

1. Choose an arbitrary part from the first parent.
2. Copy this part to the first child.
3. Copy the numbers that are not in the first part, to the first child:
   - starting right from cut point of the copied part,
   - using the order of the second parent
   - and wrapping around at the end.
4. Analogous for the second child, with parent roles reversed.

## Order 1 crossover example





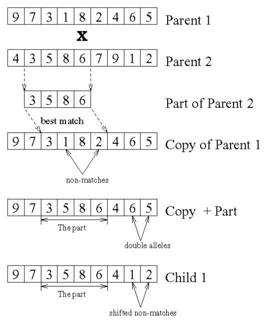

## Partially Mapped Crossover (PMX)

Informal procedure:

1. Make a copy of the first parent.
2. Choose an arbitrary part from the second parent.
3. Map this part on the copy, maximising matches.
4. Non-matching alleles of the copy are shifted to the left (replacing double alleles).
5. Analogous for the second child, with parent roles reversed.



## Partially Mapped Crossover (PMX) example






## Cycle crossover

**Basic idea**:
Each allele comes from one parent *together with its position*.

Informal procedure:

1. Make a cycle of alleles from P1 in the following way.
   (a) Start with the first allele of P1.
   (b) Look at the allele at the *same position* in P2.
   (c) Go to the position with the *same allele* in P1.
   (d) Add this allele to the cycle.
   (e) Repeat step b through d until you arrive at the first allele of P1.
2. Put the alleles of the cycle in the first child on the positions they have in the first parent.



## Genetic operators (9)

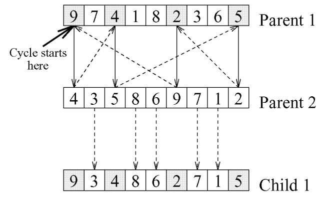
Informal procedure (coninued):

3. Fill the rest of the positions with the alleles in the corresponding positions at the second parent.

4. Analogous for the second child, with parent roles reversed.



## Cycle crossover example




Cycle is 9 4 5 2